PeAR: Efficient Binary-Only Fuzzing Through Static Binary Rewriting

Alvin Charles
School of Computing
The Australian National University
Canberra, ACT, Australia

Adrian Herrera
School of Computing
The Australian National University
Canberra, ACT, Australia

Peter Oslington
School of Computing
The Australian National University
Canberra, ACT, Australia

Alwen Tiu
School of Computing
The Australian National University
Canberra, ACT, Australia

ABSTRACT

Binary-only fuzzing is a key technique for finding bugs in close-source software. Without access to source code, the fuzzer must rely on static or dynamic binary instrumentation for coverage guidance. Unfortunately, static instrumentation is often unscalable and unsound, while dynamic instrumentation is often slow. We propose PeAR to tackle the challenges associated with using static binary instrumentation (SBI) for fuzzing. PeAR uses an off-the-shelf SBI framework to statically insert fuzzing instrumentation. Unlike existing SBI-based fuzzers, PeAR supports modern fuzzer features—namely, deferred initialization, persistent mode, and shared memory fuzzing—that radically improve fuzzer throughput.

We evaluate PeAR over 3.5 CPU-yrs of fuzzing on the FuzzBench and Magma benchmark suites and find that PeAR: (i) successfully instruments 83 % of FuzzBench targets (compared to `afl-dyninst`, which instruments 70 % of targets); (ii) outperforms other binary-only fuzzing performance by 29 % and reaches throughput at least 2× higher than the state-of-the-art when using persistent mode; and (iii) attains coverage comparable to compiler-based instrumentation. We find that SBI is a practical technique for binary-only fuzzing, and that modern binary rewriting frameworks can apply complex instrumentation with high granularity and negligible performance compromise.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Fuzzing, Binary rewriting

1 INTRODUCTION

Fuzzing is a popular and effective bug-finding technique. It relies on the ability to quickly generate a large number of inputs and test those inputs against a target program. These inputs are generated to explore corner cases in the target's behavior, which existing tests may not cover and where bugs may lurk. Modern *greybox* fuzzers rely on a feedback loop to determine whether these corner cases are being reached. Feedback is derived from program instrumentation that measures and tracks code coverage, guiding the fuzzer to uncover and explore new target behaviors. This instrumentation is typically injected at compile time. However, there are often cases where source code is unavailable (e.g., when testing close-source software), requiring alternative techniques for tracking coverage.

Binary-only fuzzing—fuzzing when source code is unavailable—relies on either static binary instrumentation (SBI) or dynamic binary instrumentation (DBI) to track code coverage. SBI frameworks apply instrumentation to target binaries by generating new binaries (e.g., via rewriting) containing the desired instrumentation. SBI has a relatively low runtime performance cost (compared to DBI). However, reliably and correctly applying instrumentation statically is challenging, with many SBI frameworks requiring strong assumptions about the target binary [?].

In contrast, DBI applies instrumentation to the target at runtime. DBI techniques include: (i) just-in-time (JIT) recompilation (e.g., Intel Pin [?], Dyninst [?]); (ii) emulation frameworks (e.g., QEMU [?], Icicle [?]); and (iii) probe-based instrumentation (e.g., `kprobes` in the Linux kernel). While applying DBI is simpler (compared to SBI), it comes with significant performance costs (e.g., 10–100× when fuzzing the LAVA-M benchmark suite with AFL [?]). Thus, improving the accuracy and generalizability of SBI has seen a renewed focus.

Despite advances in SBI fuzzing frameworks [? ? ? ?], we found that many frameworks lacked features provided by modern compiler- and DBI-based frameworks. For example, techniques such as deferred initialization, persistent mode, and shared memory fuzzing provide significant performance boosts to fuzzer performance. Unfortunately, none of the existing SBI fuzzing frameworks provide these features. Moreover, we found that most of these frameworks failed to generalize and accurately instrument a large swathe of common fuzz targets. However, recent works [? ?] have shown that achieving highly-accurate static binary rewriting is now possible. Can these advances be used to implement advanced fuzzing techniques in binary-only fuzzers?

We answer this question by developing PeAR, an efficient binary-only fuzzing framework that implements advanced fuzzing techniques. PeAR uses an off-the-shelf static binary rewriter based on the GrammaTech Intermediate Representation for Binaries [?] (GTIRB) to inject fuzzing instrumentation with features and performance comparable to state-of-the-art compiler-based solutions. In summary, we contribute:

- PeAR, an SBI fuzzing framework for injecting modern fuzzing instrumentation across a wide range of targets (Section 3); and

- A comprehensive evaluation (over 3.5 CPU-yrs) of PEAR and three other state-of-the-art SBI fuzzing frameworks (`afl-dyninst` [?], E9AFL [?], and ZAFL [?]) on the FUZZBENCH and Magma benchmark suites (Section 4).

Our results show that PEAR improves fuzzing throughput by up to 29 % over the other three SBI-based fuzzers. Moreover, PEAR significantly outperforms comparable binary-only fuzzers, achieving between 2–31× speedup across our benchmarks, all while achieving comparable levels of code coverage. PEAR is available as an open-source project.[1]

2 BACKGROUND AND RELATED WORK

2.1 Greybox Fuzzing

Fuzzing is an important technique for automated bug discovery. Fuzzers typically require a target program and a set of valid inputs—known as "seeds"—to bootstrap the fuzzer. In a fuzzing campaign, new inputs are generated from these seeds (typically via random mutation) and are tested against the target. If the target crashes, the crashing input is saved for further root-cause analysis post campaign.

Coverage-guided greybox fuzzing extends this idea by instrumenting the target to collect coverage information during program execution. Collected coverage information is then used to further guide input generation. This instrumentation must have low runtime overhead, allowing the fuzzer to maintain a high throughput and maximizing the number of inputs the fuzzer executes during a campaign.

In addition to low-overhead instrumentation, modern greybox fuzzers—popularized by AFL++ [?]—use the following techniques to boost performance and maximize throughput.

Forkserver. Greybox fuzzers must repeatedly execute the target with new inputs. However, spawning a new process for each target is a costly operation. Moreover, time is wasted waiting for the target process to initialize before an input is executed. To avoid this overhead, modern fuzzers use a *forkserver* that ensures the target is only initialized once. Future processes are then efficiently cloned (due to `fork`'s copy-on-write semantics) from the initialized process.

Deferred initialization. Target binaries may have complex initialization procedures that slow down the fuzzer. To combat this, the forkserver can be initialized at a user-specified location. This *deferred initialization* can be used to skip target-specific initialization overheads.

Persistent mode. Rather than spawning a new process per input, *persistent mode* allows one process to execute multiple inputs. This is achieved by wrapping the relevant target code in a loop, with each loop iteration executing a single input. While this increases fuzzer throughput, care must be taken to correctly reset the target's state after executing each input (preventing future inputs from executing in an invalid state).

Shared memory fuzzing. Fuzzers typically send inputs to the target via the filesystem. The target is then expected to open and handle this file. Reading and writing these inputs to the filesystem incurs a performance penalty. However, *shared memory fuzzing* allows the target to read inputs directly from memory, avoiding interaction with the filesystem.

[1] https://github.com/avncharlie/PeAR

2.2 Static Binary Instrumentation

Without access to source code, a fuzzer must rely on static or dynamic binary instrumentation for guidance. While emulation frameworks (e.g., QEMU [?], ICICLE [?]) are commonly used to fuzz targets compiled for different architectures, they suffer from significant reductions in fuzzer throughput.

In contrast, *static binary instrumentation* (SBI) does not suffer from the same throughput reductions. However, SBI is less accurate and reliable, because recovering arbitrary control flow from a binary is undecidable. Unfortunately, this impacts the usability of SBI: ?] compared ten SBI frameworks and found that—despite broad support for AFL++ instrumentation—only four successfully instrumented *any* of the test binaries. ?] undertook a similar study with similar results. These studies highlight the ongoing challenges of SBI.

We summarize and compare five popular SBI fuzzing frameworks in Table 1. E9AFL, `afl-dyninst`, and STOCHFUZZ rewrite target instructions with a *trampoline*, a small snippet of code that redirects control flow to the instrumentation (and ensures control flow returns to the original code). While trampolines are performant, they may not always be possible; e.g., if the target instruction(s) is smaller than the trampoline. `RetroWrite` uses *reassembleable disassembly*, exploiting position independent code (PIC) and relocation data to generate an assembly file that can be instrumented with fuzzer instrumentation. However, not all targets are compiled with PIC. Finally, ZAFL lifts the binary into an intermediate representation (IR), rewrites this IR, and then lowers it back down to native code. While performant, the rewriter may fail if the lifter encounters unsupported instructions.

Table 1 also shows the fuzzing features supported by these five frameworks. While four frameworks support the forkserver model, none of them support the other features described in Section 2.1. In contrast, AFL++'s QEMU mode supports all of these features (albeit with larger performance overheads). This lack of modern fuzzer features—combined with the unreliability of existing SBI frameworks—motivates us to develop PEAR, which we describe in the following section.

3 PEAR

We present a high-level overview of PEAR in Fig. 1. PEAR instruments target binaries using `Ddisasm` [?] and the GrammaTech Intermediate Representation for Binaries [?] (GTIRB). First, `Ddisasm` disassembles the input binary and decodes a superset of possible instructions to create a set of Datalog facts. These facts are then analyzed (e.g., for symbolization and to determine control flow) and refined before being translated to GTIRB. We then use the `gtirb-rewriting` framework [?] to insert coverage instrumentation, the forkserver, and the code necessary to enable deferred initialization, persistent mode, and shared memory fuzzing. The remainder of this section describes these steps in greater depth.

Table 1: Comparison of SBI fuzzing frameworks. We compare frameworks across two dimensions: the rewriter they use, and the fuzzing features they support.

Name	Rewriter		Fuzzing			
	Name	Type	Forkserver	Defer. init.	Pers. mode	Sh. mem. fuzzing
afl-dyninst [?]	Dyninst [?]	Trampoline	✓	✗	✗	✗
E9AFL [?]	E9 [?]	Trampoline	✓	✗	✗	✗
RetroWrite [?]	–	Reassembleable disassembler	✗	✗	✗	✗
StochFuzz [?]	–	Trampoline	✓	✗	✗	✗
ZAFL [?]	Zipr [?]	Direct rewriter	✓	✗	✗	✗
PEAR	Ddisasm [?]	Reassembleable disassembler	✓	✓	✓	✓

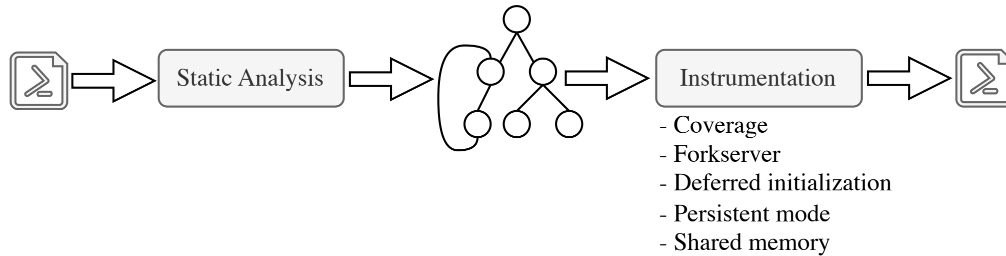


Figure 1: PEAR overview. The target binary is disassembled and statically analyzed before being instrumented and rewritten.

3.1 Static Analysis

PEAR uses reassembleable disassembly [?] to achieve efficient static binary instrumentation. We use Ddisasm—a fast and accurate disassembler implemented using Datalog—to disassemble the target binary and translate it to the GTIRB intermediate representation for further analysis and instrumentation. We choose Ddisasm because of its high success rates in a range of dissasembly and rewriting benchmarks [? ? ?]. Indeed, our evaluation reinforces these findings, with PEAR successfully instrumenting more binaries than other SBI-based tools we compared against. The resulting GTIRB output (a serialized protocol buffer) contains all of the information required for inserting fuzzing instrumentation.

3.2 Instrumentation

PEAR's instrumentation consists of: coverage tracing, deferred initialization, persistent mode, and shared memory fuzzing. This instrumentation supports x64 Linux ELF and Windows PE executables and is compatible with the state-of-the-art AFL++ fuzzer. We describe the design and implementation of this instrumentation in the following sections.

3.2.1 Coverage Tracing. PEAR tracks edge coverage by inserting trampolines to a tracer function at the start of a GTIRB basic block.[2] This approach is inspired by the __afl_maybe_log function from AFL++'s assembly-level instrumentation. Like __afl_maybe_log, PEAR's tracing function takes a single argument: a random integer identifying the current block. Listing 1 shows our trampoline code, where <BLOCK_ID> (Line 5) is the basic block identifier.

```
1  lea    rsp,[rsp-0x98]
2  mov    QWORD PTR [rsp], rdx
3  mov    QWORD PTR [rsp+0x8], rcx
4  mov    QWORD PTR [rsp+0x10], rax
5  mov    rcx, <BLOCK_ID>
6  call   __afl_trace
7  mov    rax, QWORD PTR [rsp+0x10]
8  mov    rcx, QWORD PTR [rsp+0x8]
9  mov    rdx, QWORD PTR [rsp]
10 lea    rsp, [rsp+0x98]
```

Listing 1: Basic block trampoline.

Unlike AFL++—which represents the target as a collection of *intraprocedural* control-flow graphs (CFG)—GTIRB represents the target as a single *interprocedural* CFG, where blocks can be terminated by call instructions (not just branches) and "fallthrough" edges connect the caller block to a successor block (containing the instructions following the call). To avoid unnecessary instrumentation, PEAR does not instrument blocks with an incoming fallthrough edge.

Listing 2 shows the __afl_trace function called by the trampoline. This function follows the "classic" edge coverage approach used by AFL, xor-ing the previous block identifier (__afl_prev_loc) with the current block identifier (passed in rcx). The result of this xor—an edge identifier—is used as a lookup into the coverage map (pointed to by __afl_area_ptr) where an edge counter is incremented.

[2]Unfortunately, gtirb-rewriting does not support inserting instrumentation code using non-live registers at patch locations, which is required for sound insertion of inlined instrumentation.

```
; Save state
lahf
seto    al

; Record path in coverage map
mov     rdx,[rip + __afl_area_ptr]
xor     rcx, QWORD PTR [rip + __afl_prev_loc]
xor     QWORD PTR [rip + __afl_prev_loc], rcx
shr     QWORD PTR [rip + __afl_prev_loc], 1
inc     BYTE PTR [rdx + rcx * 1]
adc     BYTE PTR [rdx + rcx * 1], 0x0

; Restore state
add     al,0x7f
sahf

ret
```

Listing 2: Edge coverage tracing.

3.2.2 *Initialization.*

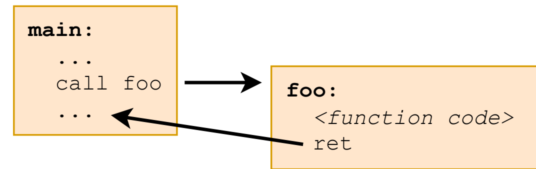
Coverage Map Setup. Listing 3 shows the initialization routine inserted by PEAR.[3] A call to this routine is inserted at the target's entrypoint. This routine sets up a dummy map area (allowing the target to run without a fuzzer) before attaching to the shared memory map (Line 11). Per Section 3.2.1, this shared memory map stores edge hit counts and is used to communicate coverage information to the fuzzer. Notably, `gtirb-rewriting` does not support adding data to the target's `.bss` section, forcing PEAR to use a dynamic memory allocation for the dummy map area (Line 2).

```
void __afl_setup(void) {
  __afl_area_ptr_dummy =  malloc(0x10000);
  if (__afl_area_ptr_dummy == NULL) {
    fprintf(stderr, "ERROR: malloc to setup dummy map
        failed\n");
    _exit(1);
  }

  if (getenv("AFL_DEBUG"))
    __afl_debug = 1;

  __afl_map_shm();
}
```

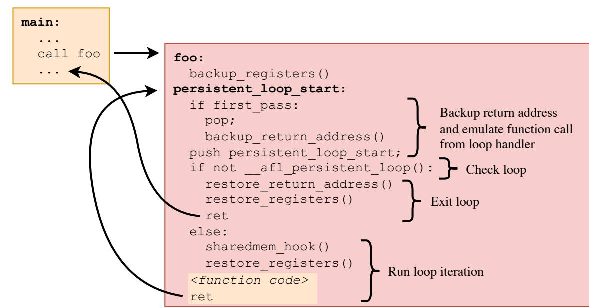
Listing 3: Coverage map initialization.

Deferred Initialization. AFL++ recognizes a deferred forkserver by looking for the `#SIG_AFL_DEFER_FORKSRV#` string in the target binary. PEAR allows the user to specify the forkserver's location (as a function or address). This location defines when the target process has been fully initialized and can be cloned. The forkserver ensures the target is only initialized once, rather than during the execution of every input. Notably, PEAR does not provide a forkserver on Windows due to the lack of support for `fork` operations [?].

3.2.3 Persistent Mode. Figure 2 summarizes PEAR's persistent mode instrumentation, while Listing 4 shows the corresponding assembly code. In Fig. 2a, the function `foo` is selected as a fuzz target. If source code were available, we would wrap the call to `foo` within a loop—e.g., using AFL++'s `__afl_persistent_loop`



(a) Original function call.



(b) After persistent mode application.

Figure 2: Example of PEAR's persistent mode instrumentation.

macro—signaling to the fuzzer that `foo` should be executed with multiple inputs *without* spawning a new proess per input.

Listing 4 shows the persistent mode handler inserted at `foo`'s entrypoint. The handler first saves the current register state before entering the persistent loop. The first time the loop is entered, the caller's return address (in `main`) is saved to memory (Lines 11 to 12). In the loop body, the start address of the loop replaces the caller's return address (Lines 16 to 17), ensuring the persistent loop is always executed. After looping for `<PERSISTENT_MODE_COUNT>` iterations (Lines 20 to 31) the original return address and register state are restored (Lines 34 to 38).

```
; Backup register state (omitted for brevity)

; Start of persistent loop
.Lsetup_loop:
movzx    eax, BYTE PTR first_pass[rip]
test     al, al
je       .Lnot_first_pass

; On first pass, save and overwrite the return address
; (first_pass is set in __afl_persistent_loop)
pop     rax
mov     QWORD PTR [rip + p_mode_ret_addr_backup], rax

.Lnot_first_pass:
; On subsequent passes, set new return address
lea     rax, [rip + .Lsetup_loop]
push    rax

; Check whether to continue loop
mov     rcx, rsp
lea     rsp, [rsp - 0x80]
and     rsp, 0xfffffffffffffff0
push    rcx
push    rcx
mov     edi, <PERSISTENT_MODE_COUNT>
```

[3]This code is written in C, compiled into its own object file, and later linked with the instrumented target.

```
26  call     __afl_persistent_loop
27  pop      rcx
28  mov      rsp, rcx
29
30  test     eax,eax
31  jne      .Lstart_func
32
33  ; Break from loop, restoring the original return address
34  mov      rax, QWORD PTR [rip+p_mode_ret_addr_backup]
35  lea      rsp,[rsp+0x8]
36  push     rax
37  ; Restore register state (omitted for brevity)
38  ret
39
40  ; The function to fuzz
41  .Lstart_func:
```

Listing 4: Persistent mode handler.

3.2.4 Shared Memory Fuzzing. PEAR implements shared memory fuzzing using a shared memory hook, similar to AFL++'s QEMU mode. Listing 5 shows the function signature of this hook.

```
1   struct __attribute__((__packed__)) x86_64_regs {
2     uint64_t rax, rbx, rcx, rdx, rdi, rsi,
3         r8, r9, r10, r11, r12, r13, r14, r15;
4     uint8_t xmm_regs[16][16];
5   };
6
7   void __afl_rewrite_sharedmem_hook(
8     struct x86_64_regs *regs,
9     uint8_t *input_buf,
10    uint32_t input_buf_len
11  );
```

Listing 5: Shared memory hook.

The hook function takes (a) the current state of program registers, (b) a pointer to the current input, and (c) the length of the input. The members of the `x86_64_regs` struct can be modified within the hook function to the values the user wants to set the correspondingly named registers in the program after the hook has been called.

The user can set up the hook function to transfer the current test case to the program by modifying program registers. The specifics of how the hook is implemented depend on how the target processes test data and where in the program the hook function is called. The user can select one of the following options to trigger the hook function:

- at the beginning of the persistent mode loop;
- at a user-selected location, specified through code address or function name; or
- immediately after forkserver initialisation.

If the hook is called at a user-selected location or after forkserver initialisation, the input `x86_64_regs` struct will accurately contain the values of program registers at the point the hook function was called. However, this is not always the case when the hook is called as part of the persistent mode loop. In all cases, the values of the program registers will always be set to the contents of the `x86_64_regs` struct after the hook function has been called.

To store program registers, PEAR inserts a section of global data called `p_mode_register_backup` to the target binary. This location is passed to the shared memory hook as the `x86_64_regs` struct. Two assembly patches are used to backup and restore program registers from this location. Shared memory hooks are created by inserting a patch to save the registers into `p_mode_register_backup`, followed by the assembly patch in Listing 6, and finally restoring the registers from `p_mode_register_backup`. The assembly patch shown in Listing 6 is used to call the shared memory hook function with the necessary arguments. The `__afl_fuzz_ptr` and `__afl_fuzz_len` store a pointer to the shared memory test case and the length of the test case, respectively. These variables are initialized by the `__afl_start_forkserver` function when configuring shared memory fuzzing.

```
1   ; Align stack.
2   mov      rcx, rsp
3   lea      rsp, [rsp - 0x80]
4   and      rsp, 0xfffffffffffffff0
5   push     rcx
6   push     rcx
7
8   ; arg 1: saved registers
9   lea rdi, [rip+p_mode_reg_backup]
10
11  ; arg 2: testcase
12  mov rsi, [rip+__afl_fuzz_ptr]
13
14  ; arg 3: testcase length
15  mov rax, [rip+__afl_fuzz_len]
16  mov edx, [rax]
17  call __afl_rewrite_sharedmem_hook
18
19  pop      rcx
20  mov      rsp, rcx
```

Listing 6: Calling the shared memory hook.

4 EVALUATION

We conduct our evaluation on the FUZZBENCH [?] and Magma [?] benchmark suites, comparing PEAR against other state-of-the-art binary-only fuzzers that use AFL or AFL++ as the backend fuzzer. Note that we did not compare PEAR with StochFuzz [?] as StochFuzz uses a modified AFL to support an iterative stochastic binary rewriting as part of its fuzzing runs, making a straightforward comparison trickier. Our evaluation follows the recommendations made by ?] and focuses on the following metrics:

Code coverage. The amount of code covered by the fuzzer on a particular target.

Speed. The number of iterations achieved (per unit of time) on a particular target. The faster the fuzzer, the quicker it may be to trigger bugs.

Robustness. The ability of a binary-only fuzzer to successfully instrument a given target. SBI-based fuzzers have traditionally failed to instrument complex "real-world" targets.

Bug finding. The number of bugs triggered on a particular target.

The first three metrics are evaluated through Experiment 1 (Section 4.1 and Experiment 2 (Section 4.2), while the last metric is evaluated through Experiment 3 (Section 4.3).

To evaluate the first three metrics, we compare PEAR against the following binary-only fuzzers: AFL++ QEMU [?], E9AFL [?] and `afl-dyninst` [?]. E9AFL requires a shared memory map size of 64 KiB. For `afl-dyninst`, we use the "experimental performance"

mode as recommended by the authors. As a baseline we also included AFL++ with link-time optimization (LTO). We attempted to evaluate ZAFL [?] and RetroWrite [?]. ZAFL was unable to instrument any of the FuzzBench targets, while RetroWrite only supports PIE binaries (while FuzzBench produces non-PIE binaries).

For the bug finding efficacy metric, we compare PeAR against AFL++ source code instrumentation. Since PeAR uses AFL++ as the backend fuzzer, this comparison is really about measuring the consistency of PeAR instrumentation with respect to AFL++ compiler instrumentation. The ability of a fuzzer to find bugs is often a direct consequence of its search heuristics, which sits outside the instrumentation PeAR introduces.

4.1 Experiment 1: Base Performance

In this experiment, we wanted to compare the performance of PeAR and state of the art binary-only fuzzers alongside compiler-level AFL instrumentation.

We attempted to test RetroWrite, ZAFL, E9AFL, `afl-dyninst`, AFL QEMU mode and PeAR as our selected binary-only fuzzers and AFL++ LTO mode, AFL++ LLVM mode and AFL++ GCC mode as our selected AFL compiler-level fuzzers. AFL++ LTO mode uses a different style of edge coverage instrumentation than the other fuzzers. We include it as an upper ceiling for AFL instrumentation but do not consider the coverage it reaches directly comparable to other evaluated fuzzers.

We used an identical build process to build all programs before applying instrumentation using the selected binary-only fuzzing tools. All binaries were built using `clang` and `clang++` with no optimisation and no program sanitisers. We similarly disabled optimisation and sanitisers when building targets with the different AFL++ instrumentation modes. We also explicitly set the coverage map size to 64kB as all selected binary-only fuzzers only operate with this map size.

4.1.1 FuzzBench Integration. We discuss the process of integrating the selected binary-only fuzzers to FuzzBench.

- **RetroWrite**. RetroWrite only supports PIE binaries. However, the FuzzBench target build process produces non-PIE binaries. Due to this, we were unable to integrate RetroWrite into FuzzBench. RetroWrite operates by generating reassemblable assembly from a binary before instrumenting it using AFL++ assembly mode. We decided to infer the potential performance of RetroWrite by attempting to integrate AFL++ assembly mode into FuzzBench.
- **ZAFL**. ZAFL was unable to instrument any FuzzBench benchmark. As such, we excluded it from the experiment.
- **E9AFL**. E9AFL requires the shared memory map size to be 64kB and stops working on the AFL++ commit that increases the default map size. To integrate E9AFL, we set AFL's default map size to 64kB.
- **`afl-dyninst`**. We instrument binaries using `afl-dyninst`'s 'experimental performance mode', which is recommended as a default option by the authors of the tool. We find more benchmarks are instrumented successfully with this mode than without it.
- **PeAR**. The target binaries for the curl_curl_fuzzer_http| and openssl_x509 benchmarks both include the instruction `"andb %bh,(%rcx,%riz,2)"`. The register `%riz` is a pseudo-register that always evaluates to zero. It is used to generate instructions of specific lengths for alignment reasons. PeAR correctly retains this instruction in its output assembly, but the GNU assembler is unable to parse and assemble it. To sidestep this issue, we replace the instruction `"andb %bh,(%rcx,%riz,2)"` with the functionally equivalent instruction `"andb (%rcx), %bh"`.

See Table 2 for the benchmarks the binary-only fuzzers we integrated were able to instrument successfully. Failing benchmark builds fell into two categories:

(1) **Failed instrumentation application**. In these cases, the tool could not produce an instrumented binary at all. We denote this with a double cross.
(2) **Faulty instrumentation**. In these cases, the tool produced a rewritten but faulty binary. These binaries crash on program test cases and seed inputs. We denote this with a single cross.

We exclude the `libjpeg-turbo_libjpeg_turbo_fuzzer` benchmark as this benchmark was broken at the time of conducting our experiment.

4.1.2 Experimental Setup. We ran our experiment on a slightly different set of fuzzers than originally intended for the reasons discussed earlier. We choose E9AFL, `afl-dyninst`, AFL QEMU mode and PeAR as our selected binary-only fuzzers and AFL++ LTO mode, LLVM mode, GCC mode and assembly mode as our baseline AFL compiler-level fuzzers.

After running our experiment, we noticed the AFL++ assembly mode fuzzer did not appear to be working correctly. The fuzzer had extremely low execution speeds and reached lower coverage than all other fuzzers, including AFL QEMU mode. We attempted to diagnose the issue by manually inspecting the binaries built by this fuzzer for any anomalies but did not find any. We checked our FuzzBench integration of AFL++ assembly mode with other researchers who could not find any issues with it. We are unsure why this fuzzer is not integrating with FuzzBench correctly. As we are unable to resolve this issue, we exclude the results of AFL++ assembly mode from our experiment. This unfortunately means we are unable to compare RetroWrite against PeAR.

Limited disk space capped the number of benchmarks we could test in one experiment. The research machine we ran the experiment with had 400GB of available storage space with no way to easily increase disk space. FuzzBench creates Docker containers to run experiment trials, which consume a lot of disk space. Through trial and error, we found that our experiment could safely run with six benchmarks without consuming all available disk space. To select these six benchmarks, we prioritised:

(1) Benchmarks supported by the chosen fuzzers. We made sure to include the only two benchmarks E9AFL supported in our evaluation.
(2) Benchmarks that chosen fuzzers could explore thoroughly. When running tests to set up our experiment, we found our selected fuzzers reached very little coverage on specific benchmarks. This was as we ran our experiment with no

Table 2: FUZZBENCH targets successfully instrumented by our evaluated fuzzers. A single ✗ indicates a faulty instrumented binary, while a double ✗✗ indicates no instrumented binary was generated.

Target	QEMU mode	PEAR	afl-dyninst	E9AFL
bloaty_fuzz_target	✓	✗	✗	✗✗
curl_curl_fuzzer_http	✓	✓	✓	✗✗
freetype2_ftfuzzer	✓	✓	✓	✓
harfbuzz_hb-shape-fuzzer	✓	✓	✗	✗✗
jsoncpp_jsoncpp_fuzzer	✓	✓	✓	✗✗
lcms_cms_transform_fuzzer	✓	✓	✓	✗✗
libpcap_fuzz_both	✓	✓	✗✗	✗✗
libpng_libpng_read_fuzzer	✓	✓	✓	✗✗
libxml2_xml	✓	✓	✗	✗✗
libxslt_xpath	✓	✓	✓	✗✗
mbedtls_fuzz_dtlsclient	✓	✓	✓	✗✗
openh264_decoder_fuzzer	✓	✓	✗	✗✗
openssl_x509	✓	✓	✗	✗✗
openthread_ot-ip6-send-fuzzer	✓	✓	✓	✗✗
php_php-fuzz-parser_0dbedb	✓	✗✗	✓	✗✗
proj4_proj_crs_to_crs_fuzzer	✓	✗	✓	✗✗
re2_fuzzer	✓	✓	✓	✗✗
sqlite3_ossfuzz	✓	✗✗	✓	✗✗
stb_stbi_read_fuzzer	✓	✓	✓	✗✗
systemd_fuzz-link-parser	✓	✓	✓	✗✗
vorbis_decode_fuzzer	✓	✓	✓	✓
woff2_convert_woff2ttf_fuzzer	✓	✓	✓	✗✗
zlib_zlib_uncompress_fuzzer	✓	✓	✓	✗✗
# successful instrumentations	23	19	17	2

seed inputs or dictionaries. We only select benchmarks our selected fuzzers are able to adequately explore.

Using this criteria, we select the following benchmarks to run our experiment with: `libpng_libpng_read_fuzzer`, `libxslt_xpath`, `curl_curl_fuzzer_http`, `vorbis_decode_fuzzer`, `mbedtls_fuzz_dtlsclient`, and `freetype2_ftfuzzer`.

We ran the experiment with no seed inputs and no dictionaries for consistency between all fuzzers. Each fuzzer-benchmark pair ran for 10 trials. Each trial was 24 h. The experiment trials were run on an Intel Xeon Gold 6252 CPU 2.10GHz processor with 187 GiB of memory across 60 concurrent trials. The experiment was run using Ubuntu 20.04.6 LTS running Linux 5.4.0-164-generic. The experiment was run using a fork of FUZZBENCH with our added fuzzer integrations. The version of FUZZBENCH we use was forked on commit `9c6a3950e46d53f4d085f95c977af3836574d49b`.

4.1.3 Results. The full results of this experiment can be seen in Appendix A.

PEAR is the most comparable binary-only fuzzer to AFL compiler-level instrumentation among the binary-only fuzzers we evaluated. In 4 out of 6 benchmarks there is no statistically significant difference between the coverage reached by PEAR and at least one AFL compiler fuzzer. This is also observed with `afl-dyninst` in 2 benchmarks and AFL QEMU mode in 1 benchmark. In no benchmark does any binary-only fuzzer reach statistically significantly higher coverage then any AFL compiler fuzzer.

PEAR and `afl-dyninst` achieve the highest coverages among the binary-only fuzzers we evaluated. PEAR and `afl-dyninst` perform similarly in this regard with no statistically significant difference in reached coverage between them across 5 out of 6 benchmarks. PEAR and `afl-dyninst` consistently outperform AFL QEMU mode in reaching coverage. PEAR achieves statistically higher coverage than AFL QEMU mode in 4 benchmarks, and `afl-dyninst` similarly does so in 3 benchmarks. E9AFL reaches higher coverage than PEAR and `afl-dyninst` in one benchmark. In the other benchmark it supports, it reaches lower coverage than all other binary-only fuzzers, including AFL QEMU mode.

PEAR has the highest performance among binary-only fuzzers in four benchmarks, with `afl-dyninst` consistently following as the next most performant binary fuzzer. In these cases, PEAR outperforms `afl-dyninst` by 6.8 % to 28.9 percent . In two benchmarks, `afl-dyninst` is the most performant binary-only fuzzer with PEAR following as the next most performant binary fuzzer. In both these cases, `afl-dyninst` outperforms PEAR by 6.3 %. In two benchmarks PEAR outperforms at least one AFL compiler fuzzer, while no other binary-only fuzzer outperforms any AFL compiler fuzzer.

4.2 Experiment 2: Maximum Possible Performance

We run a second experiment to evaluate the effectiveness of persistent mode binary-only fuzzing. We reuse the FUZZBENCH integrations we set up for the previous experiment and modify them to enable PEAR and QEMU's persistent mode. In this experiment, we only test binary-only fuzzers.

4.2.1 Experimental Setup. We chose E9AFL, `afl-dyninst`, AFL QEMU mode with persistent mode and PEAR with persistent mode

as our selected fuzzers. We run our experiments on the freetype2_ftfuzzer and the vorbis_decode_fuzzer benchmarks, as these are the only benchmarks supported by all selected fuzzers.

As with our first experiment, we ran this experiment with no seed inputs and no dictionaries, 10 trials per fuzzer-benchmark pair and 24 hours per trial. We run it on the same machine running the same operating system and same version of fuzzbench as our first experiment. We ran this experiment using 90 concurrent trials.

4.2.2 Results. The results of this experiment can be seen in Fig. 12 for the freetype2_ftfuzzer benchmark and in Fig. 13 for the vorbis_decode_fuzzer benchmark. Using persistent mode, PeAR is:

- 2.04 to 2.16 times faster than AFL QEMU mode with persistent mode.
- 3.15 to 3.88 times faster than `afl-dyninst`.
- 12.58 to 31.9 times faster than E9AFL.

This performance boost allows PeAR to reach a statistically significant difference in code coverage between itself and the other evaluated fuzzers. Persistent mode binary-only fuzzing using QEMU mode suffers the performance penalties of emulation. Using persistent mode with SBI-based binary-only fuzzing significantly improves fuzzing performance without this downside.

4.3 Experiment 3: Bug finding efficacy

This experiment is meant to test the bug finding efficacy of PeAR against AFL++. The intention is to check whether the binary instrumentation performed by PeAR affected the effectiveness of afl-fuzz to find bugs, when compared to the source code instrumentation performed by AFL++. Since PeAR uses afl-fuzz to perform the fuzzing loop, the comparison is really about testing to what degree our binary-only instrumentation is faithful to the source code instrumentation of AFL++, in terms of bug finding. It is not expected that the binary-only instrumentation would outperform AFL++, and an optimum result would be for PeAR to achieve a comparable bug finding efficacy to the AFL++.

To allow for comparisons between benchmarks with shared memory enabled and dissabled, PeAR was setup as two different fuzzers under Magma. The base PeAR was benchmarked, as well as PeAR with shared memory mode enabled, which is referred to as PeAR Shared Memory in benchmarks.

4.3.1 Magma benchmark suite. Benchmarking in this experiment was carried out using the Magma ground-truth fuzzing benchmark suite [?]. The Magma benchmark suite is a ground truth fuzzing benchmark suite that uses a series of past security vulnerabilities in a suite of key open source projects to provide effective benchmarking of fuzzers against real world code and vulnerabilities. The current public version of Magma (v1.2) implements the following targets for fuzzing: libpng [?], libsndfile [?], libtiff [?], libxml2 [?], lua [?], openssl [?], php [?], poppler [?], sqlite3 [?]. For each of these targets, the Magma benchmark provides the required scripts to compile each project, as well as list of bugs to inject into the source code before compilation. Magma also provides a wide range of fuzzers which can be used, but the only upstream fuzzer benchmarked here is AFL++.

4.3.2 Experiment setup. All PeAR benchmarks instrumented binaries using persistent binaries wherever possible, which was for all binaries that provide an appropriate libfuzz API. As with many binary modification tools, PeAR was not able to successfully instrument all targets available for benchmarking under the Magma suite. A summary of the targets that were succesfully instrumented can be found in Table 3. Both the poppler and sqlite3 targets were unable to build as the instrumentation produced IR that was incompatible with GTIRB's re assembler. The php target did successfully build under PeAR, but initial testing revealed that the instrumented binary was unable to detect any bugs.

Similarly AFL++ was able to successfully compile an instrumented version of the libsndfile target, but was unable to succesfully locate any bugs, indicating that an error was occurring in the instrumentation.

All target programs were complied using `clang-9` or `clang++-9` as appropriate, with no additional performance flags set. In order to fuzz targets which present the libfuzz API (which most Magma targets do), the fuzzer needs to provide a wrapper that provides a main function, and handles passing the data from the fuzzer to the `LLVMFuzzerTestOneInput` function. As PeAR uses the AFL++ fuzzer, a modified version of Magma's AFL driver was used.

AFL cmplog AFL++ supports a mode called CmpLog instrumentation, which allows for logging of the results of all comparison operations for the mutator to use. However, initial testing seemed to indicate that on the benchmarks used in the Magma suite, CmpLog mode disadvantaged AFL++ in speed at finding bugs on short trials. Therefore, to ensure an accurate comparison between PeAR and AFL++, CmpLog mode was disabled for the full data runs.

Hardware Full hardware specifications for the machine used in benchmarks is listed in Table 4.

4.3.3 Testing structure. We conducted 30 runs of each combination of target and binary, with each run running for 24 hours, to provide enough data to effectively judge fuzzer performance [?].

Benchmarks for each fuzzers were carried out with 10 simultaneous runs for each of the 13 binaries that PeAR was able to successfully instrument, resulting in 130 concurrent trials running.

The initial set of 10 runs suggested that the PeAR Shared Memory benchmarks were not correctly locating bugs due to an error in the instrumentation or shared memory updating process. As such, to reduce the benchmark time required, PeAR Shared Memory was not included in the second and third set of runs.

The total testing time across all trials came to 10 days of runtime.

4.4 Instrumentation consistency

While any version of PeAR was only able in instrument five of the eight targets that Magma provides to benchmark, compared to AFL++ eight of nine, all versions of PeAR encountered the same instrumentation errors, indicating that between all versions of PeAR were instrumenting programs consistently with each other.

4.5 Bug finding efficacy

All results for AFL++ and PeAR presented are averaged across all 30 trials run for each target, with error bars representing the standard deviation in the data. For PeAR Shared Memory, averages were only across the 10 trials run.

Total bugs in all binaries for each target reached by each fuzzer is presented in Figure 3. Magma defines reached bugs as bugs where

Table 3: Magma benchmark successfully instrumented under various fuzzers. ✓✓ indicates persistent mode, ✓ indicates non-persistent mode. For failed instrumentation, ✗✗ indicates failure to produce a binary, ✗ indicates production of a binary that did not correctly fuzz.

Target	Driver	AFL++	PEAR	PEAR Shared Memory
libpng	libpng_read_fuzzer	✓✓	✓✓	✓✓
libsndfile	sndfile_fuzzer	✗	✓✓	✓✓
libtiff	tiff_read_rgba_fuzzer	✓✓	✓✓	✓✓
libtiff	tiffcp	✓	✓	✓
libxml2	libxml2_xml_read_memory_fuzzer	✓✓	✓✓	✓✓
libxml2	xmllint	✓	✓	✓
lua	lua	✓✓	✓✓	✓✓
openssl	asn1	✓✓	✓✓	✓✓
openssl	asn1parse	✓✓	✓✓	✓✓
openssl	bignum	✓✓	✓✓	✓✓
openssl	server	✓✓	✓✓	✓✓
openssl	client	✓✓	✓✓	✓✓
openssl	x509	✓✓	✓✓	✓✓
php	json	✓✓	✗	✗
php	exif	✓✓	✗	✗
php	unserialize	✓✓	✗	✗
php	parser	✓✓	✗	✗
poppler	pdf_fuzzer	✓✓	✗✗	✗✗
poppler	pdfimage	✓✓	✗✗	✗✗
poppler	pdftoppm	✓✓	✗✗	✗✗
sqlite3	sqlite3_fuzz	✓✓	✗✗	✗✗

Table 4: Benchmark server hardware specifications

Processor	
Processor	Intel® Xeon® Gold 6252
Core frequency	2.10GHz
Number of processors	4
Physical cores	24
Logical threads	48
Memory	
Capacity	191GB
Storage	
Capacity	891GB

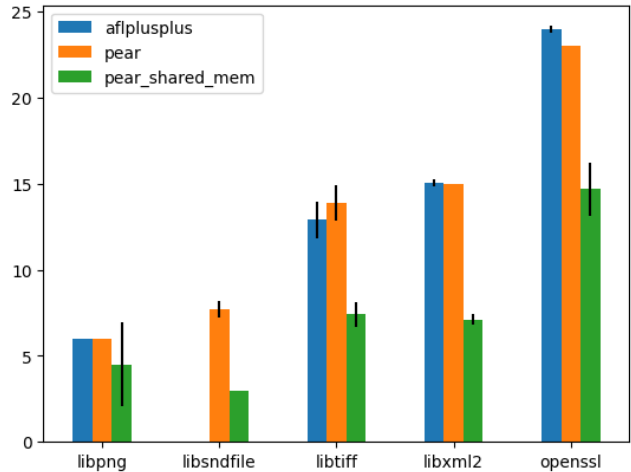


Figure 3: Total bugs found across all binaries for each benchmarked target, averaged across all runs

the context of the bug is reached, but the required program state to trigger faulty behaviour from the program has not necessarily been fulfilled. On the other hand, triggered bugs are ones where the all requirements for the bug to result in incorrect behaviour from the program. Total triggered bugs by each fuzzer for each target is presented in Figure 4.

Due to the highly stochastic nature of fuzzing campains, determining the importance of differences between each fuzzer can be challenging. As such, a Mann-Whitney U test was used to determine if the benchmarks indicated any statistically significant difference between the fuzzers benchmarked. Results from the Mann-Whitney U test are presented in Figure 5.

Examination of the raw results indicates that in general, we see similar numbers of bugs observed in the 24 hour runs between AFL++ and PEAR, with the PEAR Shared Memory implementation finding a significantly reduced number of bugs. This data is statistically supported by the Mann-Whitney U tests in Figure 5, where we see in most cases, there is no statistically significant different between the number of bugs found and triggered by AFL++ and

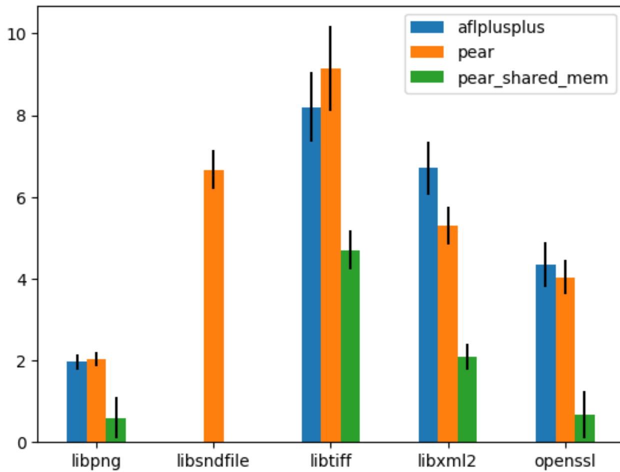


Figure 4: Total triggered found across all binaries for each benchmarked target, averaged across all runs

Table 5: Number of trials where each bug was found, where any fuzzer found the bug in fewer that 20 out of 30 trials.

Driver	Bug	AFL++	PEAR
`tiffcp`	TIF006	26	16
`tiffcp`	TIF009	24	17
`tiffcp`	TIF010	20	30
`tiffcp`	TIF002	1	0
`tiff_read_rgba_fuzzer`	TIF008	3	12
`tiff_read_rgba_fuzzer`	TIF002	7	29
`ssl server`	SSL020	30	0

PEAR. PEAR Shared Memory consistently produced p values very close to zero in all cases, indicating a strong statistical significance to the shared memory implementation performing worse that the non shared memory implementations.

4.5.1 Performance comparison between AFL++ and PEAR. In most cases benchmarked here, we did not find a statistically significant difference between the number of bugs found or triggered by AFL++ and PEAR, with a few exceptions.

On benchmarks of the libtiff target, PEAR performed a statistically significant margin better than AFL++ at number of bugs found and at triggering bugs. While both fuzzers found the same set of bug across all 30 trials, AFL++ was less consistent in finding some of the libtiff bugs. Bugs that were not consistently found by all trials carried out are presented in Table 5, with any bugs found in 20 or fewer trials of any of the fuzzers included, as these are the bugs where we see meaningful performance differences between the fuzzers. We can see from the table that there is fairly significant variation across several of the bugs testing in the libtiff benchmark, but PEAR is generally more consistent at finding libtiff bugs, particularly on more challenging bugs like TIF008 and TIF002.

When fuzzing openssl, PEAR found a very statistically significant margin less bugs than AFL++. However, this difference is entirely due to all PEAR implementation being unable to locate one specific bug in the openssl server, labelled as SSL020 in Magma, which was successfully found in all AFL++ trials, but found in no trials by any PeAR implementation. This suggests that, under the specific conditions in which the SSL020 bug is introduced, PeAR may be instrumenting inconsistently with AFL++, resulting in some bugs being missed.

While PEAR being unable to locate the correct context of the SSL020 bug did translate into also being unable to trigger it (see Table 6), the larger variation and smaller sample size of the triggered data resulted in SSL020 not causing a statistically significant deviation in the bug triggering results for the openssl benchmark.

On the libxml benchmark, PEAR successfully triggered a very statistically significantly lower number of bugs. The raw number of trials in which each bug was triggered is show in Table 6, again limited to trials where any of the fuzzers triggered the bug in less than 20 trials. This was again primarily due to PEAR being unable to trigger the XML001 bug in any trial, despite AFL++ being able to locate the bug in many trials. While PEAR also missed the XML012 bug, AFL++ only located the bug in exactly one trial, so it is likely that this is due to the highly variable nature of fuzzing runs, combined with a bug that is challenging to find in the 24 hour campaign lifespan. PEAR being consistently unable to trigger the XML001 bug again supports the finding from the openssl trial, which is that under some specific conditions, PeAR is not instrumenting consistently with AFL++.

XML001 is assigned CVE-2017-9047, which uses an incorrect value in a bounds check to allow for a buffer overflow to cause a program crash.

4.5.2 Reduced performance of PEAR Shared Memory. The initial benchmark runs indicated a significant performance decrease between PEAR caused by enabling the shared memory mode in effectiveness at bug finding. The expectation was that shared memory fuzzing should reduce fuzzing overheads, allowing increased execution speed and therefore better bug finding performance. The significant bug reduction in bug finding performance of the shared memory mode is likely indicative of an error in either the instrumentation or the custom hook used to setup the data, but a detailed investigation of the shared memory performance was not carried out.

4.6 Summary of findings

We summarize here the results of our experiments along the line of the four metrics listed in the beginning of this section.

Code coverage PEAR achieves comparable code coverage when compared against state-of-the-art binary-only fuzzers for most benchmarks.

Speed In its base configuration (without persistent mode), PEAR achieves the highest performance compared to other binary-only fuzzers. With the persistent mode enabled, PEAR significantly outperformed all other binary-only fuzzers, on average performing twice as fast as AFL++-QEMU, three times faster than `afl-dyninst` and 12 to 31 times faster than E9AFL.

Robustness The only two SBI fuzzing frameworks that support FUZZBENCH are `afl-dyninst` and E9AFL, with the latter failed to instrument most of the benchmarks. PEAR has a slight advantage

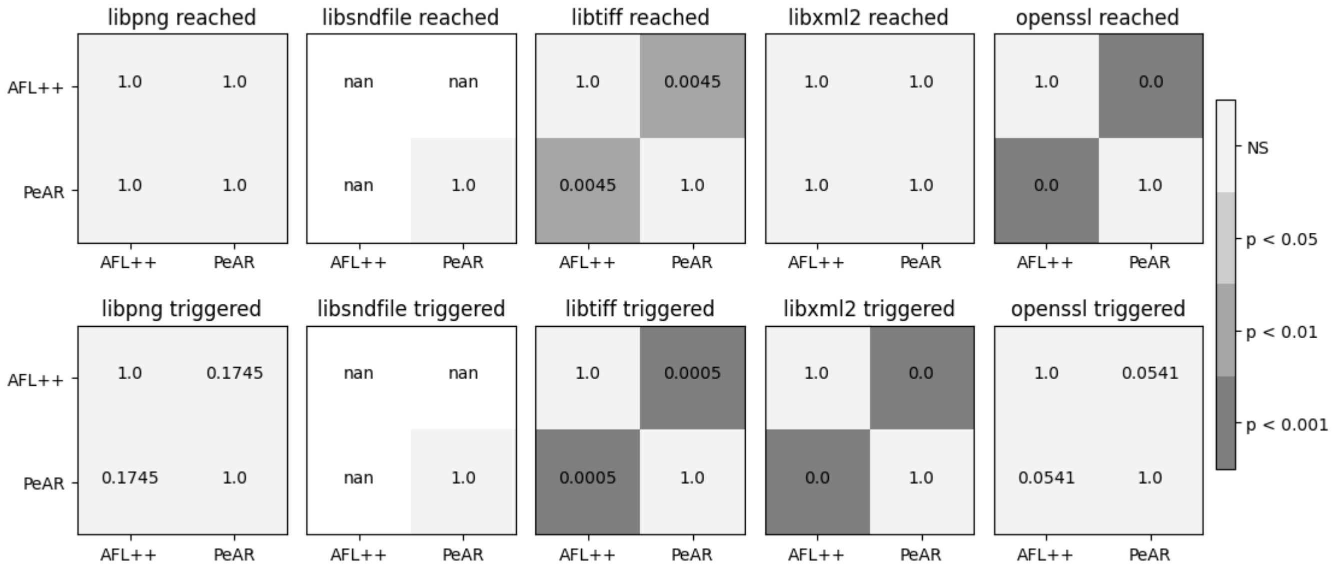


Figure 5: Statistical significance of pair wise comparisons between number of bugs found by each fuzzer. Raw data in Figure 3 and Figure 4. In all cases, PeAR Shared Memory had a p value of 0 compared to all other tests, so is excluded for clarity.

Table 6: Number of trials where each bug was triggered, where any fuzzer triggered the bug in fewer that 20 out of 30 trials.

Driver	Bug	AFL++	PeAR
`tiffcp`	TIF006	26	16
`tiffcp`	TIF009	24	17
`tiff_read_rgba_fuzzer`	TIF008	3	16
`tiff_read_rgba_fuzzer`	TIF002	4	18
`xmllint`	XML001	12	0
`libxml2_xml_read_memory_fuzzer`	XML001	30	0
`libxml2_xml_read_memory_fuzzer`	XML012	1	0
`ssl server`	SSL020	9	0

over `afl-dyninst`, and is approaching the robustness of AFL++-QEMU (which is the gold standard in terms of robustness), despite using only SBI-based instrumentation.

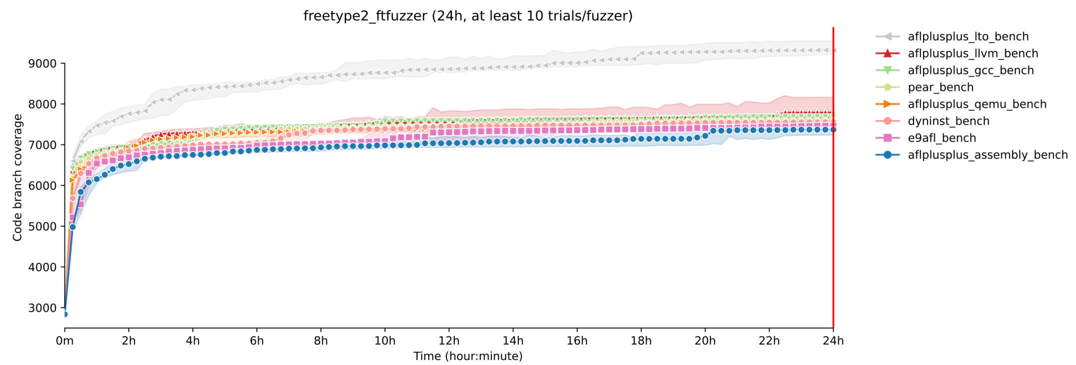
Bug finding Across 30 repeats of 24 hour trials, for most benchmark targets provided by the Magma suite, we did not find a statistically significant difference between the number of bugs found and triggered between AFL++ and PeAR, indicating in most cases PeAR is instrumenting binaries in a consistent manner with the instrumentation produced by the AFL++ compiler. When benchmarking libtiff, PeAR showed a statistically significant improvement over AFL++ at finding and triggering bugs. In contrast, in some trials, namely libxml and openssl, we found that all PeAR implementations were consistently unable to find or trigger a single bug, indicating that in some contexts, PeAR instrument binaries slightly inconsistently with AFL++ instrumentation.

5 CONCLUSION AND FUTURE WORK

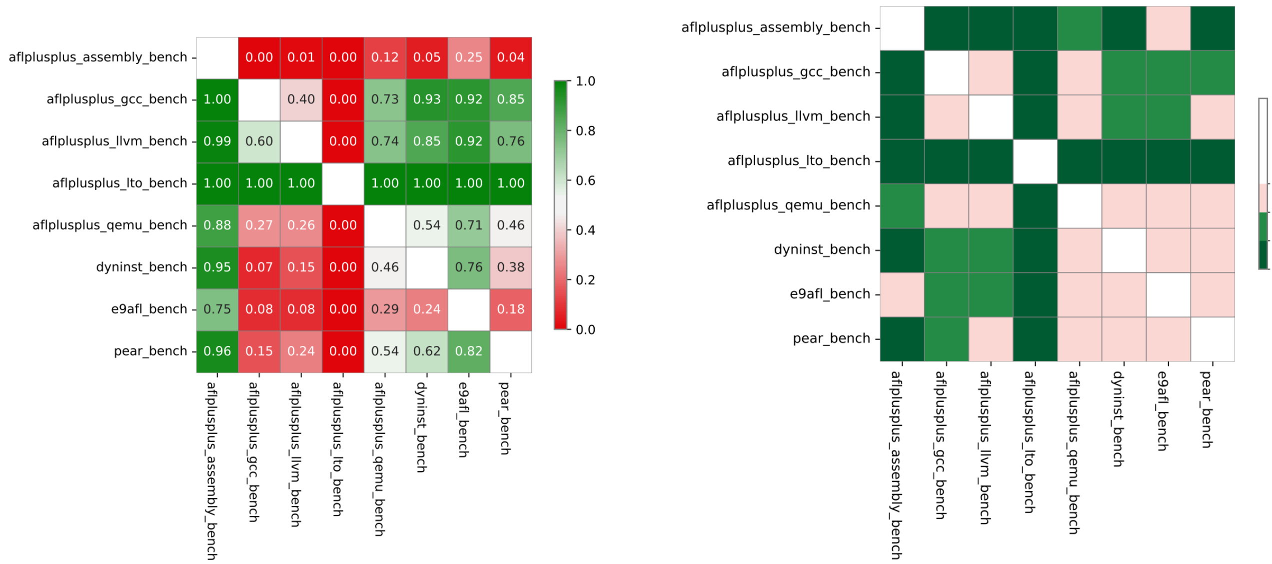
In light of recent advances in SBI frameworks such as GTIRB, we re-examine the viability of using SBI to implement and effective and efficient binary-only fuzzing framework. We show that despite the many challenges in implementing SBI fuzzing frameworks [**?** **?**], it is indeed possible to implement an SBI fuzzing framework that is robust and with comparable code coverage and bug finding efficacy and performance to a compiler instrumentation. Our implementation, PeAR, significantly outperforms the state-of-the-art binary-only fuzzers, achieving between 2–31× speedup across the benchmarks we evaluated, and without any statistical significant degradation in code coverage and bug finding efficacy. The keys to this performance is the ability of PeAR to incorporate advanced optimisation techniques such as deferred initialisation, persistent mode and shared memory fuzzing that are not supported by any other SBI fuzzing frameworks.

For future work, we plan to implement support for a wider range of architetures and platforms. An experimental version of PeAR for Windows PE binary is currently in development. We also plan to add other advanced instrumentations, in particular, adding support to instrument sanitizers into binaries, like what is done in RetroWrite.

A APPENDIX: EXPERIMENT RESULTS

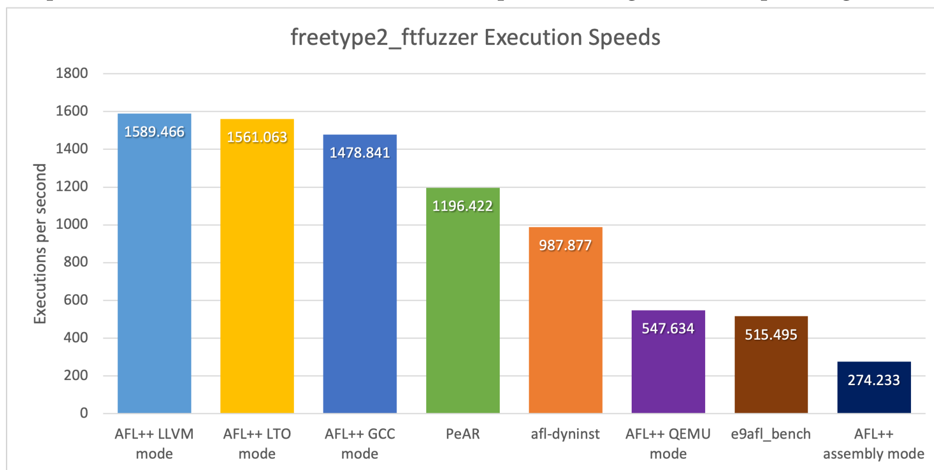


(a) Mean code coverage growth over time



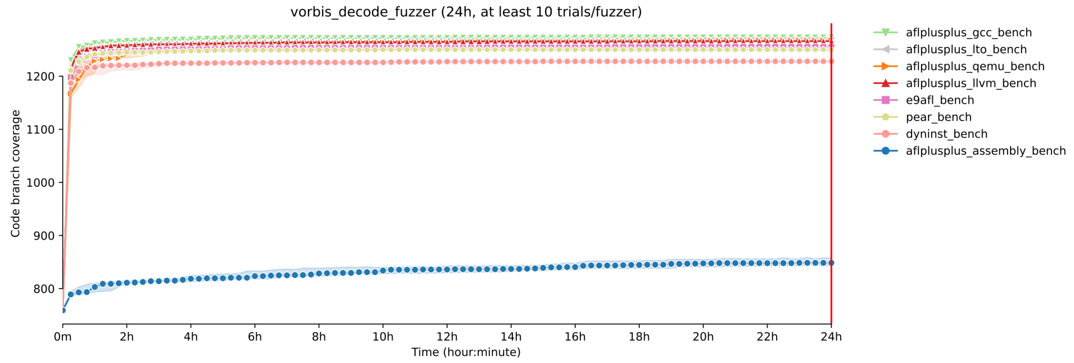
(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

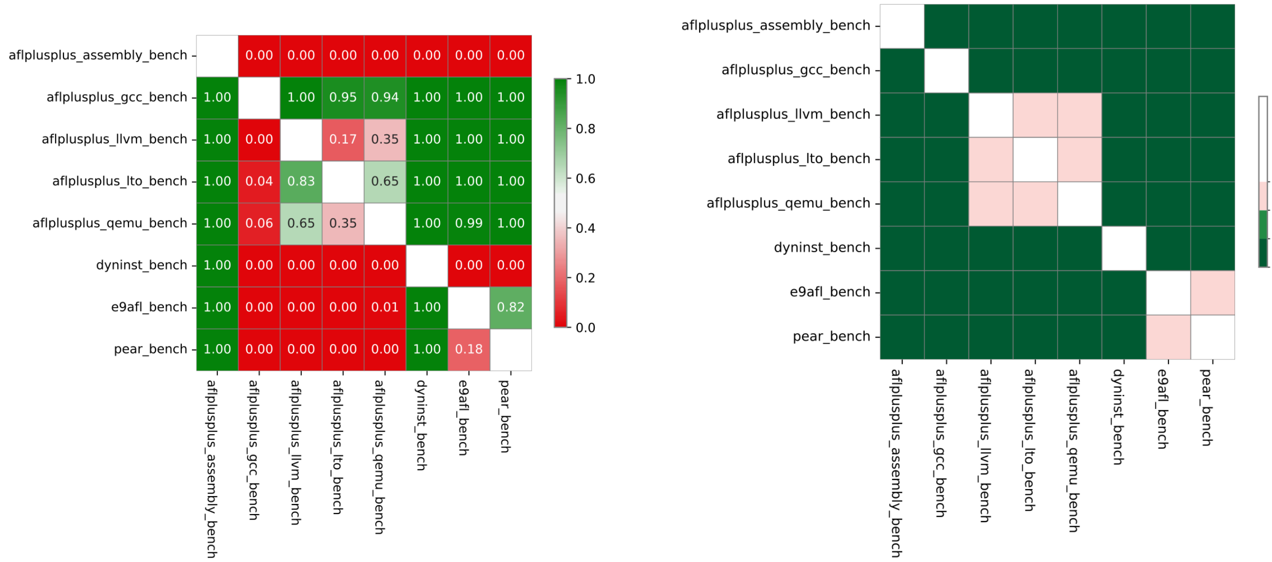


(d) Mean executions per second

Figure 6: Results for freetype2_ftfuzzer benchmark

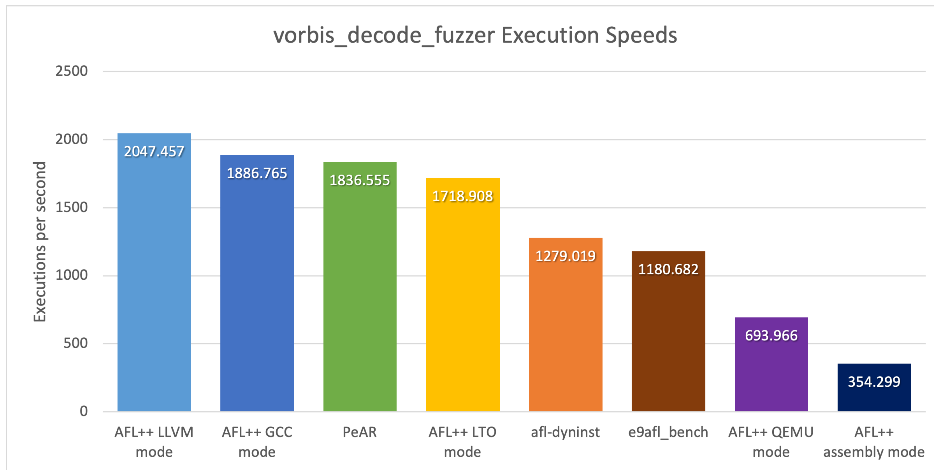


(a) Mean code coverage growth over time



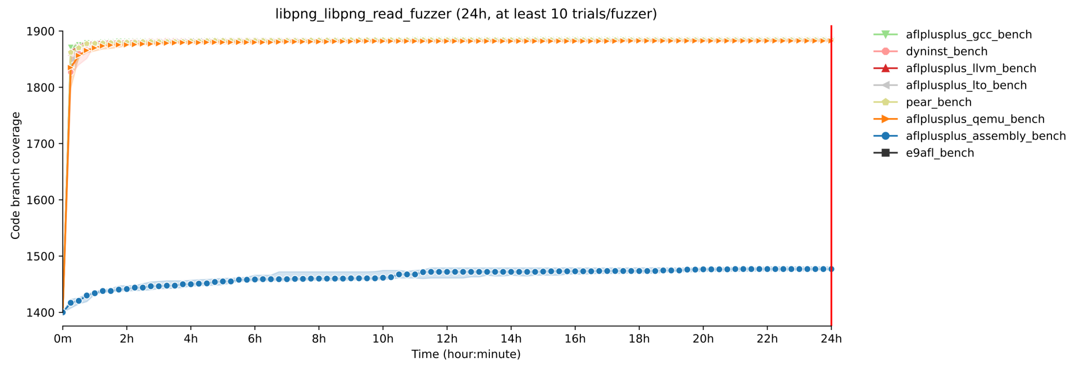
(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

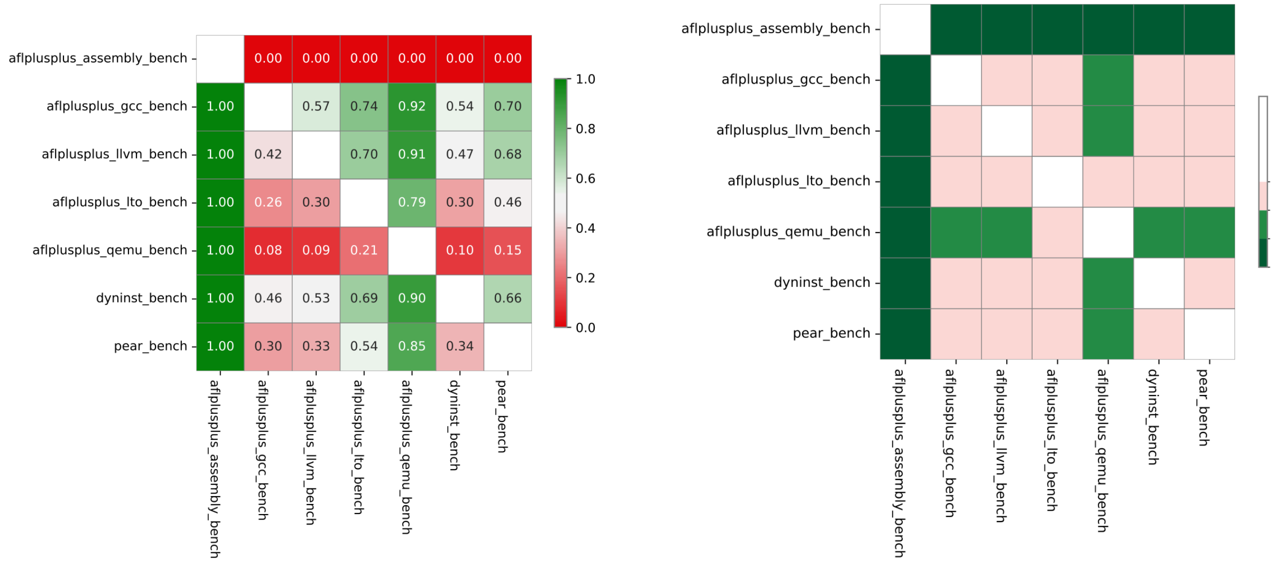


(d) Mean executions per second

Figure 7: Results for vorbis_decode_fuzzer benchmark

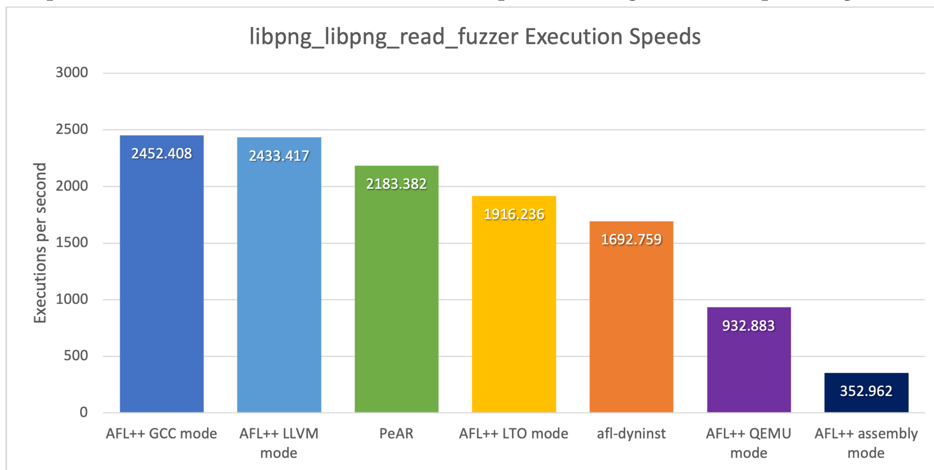


(a) Mean code coverage growth over time



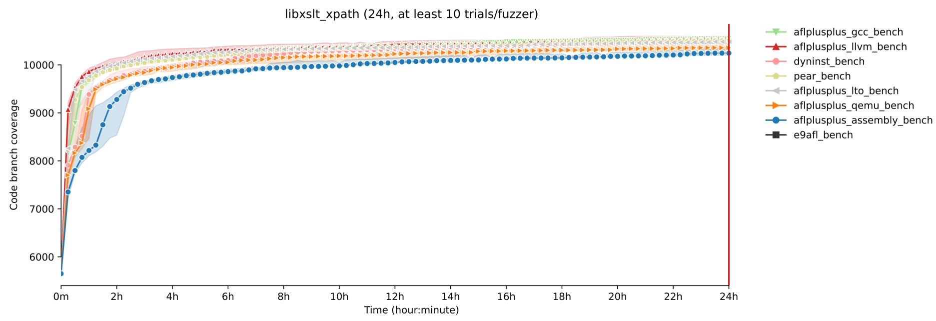
(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

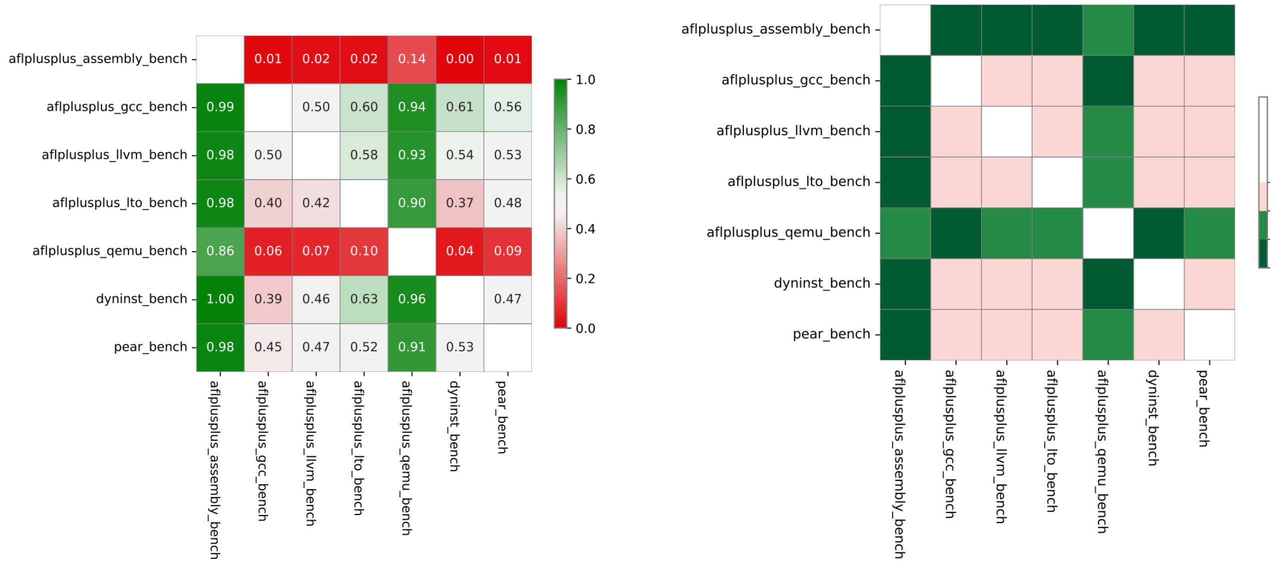


(d) Mean executions per second

Figure 8: Results for libpng_libpng_read_fuzzer benchmark



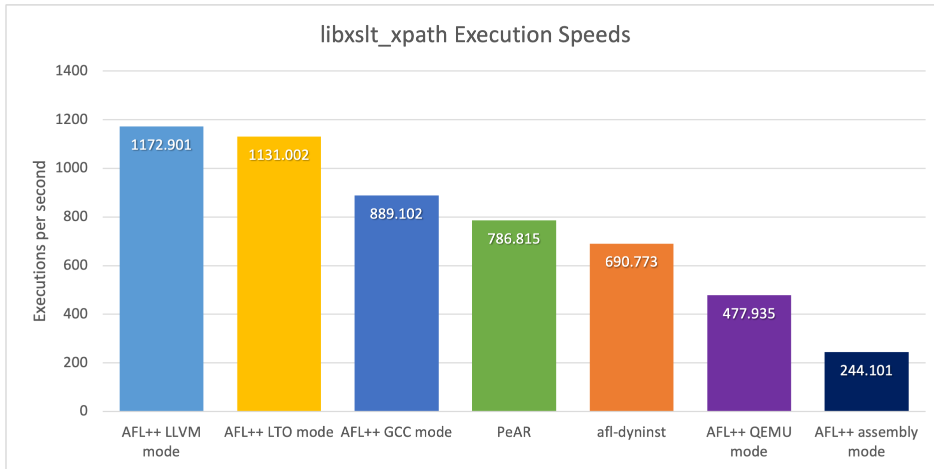
(a) Mean code coverage growth over time



	aflplusplus_assembly_bench	aflplusplus_gcc_bench	aflplusplus_llvm_bench	aflplusplus_lto_bench	aflplusplus_qemu_bench	dyninst_bench	pear_bench
aflplusplus_assembly_bench		0.01	0.02	0.02	0.14	0.00	0.01
aflplusplus_gcc_bench	0.99		0.50	0.60	0.94	0.61	0.56
aflplusplus_llvm_bench	0.98	0.50		0.58	0.93	0.54	0.53
aflplusplus_lto_bench	0.98	0.40	0.42		0.90	0.37	0.48
aflplusplus_qemu_bench	0.86	0.06	0.07	0.10		0.04	0.09
dyninst_bench	1.00	0.39	0.46	0.63	0.96		0.47
pear_bench	0.98	0.45	0.47	0.52	0.91	0.53	

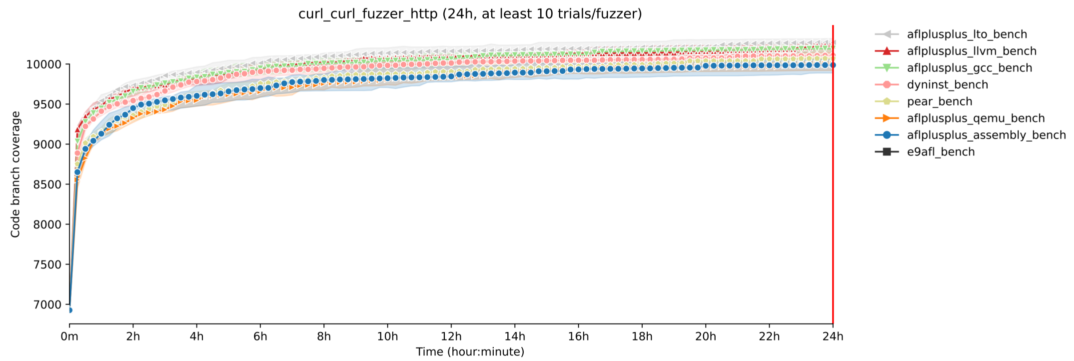
(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

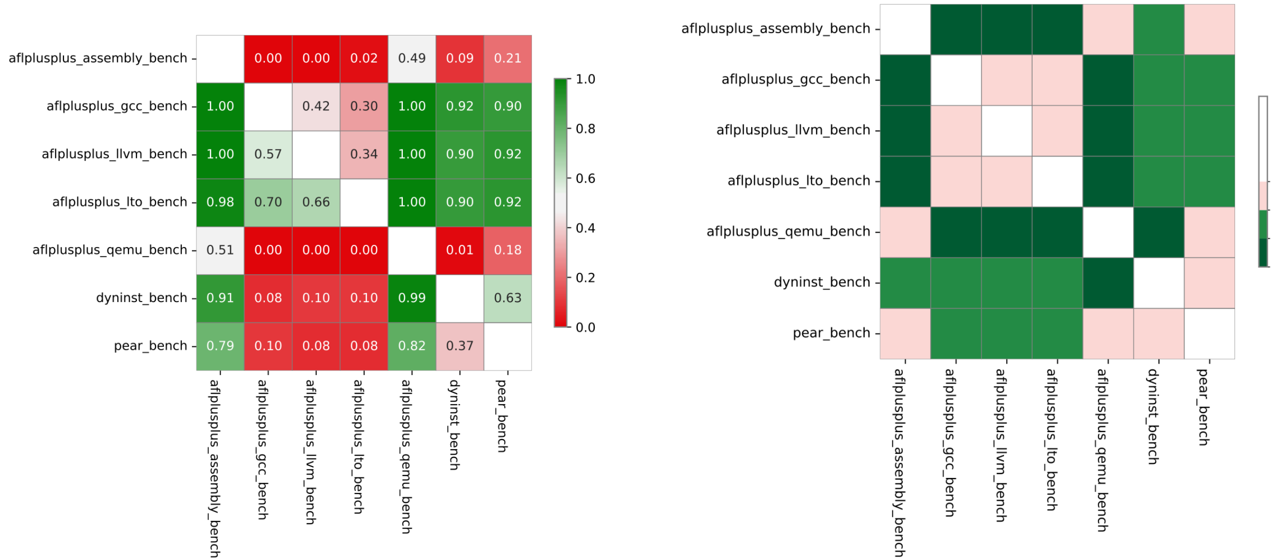


(d) Mean executions per second

Figure 9: Results for libxslt_xpath benchmark

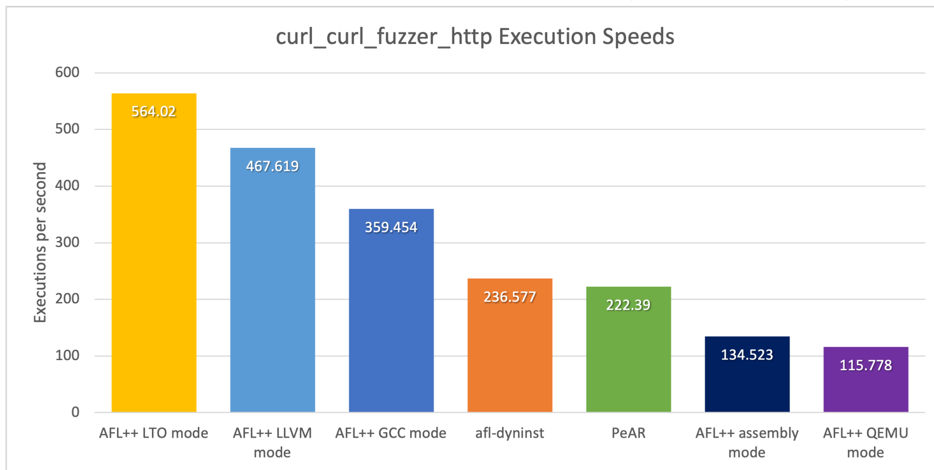


(a) Mean code coverage growth over time



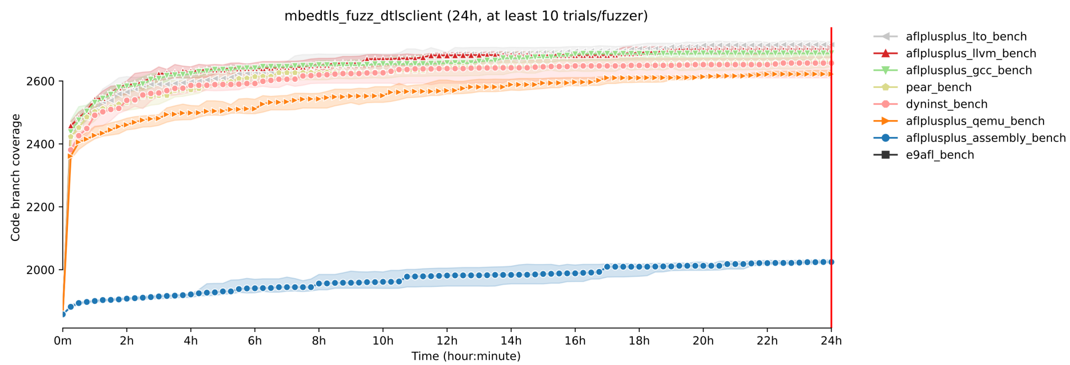
(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

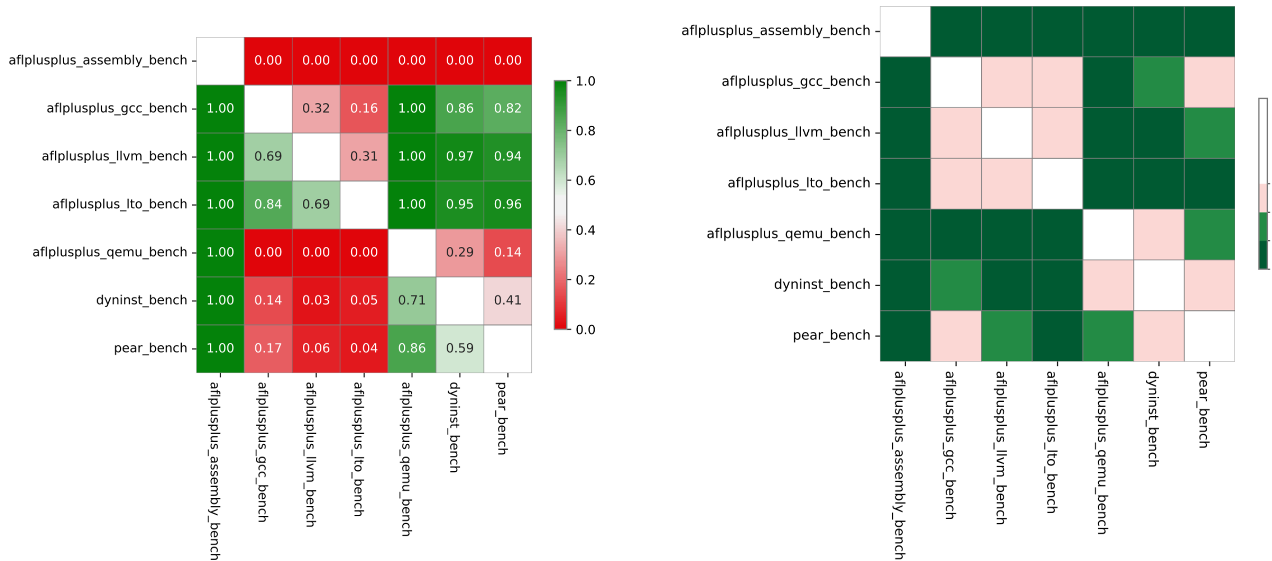


(d) Mean executions per second

Figure 10: Results for curl_curl_fuzzer_http benchmark

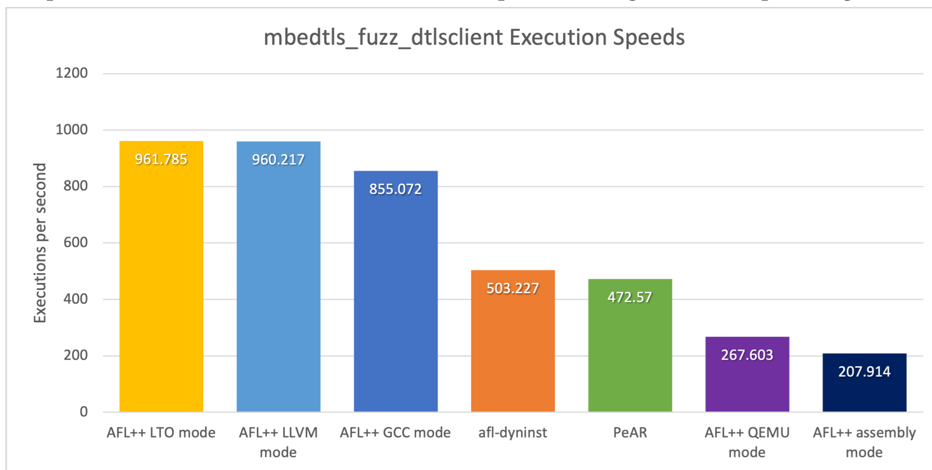


(a) Mean code coverage growth over time



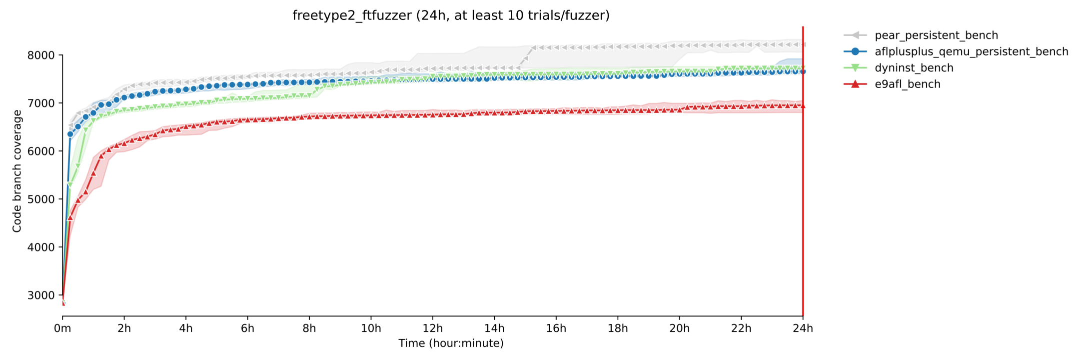
(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

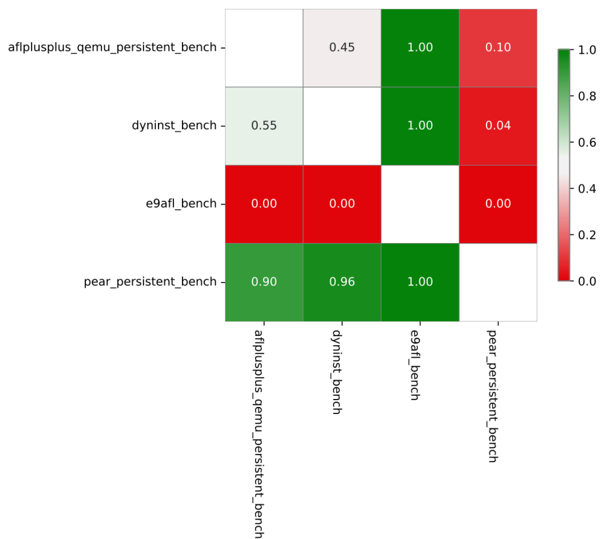


(d) Mean executions per second

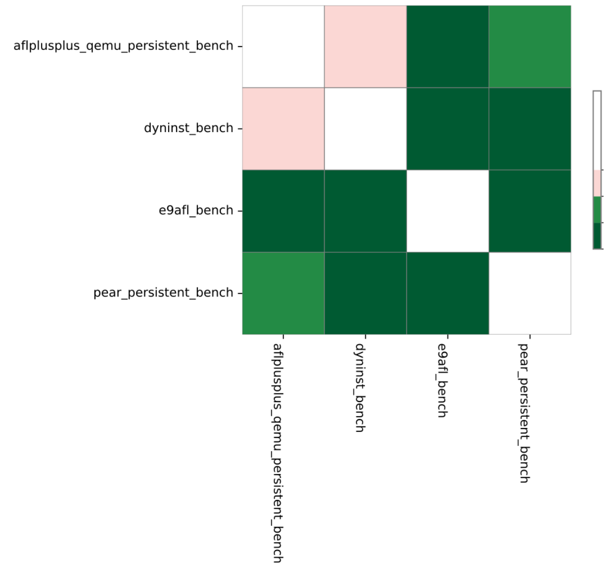
Figure 11: Results for mbedtls_fuzz_dtlsclient benchmark



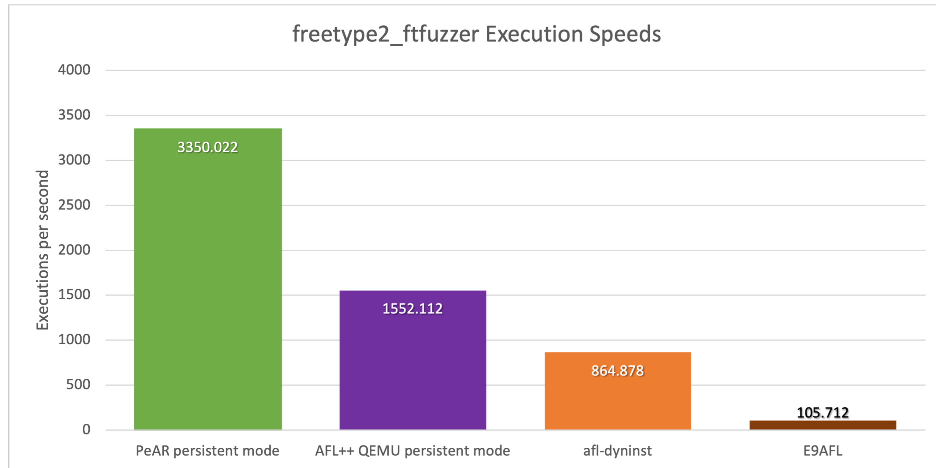
(a) Mean code coverage growth over time



(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

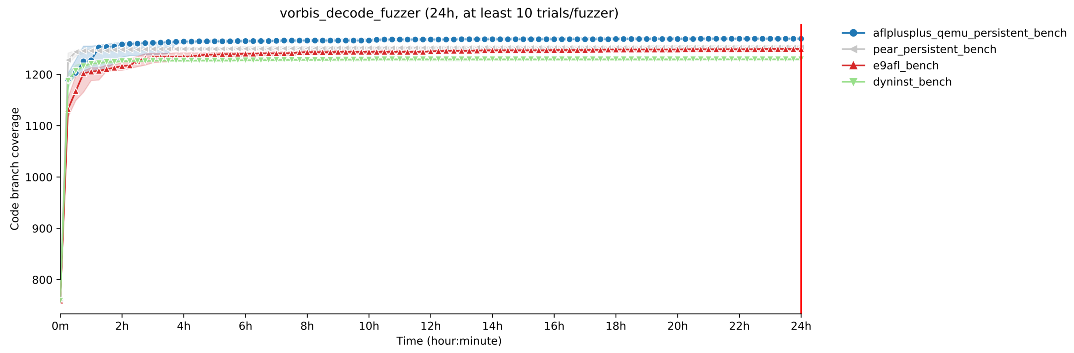


(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.

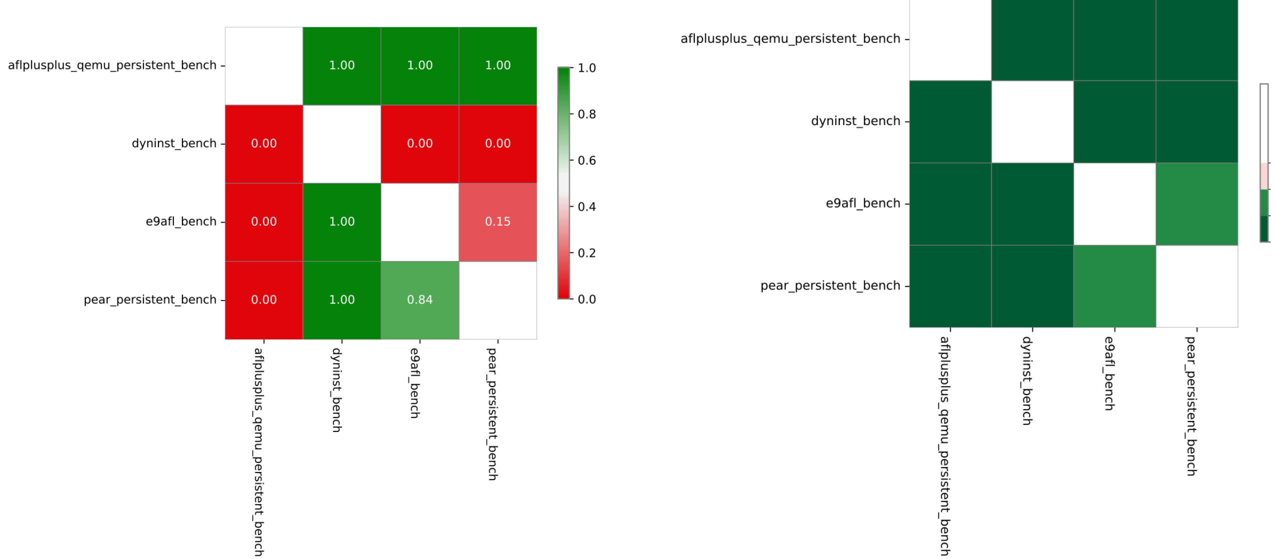


(d) Mean executions per second

Figure 12: Results for freetype2_ftfuzzer benchmark

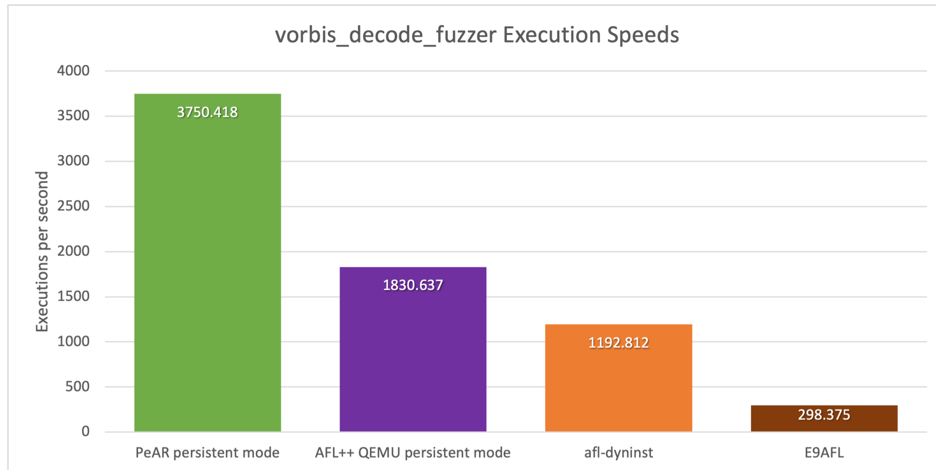


(a) Mean code coverage growth over time



(b) Varga-Delaney A12 measure. Green cells indicate the probability the fuzzer in the row will outperform the fuzzer in the column.

(c) p values of pairwise Mann-Whitney U tests. Green cells indicate the explored coverage of a fuzzer pair is significantly different.



(d) Mean executions per second

Figure 13: Results for vorbis_decode_fuzzer benchmark