RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing

Mingzhe Wang, Jie Liang, Chijin Zhou, and Yu Jiang, *Tsinghua University;*
Rui Wang, *Capital Normal University;* Chengnian Sun, *Waterloo University;*
Jiaguang Sun, *Tsinghua University*

https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing

Mingzhe Wang
Tsinghua University

Jie Liang
Tsinghua University

Chijin Zhou
Tsinghua University

Yu Jiang
Tsinghua University

Rui Wang
Capital Normal University

Chengnian Sun
Waterloo University

Jiaguang Sun
Tsinghua University

Abstract

Coverage-guided fuzzers use program coverage measurements to explore different program paths efficiently. The coverage pipeline consists of runtime collection and post-execution processing procedures. First, the target program executes instrumentation code to collect coverage information. Then the fuzzer performs an expensive analysis on the collected data, yet most program executions lead to no increases in coverage. Inefficient implementations of these steps significantly reduce the fuzzer's overall throughput.

In this paper, we propose RIFF, a highly efficient program coverage measurement mechanism to reduce fuzzing overhead. For the target program, RIFF moves computations originally done at runtime to instrumentation-time through static program analysis, thus reducing instrumentation code to a bare minimum. For the fuzzer, RIFF processes coverage with different levels of granularity and utilizes vector instructions to improve throughput.

We implement RIFF in state-of-the-art fuzzers such as AFL and MOpt and evaluate its performance on real-world programs in Google's FuzzBench and fuzzer-test-suite. The results show that RIFF improves coverage measurement efficiency of fuzzers by 23× and 6× during runtime collection and post-execution processing, respectively. As a result, the fuzzers complete 147% more executions, and use only 6.53 hours to reach the 24-hour coverage of baseline fuzzers on average.

1 Introduction

Fuzzing is an automated testing technique that attempts to detect bugs and vulnerabilities in programs [1, 3, 9, 13, 14, 24, 27, 31, 35, 36]. Coverage-guided fuzzing improves bug-detection ability of fuzzers by leveraging program coverage measurements to guide fuzzing towards exploring new program states [4, 8, 20, 29, 40]. These fuzzers perform the following steps: ① the fuzzer selects an input from the corpus and performs mutation operations to generate new inputs; ② the fuzzer executes the target program with mutated inputs and collects coverage statistics of these runs; ③ the fuzzer saves the input to the corpus if it can trigger bugs or find new program states. With proper coverage guidance, fuzzers can improve their efficiency by prioritizing mutation on *interesting* inputs in the corpus and discarding inputs that do not reach any new program states.

Generally speaking, the coverage pipeline of fuzzers consists of two stages: runtime coverage information collection and post-execution processing: first, the target program is instrumented with coverage collection code, which updates an array of counters to record the runtime execution trace; after the completion of an execution, the fuzzer processes the values in the array to check whether each execution reaches any new program states.

An instrumented program executes many more instructions compared to a non-instrumented binary. Since fuzzers continuously execute random inputs, a slight slow-down can significantly impact overall fuzzing performance. We analyze the source of overhead using many microarchitectural performance counters.

For the target program, fuzzers insert instrumentation code for coverage collection at each basic block. The collection code saves the current register context, loads the base address for the counter region, computes the counter index, updates the corresponding counter value, restores the context, and transfers control back to the program logic (see Figure 2). The code is executed frequently, and can contain dozens of instructions encoded in around a hundred bytes. Furthermore, modern processors use a multi-tier cache subsystem to reduce memory latency. Because the collection code updates the counter array, it adds many loads or stores to the instruction stream. These memory accesses stress the memory subsystem by competing with the program logic for instruction cache. The extra memory latency reduces the overall execution speed of programs.

For the fuzzer itself, the instructions which process coverage do not uncover new states in most cases. While new program states are extremely rare, fuzzers need to perform the following operations: convert the raw coverage information into features, then check the database of known features, and update the database to add the newly discovered features [42]. This algorithm is implemented using memory write-, integer comparison- and conditional branching instructions. The complex nature of the code prevents the compiler from optimizing it. Consequently, the instructions emitted by the compiler cannot fully utilize instruction-level parallelism supported by the processor's execution engine.

In this paper, we propose RIFF to reduce instruction footprints of coverage pipelines and improve fuzzing throughput. RIFF utilizes compiler analyses and leverages low-level features directly exposed by the processor's instruction set architecture. Specifically, ① RIFF reduces the amount of instructions executed for *runtime coverage collection* in the target program. First, RIFF removes edge index computations at runtime by pre-computing the edge transfers at instrumentation-time. Next, RIFF eliminates the instructions for loading the base of the counter region by assigning the region a link-time determined static address. Thus, RIFF can use only one instruction encoded in 6 bytes per instrumentation site. ② RIFF removes unnecessary instructions when *processing coverage* in fuzzers by dividing processing granularity into three stages, where the first stage handles simple scenarios fast, while the last stage is suited for more sophisticated scenarios. For the most common case, RIFF scans the coverage region and skips zero-valued chunks using vector instructions, analyzing 16, 32, or 64 counters per iteration on modern processors.

To demonstrate the effectiveness of our approach, we implement RIFF by augmenting state-of-the-art fuzzers such as AFL [40] and MOpt [29] and evaluate its performance on real-world programs from Google's fuzzer-test-suite [21] and FuzzBench [30]. On the coverage collection side, RIFF reduces the average runtime overhead of instrumentation from 207% to 8%. On the post-execution processing side, RIFF reduces coverage processing time from 217 seconds to 42 seconds with AVX2 instructions [10] and 31 seconds with AVX512 instructions [34]. As a result, the enhanced fuzzers can complete 147% more executions during the 24-hour experiments, covering 13.13% more paths and 5.60% more edges. Alternatively, the improved fuzzers need only 6.53 hours to reach the 24-hour coverage of baseline fuzzers.

In summary, this paper makes the following contributions:

- We observe that the collection and processing of program coverage measurements significantly affect the speed of fuzzing. We break down the cost of instrumentation and analysis code.
- We eliminate much of the runtime cost by using precomputing information statically, and we accelerate post-execution processing using vectorization.
- We adapt RIFF to popular fuzzers and achieve significant speedup on real-world programs. The coverage analysis algorithm of our work has been integrated into production-level fuzzer AFL++ [23].

2 Background

2.1 Stages of a Coverage Pipeline

To guide fuzzing using coverage, fuzzers use a multi-stage pipeline. Figure 1 takes AFL as an example to demonstrate how fuzzers handle coverage:

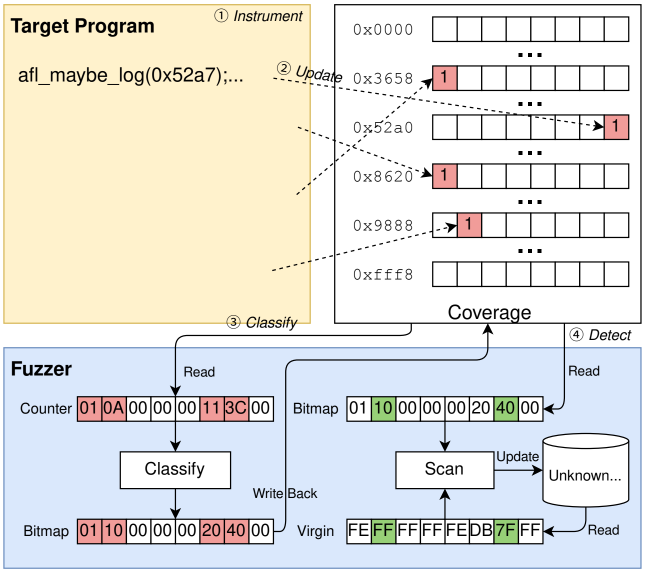


Figure 1: The coverage pipeline of the standard fuzzing tool AFL. After collecting the coverage from the target program (arrows labeled "instrument" and "update"), the fuzzer determines whether the input triggers new program behavior ("classify" and "detect").

① *Instrument*. At compile time, `afl-clang` allocates an array of 65,536 counters to store coverage as 8-bit counters. For each basic block of the target program, `afl-as` generates a random number `ID` as its identifier, then inserts a call to `afl_maybe_log(ID)` at the beginning.

After instrumentation, the fuzzer generates random inputs and executes the program on each input. For each input the fuzzer detects whether the input triggers new program states by using a database, as follows:

② *Update*. At run time, `afl_maybe_log` updates the coverage counters to collect edge coverage. The logging function hashes the identifier of the previously executed and the current block, then uses the hash as an index into the counter array to increment the pointed counter by one.

③ *Classify*. After the target program completes execution, AFL reads the coverage counters to classify them into a bitmap of features. Each 8-bit counter with nonzero value is mapped to 8 possible features. The features are represented as a bitmap, where each feature corresponds to one of the 8 bits inside the 8-bit counter. The classified result is written back to the coverage region.

④ *Detect.* With the edge transfer counts classified as a bitmap, AFL scans the database of unknown program states to detect new program behaviors: if a previously-unknown edge transfer is triggered, then the input will be labeled as "new coverage"; if a known edge transfer has different features, then it will be marked as a "new path"; otherwise, the current input is discarded. After the scan, AFL removes the newly discovered features by updating the database.

2.2 Variants of Coverage Pipeline

While the implementation varies for different fuzzers, the design mostly follows the classic coverage pipeline first introduced by AFL. Table 1 presents the instrumentation mechanism for popular fuzzers. Despite different tool chains and compiler infrastructures, all the collection methods insert code or callbacks to collect coverage. For example, although SanitizerCoverage contains a set of instrumentation options and is implemented in both Clang [7] and GCC [6], it uses callbacks and array updates to report coverage. Note that FuzzBench implements its own instrumentation for AFL [15], we only list it for completeness.

Table 1: Methods for Collecting Coverage

Method	Target	Infrastructure
afl-{clang,gcc}	Assembler	N/A
afl-clang-fast	Clang	LLVM Pass
afl-fuzzbench	Clang	SanitizerCoverage
libFuzzer	Clang	SanitizerCoverage
honggfuzz	Clang/GCC	SanitizerCoverage
Angora	Clang	LLVM Pass

Table 2 summarizes post-processing methods of coverage counters at fuzzers' side. *honggfuzz* is a special case because it processes coverage in real-time. Other fuzzers first classify the counter array to a bitmap of features, then scan the bitmap to detect the presence of new features.

Table 2: Methods for Processing Coverage

Method	Classify	Scan
AFL	Batch	Bit twiddling
libFuzzer	Per Counter	Statistics update
honggfuzz	N/A	N/A
Angora	Distill	Queued

For example, AFL implements a two-pass design. In the first pass, it performs bitmap conversion in batch; in the second pass, it applies bit twiddling hacks for acceleration. *libFuzzer* employs a one-pass design: for each non-zero byte, libFuzzer converts it to a feature index, then updates the local and global statistics with complex operations such as binary search. *Angora* takes the queued approach: in the first pass, it distills a small collection of counter index and feature mask out of the original array; in the second pass, it scans the collection to detect new coverage and pushes the modifications to the write-back queue; in the third pass, it locks the global database and applies the queued modifications.

3 Measuring Coverage Pipeline Overheads

To measure the overhead of the coverage pipeline, we select the classic fuzzer AFL as an example: as the forerunner of coverage-guided fuzzing, most coverage-guided fuzzers partially or completely inherit its design. As for the target program and workload, we use libxml2 from FuzzBench.

3.1 Cost of Instrumentation

To evaluate the overhead of coverage collection, we select all instrumentation methods provided by AFL, which cover all compiler infrastructures listed in Table 1. To have a fair comparison, we select afl-clang, afl-fuzzbench, and afl-clang-fast, because they have the same coverage update method and base compiler. We further decrease the optimization level of afl-fuzzbench to `-O2` to match with the other instrumentation methods.

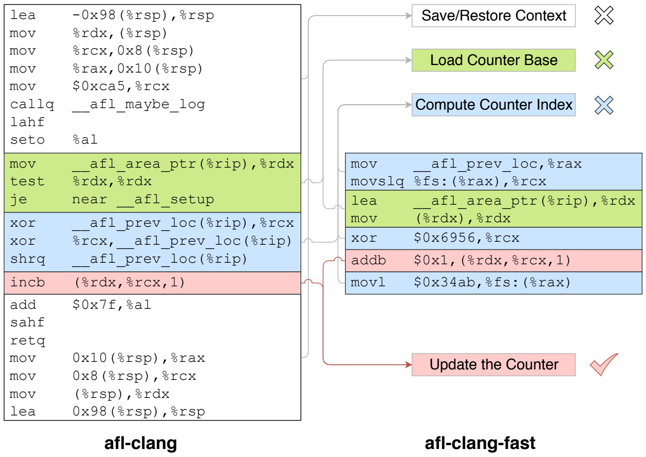
We collect performance metrics by running the target program using perf tools. To remove the one-time cost of program startup, we do a warm-up run of the program with 1 input, then use 11 more inputs separately, then calculate the average of per-execution cost. The Intel Intrinsics Guide [12] is used as the XML input.

Table 3 lists the overhead of each collection method by normalizing each metric to the non-instrumented baseline program. Looking at the "duration" column, we can see that the instrumentation significantly slows down program execution. For example, as soon as the fastest method, afl-clang-fast, finishes executing its first input, the non-instrumented program has executed more than half of the second input.

Table 3: Overheads of Instrumented Programs

Method	Duration	Instructions	L1-I	L1-D	μops
afl-clang	3.50x	4.26x	102.36x	5.16x	4.72x
afl-fuzzbench	2.45x	2.83x	19.88x	2.53x	2.14x
afl-clang-fast	1.69x	1.79x	33.58x	2.88x	2.11x

The "instruction count" column explains the slowdown. Figure 2 lists the instructions of afl-clang (the slowest method), and afl-clang-fast (the fastest method). Take afl-clang for example, for each basic block, it inserts 10 instructions encoded in 56 bytes. These instructions save the context, invoke `__afl_maybe_log`, and restore the context. In `__afl_maybe_log`, instructions totaling 44 bytes are executed, which update the last code location and increase the counter. They even contain a conditional jump which checks whether the coverage counters are initialized. The same problem is still applicable to the fastest method afl-clang-fast: of all the 7 instructions it executes, only one instruction is used to actually update the counter.



```
lea     -0x98(%rsp),%rsp
mov     %rdx,(%rsp)
mov     %rcx,0x8(%rsp)
mov     %rax,0x10(%rsp)
mov     $0xca5,%rcx
callq   __afl_maybe_log
lahf
seto    %al
mov     __afl_area_ptr(%rip),%rdx
test    %rdx,%rdx
je      near __afl_setup
xor     __afl_prev_loc(%rip),%rcx
xor     %rcx,__afl_prev_loc(%rip)
shrq    __afl_prev_loc(%rip)
incb    (%rdx,%rcx,1)
add     $0x7f,%al
sahf
retq
mov     0x10(%rsp),%rax
mov     0x8(%rsp),%rcx
mov     (%rsp),%rdx
lea     0x98(%rsp),%rsp
```

```
mov     __afl_prev_loc,%rax
movslq  %fs:(%rax),%rcx
lea     __afl_area_ptr(%rip),%rdx
mov     (%rdx),%rdx
xor     $0x6956,%rcx
addb    $0x1,(%rdx,%rcx,1)
movl    $0x34ab,%fs:(%rax)
```

Figure 2: Instructions inserted by afl-clang (22 instructions, 100 bytes) and afl-clang-fast (7 instructions, 39 bytes). Note that only the instruction marked in red updates the counter.

The instrumentation code has a significant processor cost. First, it starves the processor's front end which translates instructions to micro-ops. For each basic block, afl-clang requires executing extra instructions totaling 100 bytes, i.e. 1/256 of all the available L1 instruction cache. As a result, afl-clang experiences 101.36x more L1 instruction cache misses, and the CPU executes 3.72x more micro-ops for afl-clang produced programs.

3.2 Unnecessary Instructions in Fuzzer

Figure 3 presents the cost breakdown of afl-fuzz by sampling its CPU usage. To reduce noise introduced by fuzzing, we sample the user space CPU cycles for 5 seconds after afl-fuzz has discovered 2,000 paths of libxml2.

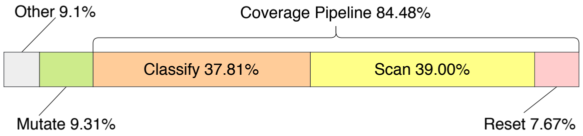


Figure 3: Breakdown of execution costs for afl-fuzz: AFL first mutates the input; after the execution completes, it classifies the coverage to bitmap, scans the bitmap for new coverage, and resets the memory for the next run.

From the figure we can see that afl-fuzz spends the majority of its time on the coverage pipeline. To detect new program states, AFL spends 84.48% of its valuable CPU time on the coverage pipeline. The overwhelmingly high percent of CPU usage implies significant problems behind the overall system design, which inevitably leads to redundancy in executed instructions.

Table 4 shows that most executions do not improve coverage. We call a counter "useless" if its value is zero, since a zero-valued counter never maps to a feature. We call a program execution "useless" if its bitmap does not contain any new feature with respect to previous executions. After AFL terminates, we collected the coverage of the first discovered 2,000 paths, and calculated useless counters (see the first row). We also compute useless executions during a 5-second time interval (see the second row). During the period, AFL had executed 67,696 inputs, where each execution required processing 64 KiB of coverage. Although it had processed over 4,231 MiB of coverage, it only discovered 2 new paths, and none of the paths covered new counters.

Table 4: Number of Processed Counters and Executions

	Total	Useless	Proportion
Counter	65,536	64,664.37	98.67%
Execution	67,696	67,694	99.997%

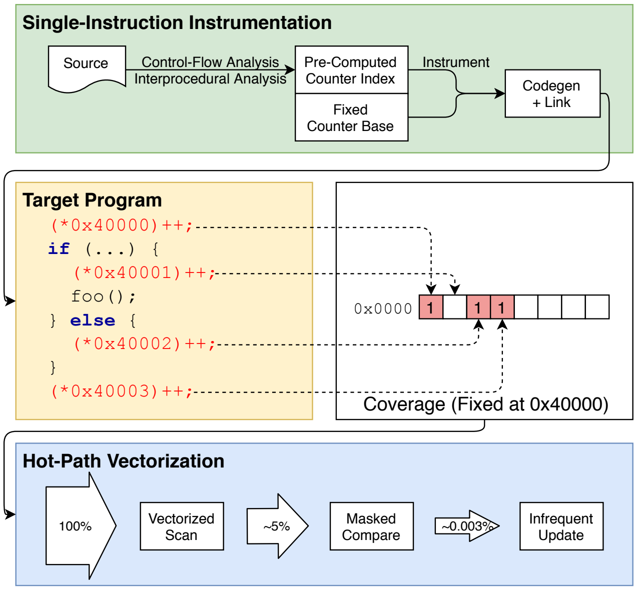
As the first row in Table 4 shows, for the coverage of the first 2,000 discovered paths, 98.67% of the processed counters were zero. In other words, executing an input only covers 871.69 counters, yet the total number of counters allocated by AFL was 65,536. Angora's instrumentation technique suffers even more, because it allocates 1MiB of memory to store coverage. The sparsity of the coverage array implies that skipping zero counters quickly during coverage analysis can be a major performance boost.

As the second row in Table 4 shows, although 99.997% of the inputs did not trigger any new program behavior, AFL still performed many computations: the first pass converted the coverage to a bitmap, and the second pass re-read it to compare with the database of unknown program states. The same applies to libFuzzer, which maintains even more statistics, including the minimum input size, the trigger frequency of each feature, the list of the rarest features, and the list of unique features covered by the current input. The analysis requires complex computations involving table lookups, linear searches, and floating-point logarithms.

The analysis logic cannot be efficiently optimized by compilers. The high-level algorithm is scattered with side effects, control-flow transfers, and data dependencies. Due to the complexity of the analysis logic, the compiler cannot perform important optimizations such as hardware-assisted loop vectorization. Only shallow optimizations, such as loop unrolling, are performed.

4 Design of RIFF

Figure 4 presents the overall design of RIFF. Similar to conventional coverage pipelines, it consists of compile-time instrumentation, runtime execution trace collection, and coverage processing.



```
(*0x40000)++;
if (...) {
  (*0x40001)++;
  foo();
} else {
  (*0x40002)++;
}
(*0x40003)++;
```

Figure 4: System overview. RIFF first instruments the program to log the execution trace at runtime. After the completion of execution, the fuzzer processes the coverage in three stages, using vectorization on the hot-path.

At compile-time, RIFF performs control-flow analysis and interprocedural analysis to pre-compute all possible control-flow edges; each edge is statically allocated a fixed counter index. The compile-time computation avoids performing the address computation at runtime. Next, RIFF inserts code to log the edge execution by incrementing the counter at the corresponding address. Finally, RIFF generates machine code with the help of the compiler's backend, without requiring runtime context saving or restoring. When the target program starts, RIFF's runtime maps the coverage counters at the fixed address specified by the compiler. The simplified instrumentation and aggregated coverage layout reduces the overhead of coverage collection. We describe the optimized instrumentation in detail in Section 4.1.

After the target program completes the execution of an input, the fuzzer enhanced with RIFF processes the coverage in three stages, where the first stage handles simple cases quickly, and the last stage handles infrequent complex cases. According to the simulation of collected coverage in Section 3.2, the first stage vectorized scan can eliminate 95.08% of the analysis cases, leaving only 4.92% of the processing job to the second stage, i.e. the masked comparison. The slowest stage only processes 0.003% of all cases.

4.1 Single-Instruction Instrumentation

As shown in Figure 2, the instrumentation code that collects coverage is expensive. Not only does it need to update the counter for each basic block, but the instrumentation code *saves and restores registers* around each counter update to preserve program logic. Moreover, the code *loads the counter base address* dynamically and *computes the counter index* by hashing the block index. RIFF reduces this code to a single instruction by performing much of this computation at compile time.

Pre-Compute Counter Index AFL uses hashing-based control-flow edge coverage. While edge-level coverage can distinguish between execution traces where block-based coverage cannot, maintaining the previous block's identifier dynamically and computing hashes at runtime is expensive. Using the compiler infrastructure RIFF performs edge-coverage computation at compile-time, and falls back to runtime computation only if static information is insufficient.

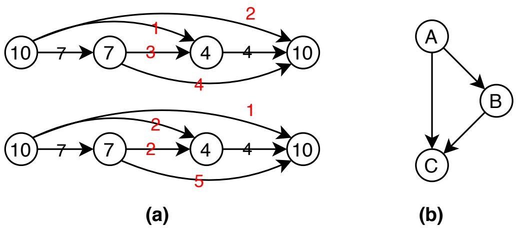


Figure 5: Problems behind raw block coverage: (a) incompleteness: multiple edge counts can map to the same block count; (b) complexity: obtaining the hit count of edge AC requires extra computation at fuzzer's side.

Figure 5 illustrates the imprecision of block-level coverage. Figure (a) shows two control-flow graphs that have different edge counts but identical block counts. In theory, for a digraph with $|V|$ vertices, there can be $\frac{|V||V-1|}{2}$ edges. Therefore, block count alone cannot determine the exact edge counts. However, in practice, the graph is very sparse, and in some cases, the edge counts can be uniquely determined by the block counts. However, calculating edge counts requires an expensive computation to solve a system of equations. As Figure (b) shows, there are three basic blocks (A, B, and C) and three edges (AB, BC, and AC). Suppose that the instrumentation scheme collects the count for basic block A, B, and C as a, b, and c respectively. While the hit count for edge AB and BC can be directly represented as a or c, the hit count of edge AC must be computed (such as $a-b$). Solving the system of equations will significantly slow down the processing at fuzzer's side.

RIFF leverages static analysis to allocate one counter for each edge. It does this by creating additional empty basic blocks when needed. As Algorithm 1 shows, if the hit count of an edge can be uniquely determined by its source or sink

Algorithm 1: Control-Flow Edge Instrumentation

```
Data: A control-flow graph $G = (V, E)$
Result: A new control-flow graph $G' = (V', E')$ and a set of
        target blocks to instrument $T \subseteq 2^{V'}$
$T \leftarrow \emptyset, V' \leftarrow V, E' \leftarrow E$ ;
for $(x, y) \in E$ do
    if $\delta^+(x) = 1$ then
        The source vertex has only one outgoing edge $(x, y)$,
          thus the hit count of $(x, y)$ equals to $x$ ;
        $T \leftarrow T \cup x$ ;
    else if $\delta^-(y) = 1$ then
        The sink vertex has only one incoming edge $(x, y)$,
          thus the hit count of $(x, y)$ equals to $y$ ;
        $T \leftarrow T \cup y$ ;
    else
        No direct representation is available ;
        Introduce a temporary vertex $t_{(x,y)}$ to represent the
          hit count of $(x, y)$ ;
        $V' \leftarrow V' \cup t_{(x,y)}$ ;
        $E' \leftarrow E' / (x, y) \cup (x, t_{(x,y)}) \cup (t_{(x,y)}, y)$ ;
        $T \leftarrow T \cup t_{(x,y)}$ ;
    end
end
```

vertex, then the block count is used for the edge. Otherwise, an empty block is allocated to represent the edge. For function calls, RIFF uses the hit count of the caller block to represent the hit count for the edge between the caller block and the callee's entry block because the counts are equal. After collecting the blocks to instrument, RIFF assigns identifiers sequentially for each block and removes instrumentation sites whose hit counts can be represented by other counters. These identifiers are used as runtime indexes for the counters in the coverage array.

Fix Counter Base AFL uses a block of shared memory for the counters. When the target program starts, the runtime library maps the shared memory into its address space and stores the base address as a global variable. While indirect addressing is flexible, computing the counter address dynamically for every basic block is inefficient. To remove extra accesses to the counter base, the address must be compile-time constants for each instrumentation site. Counter allocation with fixed addresses is done in two steps.

At the beginning of each basic block, the instrumentation code should increment its associated counter. If the base address is fixed, and the index of the array is already allocated at compile time (see Section 4.1), the address of the counter can be also computed at compile time. We can then directly increment the counter pointed by the address, using e.g., `incb $ADDR`. However, as Table 5 shows, directly encoding the target address inside the instruction requires a 7-byte instruction (scale-index-base). RIFF uses a RIP-based addressing mode [1], requiring a 6-byte instruction. Moreover, the expensive register save/restore code is no longer needed.

Table 5: Instruction Encoding of Addressing Modes

Assembly	Length	Opcode	ModRm	SIB	Disp
`incb $ADDR`	7	`0xfe`	`0x04`	`0x25`	(4 bytes)
`incb $OFFSET($rip)`	6	`0xfe`	`0x05`		(4 bytes)

Before the target program runs, the memory shared by the fuzzer should be correctly mapped to the address space of the target program. To prevent the static and dynamic linker from reusing the address for other symbols, RIFF fixes the binary's image base to the address 0x80000 (8 MiB), and reserves the address range of 0x400000 to 0x80000 for the coverage.

Indirect Control Transfers While single-instruction instrumentation is efficient, this solution cannot be used for indirect control transfers. These occur in the following instances: GNU C extensions that allow taking the address of switch labels [18], `setjmp` and `longjmp` [26], function pointers, and unwinds on C++ exceptions.

RIFF uses interprocedural control-flow analysis to discover such cases and falls back to dynamic computation. If the start of an edge representing indirect control transfer is found (e.g. `setjmp`), RIFF stores the source block ID in thread-local storage before performing the transfer. At the target of an indirect control transfer (e.g. `longjmp`), RIFF loads the source block ID and computes the counter index by hashing.

4.2 Hot-Path Vectorized Analysis

As Table 4 shows, among all coverage counters, only a small number of counters are updated by the target program; among all executions, inputs which demonstrate new program behavior are extremely rare. This observation implies that many computations performed by the fuzzer do not produce useful results. If the redundant computation is removed, the simplified logic can be accelerated using SIMD instructions (Advanced Vector Extensions on x86-64 and NEON on ARMv8).

Figure 6 demonstrates how this multi-stage processing design simplifies the logic. Stage 0 is the simplest one, which just fetches 64 bytes chunks and discards all-zero chunks. Stage 1 is invoked with the nonzero positions encoded as a mask. In Stage 1 the counters are classified as a bitmap in registers then directly compared with the database for unknown program states. The counters are discarded if no new features are discovered. Only when it is determined that the current input triggers new program behaviors is the original analysis performed by AFL invoked, in Stage 2. While this stage requires complex computations, it is rarely invoked.

[1] RIP is the instruction pointer.

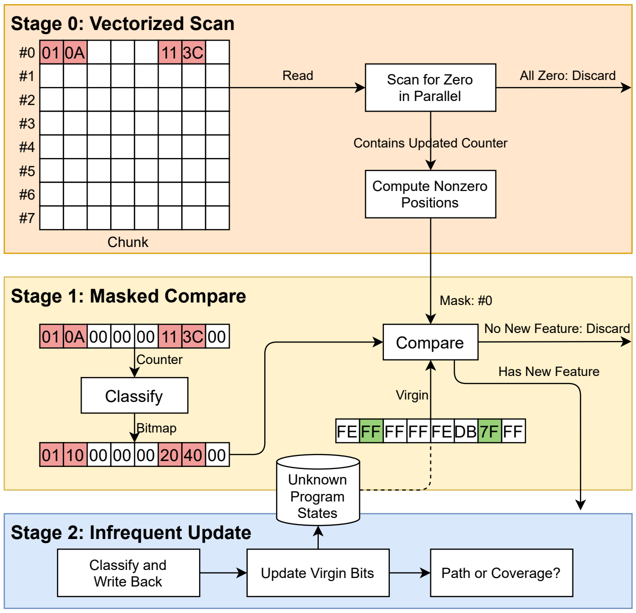


Figure 6: Processing coverage in three stages. Stage 0 filters out large chunks of zero bytes; Stage 1 checks for new coverage using masked comparisons; Stage 2 is invoked only for inputs that trigger new behaviors.

Vectorized Scan Although coverage-guided fuzzing can discover lots of code during the whole fuzz session, the covered code for a single input is lower. Because most counters are not accessed by the target program, their values stay zero after execution. Filtering out these zero counters can remove further processing stages, but the filtering operation itself requires extra computation.

To filter the zero counters efficiently, we use instructions that scan counters in parallel. Modern processors have vector processing abilities. AVX512 is a typical single-instruction-multiple-data design proposed by Intel in 2013, and it is widely supported in modern server processors. Operating on 512-bit vectors, it can compare 64 lanes of 8-bit integers in parallel (`vptestnmb`). For example, on Skylake-X based processors, it completes such a scan in 4 clock cycles. By comparison, the scalar-based processing requires 64 `testb` instructions with a latency of 1 cycle each.

The vectorized comparison encodes the comparison results inside a mask register. Each bit inside the mask register represents whether a lane inside the vector is zero. For example, if we treat 64 bytes of data chunk as 8 lanes of u64, then the result mask register contains 8 bits. If the least significant bit (0x1) is set in the mask, then the first (#0) lane is zero. Similarly, if the most significant bit (0x80) is set in the mask, then the last (#7) lane is zero. Consequently, we can skip the following tiers if all the 8 bits are set (0xff), indicating that all the lanes are zero.

Masked Compare If a chunk contains non-zero bytes, it may represent a new program behavior. Therefore, vectorized scan cannot discard the chunk and should delegate the computation to the next stage, masked compare. In this stage, the coverage is classified then compared with the database to detect new program behavior.

However, even for a non-zero chunk, it is very likely that most of the lanes are zero because of the sparsity of coverage. To remove unnecessary computation, the mask obtained from vectorized scan is used to sift the nonzero lanes: only when the mask indicates that a lane is non-zero, then the following classification is used. Otherwise, the zero lanes are discarded immediately.

For each nonzero lane, the corresponding counters are read into a register. After classifying the raw counters into bitmap using table lookups and bitwise operations, they are directly compared with the database. In most cases, the comparison will not find a difference and the bitmap is discarded. We optimize for the scenario where the bitmap is discarded to avoid updates to both the bitmap and the database.

Infrequent Update For inputs triggering interesting behavior, the processing of its coverage will reach stage 2. This stage is seldom invoked.

This stage performs the original analysis performed by standard fuzzers: first, it classifies the original counters and writes the bitmap back to memory; second, it reads the bitmap, compares it with the database, and updates the database if needed. While scanning the bitmap, it checks for changed counters and declares the run to be a “new coverage” if any are found. Otherwise, the run has discovered a “new path”.

5 Implementation

Because single-instruction instrumentation requires the precise counter address for each instrumentation point, instrumentation must be performed on the whole program, at link-time. The compiler part of RIFF is implemented on LLVM. Specifically, when compiling source files, RIFF instructs the compiler to produce LLVM bitcode instead of object files. These bitcode files are linked to the whole-program bitcode for analysis use. Next, RIFF performs instrumentation on the whole program leveraging the DominatorTreeAnalysis and BasicBlockUtils analyses, then generates machine code as a single object file. During code generation, LLVM prefers to generate 7-byte `addb` instructions over the 6-byte `incb` instructions, because the default configuration of LLVM is optimized for old or mobile-first processors, where `incb` is slower than `addb`. To force instruction selection to generate `incb` instructions, we fine-tune the LLVM target attribute by disabling the `slow-incdec` target feature.

As in conventional linking, the single object file is linked with system libraries. After this step the compiler maps the symbol denoting the start of coverage counters at a fixed address (SHN_ABS in st_shndx). As Listing 1 shows, the generated machine code only requires 6 bytes for most cases. Only on indirect transfers does RIFF fall back to the runtime hashing.

```
# Single-instruction instrumentation
incb   $INDEX(%rip)       # fe 05 ?? ?? ?? ??

# Rare case: indirect transfer (source)
mov    $PREV(%rip),%rcx   # 48 8b 0d ?? ?? ?? ??
movl   $BBID,%fs:(%rcx)   # 64 c7 01 ?? ?? ?? ??

# Rare case: indirect transfer (destination)
mov    $PREV(%rip),%rcx   # 48 8b 0d ?? ?? ?? ??
movslq %fs:(%rcx),%rax    # 64 48 63 01
xor    $BBID,%rax         # 48 35 ?? ?? ?? ??
incb   $BASE(%rax)        # fe 80 ?? ?? ?? ??
```

Listing 1: Assembly and machine code generated by RIFF.

Because vectorized coverage processing relies on the hardware support of SIMD instructions, currently we implement two variants on x86-64. If AVX512 Doubleword and Quadword Instructions (AVX512DQ) are supported, then 8 lanes of 64-bit integers are processed as a chunk. If AVX2 is supported, then the 4 lanes of 64-bit integers are processed as a chunk. Otherwise, Stage 0 is skipped entirely, and Stage 1 is executed. We implement the algorithms via intrinsic functions to take advantage of the compiler-based register allocation optimization.

6 Evaluation

To demonstrate how the reduced instruction footprint accelerates fuzzing, we evaluate the performance of RIFF on real-world settings.

For target programs, we select every program included in both Google fuzzer-test-suite and FuzzBench. Carefully picked by Google, they encompass a comprehensive set of widely-used real-world programs. For fuzzers, we select the classic industrial fuzzer AFL and the recently published MOpt.

We compile the programs with afl-clang using the default settings and compile RIFF's version with our instrumentation pipeline. For both cases we use Clang 11.0 with the same configuration (e.g., optimization level). As for the fuzzers, the baseline versions are built from the git repositories without modification. We further apply RIFF's hot-path acceleration patch to the baseline fuzzers. Note that RIFF and AFL use different instrumentation, we calibrate the raw metrics with fuzzer-test-suite's coverage binary for fairness. All the coverage used in the following analysis is based on the calibrated data.

We perform the experiments on Linux 5.8.10 with 128 GiB of RAM. The processor used is Intel Xeon Gold 6148. Its Skylake-Server microarchitecture allows acceleration with AVX2 and AVX512.

6.1 Overall Results

Figure 7 compares the time required by RIFF to reach the same coverage as AFL and MOpt respectively running for 3h, 6h, 12h, and 24h. A bar below the red line indicates a speed-up for RIFF.

The purple bars show the speedup of the long experiments run for 24 hours, where fuzzing tends to saturate (discovering few new paths). On average, to reach the final coverage of AFL and MOpt running for 24 hours, RIFF's improved versions only require 6.23 and 6.82 hours respectively. For individual programs, the improvements are consistent: even for the worst programs (freetype2 for AFL and libjpeg for MOpt), RIFF still reached the final coverage 2.1 and 0.8 hours before the baseline versions. On average, RIFF accelerates

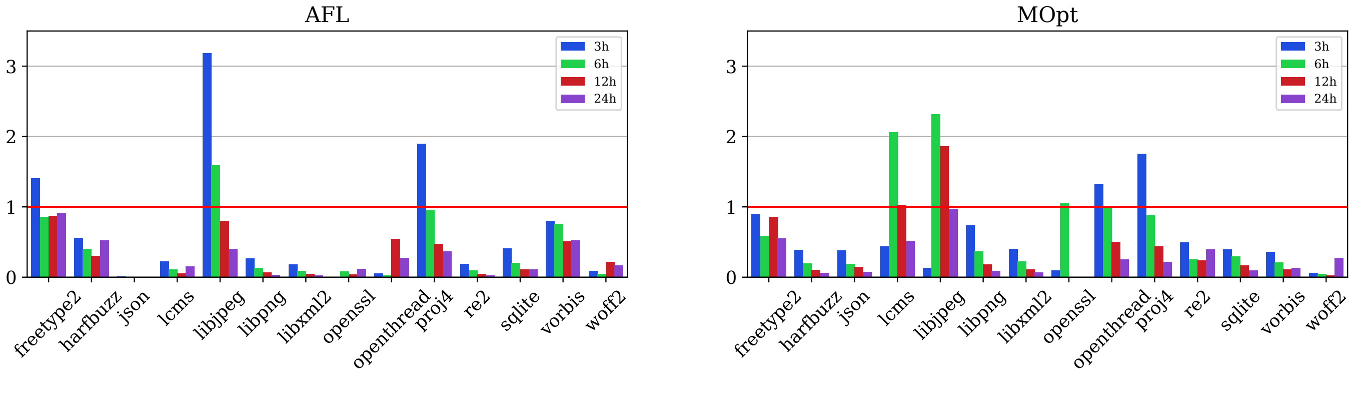


Figure 7: Normalized execution time required by RIFF to reach the same coverage as AFL and MOpt. The X axis is programs, the Y axis is the ratio between the execution times required for reaching the same coverage. A bar below the red line indicates a speed-up.

the 24-hour fuzzing by 268.74%.

The bars of 3h, 6h, and 12h show the speedup for shorter experiments. In such scenarios, saturation is less likely, and the randomness can lead to slowdowns (causing a different set of inputs to be explored). Here we can see that RIFF is still frequently performing best. For example, when fuzzing freetype2 with AFL, RIFF-based version requires 1.22 hours more to catch up with the baseline version, but its performance gradually improves as we extend the experiment, and it leads by 0.85, 1.56, and 2.10 hours at 6, 12, and 24h respectively.

Figure 8 presents the overall results after 24-hour experiments. Inside the figure, the baseline metrics from AFL and MOpt are normalized to the red horizontal line 1.0, while the corresponding metrics from RIFF's optimizations are drawn as bars. Higher bars indicate better performance.

The "covered edges" graph from Figure 8 demonstrates the overall improvement brought by RIFF. On average, RIFF improves the coverage of AFL and MOpt by 4.96% and 6.25% respectively. The improvement is consistent for individual programs: among all the 28 experiments, RIFF is best for 27. Because RIFF accelerates both the fuzzer and the target program, more executions can be completed in less time. Despite the trend of saturation for the long 24-hour trials, RIFF still managed to cover rare edges requiring a large number of executions.

The "total paths" graph from Figure 8 demonstrates that RIFF has comparably good feedback signal as the baseline versions. For most programs, RIFF improves the total number of discovered paths since it performs more execution: on average, RIFF improves the number of discovered paths by 10.79% and 15.48% over AFL and MOpt respectively. Although RIFF simplifies the computation of edge coverage, its ability in providing fuzzing signal is not reduced because of the compile-time analysis. Take re2 for example, both the baseline versions seem to discover more paths; however, paths only provide fuzz signals, thus more paths do not necessarily lead to more coverage. When the fuzz-oriented coverage is calibrated to fuzzer-test-suite's canonical coverage, RIFF-based fuzzers discover more edges.

The advantage of RIFF can be seen in the "total executions" graph. RIFF increases the number of fuzzing executions in the same amount of time to values ranging between 1.03% to 541.38%. While the randomness introduced by fuzzing algorithms can cause diminished coverage, the overall result confirms that RIFF improves the execution in general. The vastly increased number of executions can be attributed to the reduced overhead, in both the target program and the fuzzer's side.

6.2 Simplified Coverage Collection

Single-instruction instrumentation reduces the overhead of the instrumentation. To evaluate it fairly, we first fix a set of inputs, and we reuse the same inputs for all measurements for all fuzzers. For each program, we mix 1000 inputs discovered by all fuzzers; while executing the programs, we measure the time and normalize it against the non-instrumented version.

Figure 10 shows the instrumentation overhead for both afl-clang and RIFF. The figure demonstrates that the widely used instrumentation scheme afl-clang imposes heavy overhead on all the programs. Compared to the non-instrumented programs, programs instrumented by afl-clang the average execution time increases by 206.83%. The reasons can be explained by Figure 9: it executes 340.63% more instructions, which translate to 338.47% more uops and require 242.97% more L1 instruction cache refills.

RIFF reduces the footprint of instructions down to one instruction per site. On average, the coverage collection of RIFF only requires 8.40% more time to execute, while afl-clang requires 206.83% more time. In other words, RIFF reduces the overheads by 23 times. The improvement can be explained by the reduced instruction footprint: RIFF eliminates loads to counter base, shifts computation of counter index to compile-time, and removes the context saving or restoring code.

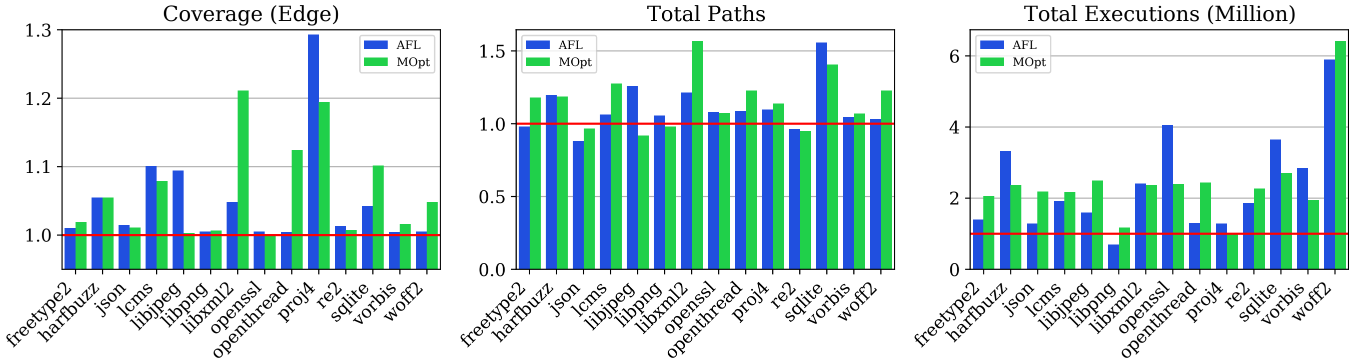


Figure 8: Normalized performance metrics for RIFF-based fuzzers after 24 hours of fuzzing. X axis is programs, Y axis is the normalized performance metric (ratio between RIFF and standard fuzzer). Bars higher than 1 (red line) indicate better performance.

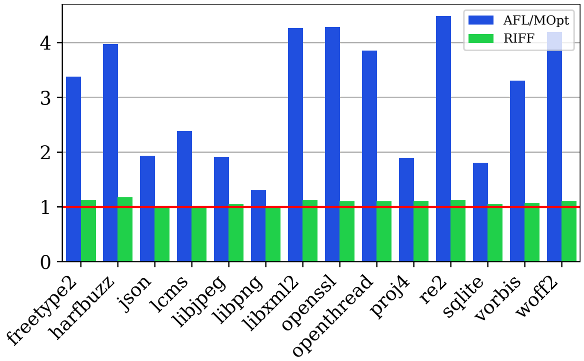


Figure 10: Normalized execution duration of fuzzed programs: time to execute 1000 on fixed inputs normalized to the time of uninstrumented programs. Lower bars indicate better performance.

6.3 Accelerated Coverage Processing

Hot-path vectorization accelerates the coverage processing at fuzzer's side. To cancel randomness from fuzzing loops and irrelevant speedups from the target programs, we extract the coverage processing routine as a library and evaluate it in isolation.

As in Section 6.2, we fix a set of inputs, then run experiments with these inputs to collect the raw coverage counter arrays. However, because all the saved inputs are rare cases which lead to new coverage, just running coverage processing routine on the saved inputs one by one exaggerates the rate of discovery. Instead, we calculate the average number of executions to discover a new input during the whole fuzz session, and run coverage processing routine on the raw coverage repeatedly this many times on the first 50 inputs. We further calculate the total processing time required to discover the first 50 inputs; we present the normalized values in Figure 11.

Figure 11 shows the benefits of hot-path vectorization. The processing time of AFL and MOpt is normalized to 1.0, shown as the red horizontal line. The bars show the processing time of RIFF.

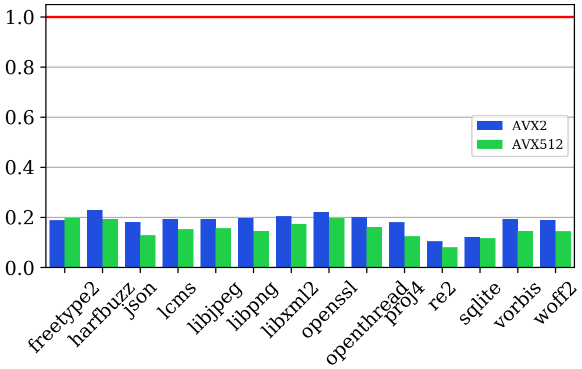


Figure 11: Coverage processing time (normalized against the baseline algorithm). Lower bars indicate better performance.

The bars for AVX2 and AVX512 of Figure 11 demonstrate RIFF's improved efficiency in coverage processing. Leveraging AVX2, RIFF uses one instruction to compare 32 coverage counters in parallel; AVX512 further extends the parallelism to 64 counters per comparison. With hardware-assisted processing, the vectorized versions improve the efficiency of the original scalar-based pipeline by 4.64x and 6.01x respectively.

7 Discussion

Currently, we only evaluated our work on x86-64 due to insufficient fuzzer support on other platforms. For example, AFL only provides experimental ARM support via QEMU. While the implementation is target-dependent, the general idea applies to all platforms: the minimal instrumentation logic can be implemented with just 4 instructions on ARMv8 or RISC-V systems; the vectorized coverage processing can use ARMv8 NEON ISA instead of AVX2 or AVX512.

As for the applicability of our improvement, we only applied our work to the industrial fuzzer AFL and the academic work MOpt due to limited resources. While they use different fuzzing algorithms, the improvements brought by RIFF are similar (see Figure 7 and 8). Our work can be easily adapted

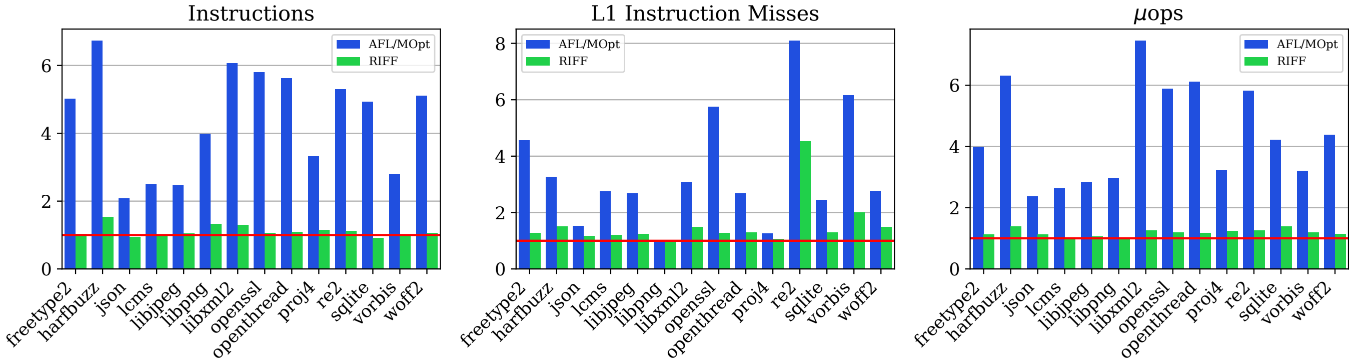


Figure 9: Metrics for RIFF-Based Programs

to more fuzzers. For example, developers of AFL++ [17] have adapted our work to their code base and conducted independent third-party evaluations with Google FuzzBench [30]. According to the result [5], our modification (labeled as "skim") was the best-performing one among all the 10 variants.

8 Related Work

8.1 Vectorized Emulation

Snapshot fuzzing [2, 16] tests the target program from partially executed system states. The program is broken into small pieces of code, and the execution of the code is emulated by the hypervisor. Because the emulation simplifies the logic to execute, multiple system states can be emulated simultaneously with vectorization.

Rather than accelerating the emulation, RIFF is focused on coverage pipeline: first, RIFF's single-instruction instrumentation combined with vectorization-based emulation and checkpointing accelerates the execution of target programs; RIFFs hot-path optimization also accelerates fuzzer's coverage analysis.

8.2 Enriching Semantics of Coverage

Since the coverage quality is crucial for input prioritization, numerous approaches have been proposed by academia which bring more semantics to coverage. For example, VUzzer [33] stores call stack information to coverage, and Angora enhances coverage with search targets [11]. Sometimes, researchers introduce data-flow features to conventional control-flow-oriented coverage. For example, Steelix [28] stores branch comparison operators, Dowser [22] records branch constraints, GreyOne [19] imports constraint conformance to tune the evolution direction of fuzzing, and [37] traces memory accesses. While these techniques can help a fuzzer to choose better inputs, the complexity introduces heavy overhead and severely limits the execution speed.

8.3 Reducing Overhead of Coverage

Not instrumenting the program eliminates overhead altogether. Researchers utilize debugger breakpoints to detect the first time a block has been covered with hardware support [32, 43]; in this scheme, only the first occurrence of a block has extra cost. However, the information of the number of times that a block has been covered is lost without any instrumentation; on the contrary, RIFF does not reduce the quality of feedback.

Another idea is to reduce the number of instrumentation points [25]. However, the cost of each instrumentation point is still high because it still needs to maintain the edge information by hashing. RIFF simplifies instrumentation points to single instructions; it is not focused on reducing the amount of instrumentation points.

8.4 Reducing Overhead of Operating System

Traditionally, fuzzing is targeted at utility programs where each execution requires `fork` a new process and then `execve` to the new binary. To remove the costly `execve`, AFL implements fork server mode [39]. To reduce the cost of `fork`, Xu et al. [38] designs a new system call `snapshot` to restore the execution state in-place. To further reduce the number of invocations of `fork`, AFL implements persistent mode [41], where a program runs continuously without restart. libFuzzer further eliminates other expensive system calls with in-process fuzzing: if the fuzz target is library, then the fuzzing is performed in-memory.

With these operating system works, the major overhead introduced by context switches of system calls has been greatly reduced. Consequently, the cost of execution has become another prominent problem. RIFF reduces the cost by reducing the instruction footprint of the coverage pipeline.

9 Conclusion

In this paper, we present RIFF to reduce the instruction footprint for fuzzing. We first observe that the coverage pipeline in fuzzing slows down the overall execution speed. We find that the heavy instruction footprint is the root cause: for target programs, the expensive instructions collect coverage inefficiently; for fuzzers, the unnecessary instructions cannot fully exploit the processor's ability. We implement RIFF to reduce the instruction footprint and achieve a 268.74% speedup for the 24-hour experiments. RIFF is being integrated by popular fuzzers such as AFL and AFL++ for use in industry and has shown significant improvements over the state of the art.

Acknowledgments

We sincerely appreciate the shepherding from Mihai Budiu and Eric Schkufza. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706200) and Ali-Tsinghua Database Testing Research Project (NO. 20212000070).

References

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016. [Online; accessed 15-May-2021].

[2] Ron Aquino. Mitigating vulnerabilities in endpoint network stacks - Microsoft Security. https://www.microsoft.com/security/blog/2020/05/04/mitigating-vulnerabilities-endpoint-network-stacks/, 2020. [Online; accessed 29-April-2021].

[3] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. Open sourcing ClusterFuzz. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html, 2019. [Online; accessed 15-May-2021].

[4] Chromium Authors. libFuzzer in Chrome. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/testing/libfuzzer/README.md, 2017. [Online; accessed 15-May-2021].

[5] FuzzBench Authors. FuzzBench: 2020-12-18 report. https://www.fuzzbench.com/reports/experimental/2020-12-18/index.html, 2020.

[6] GCC Authors. Instrumentation options (using the GNU compiler collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html, 2021. [Online; accessed 29-April-2021].

[7] LLVM Authors. SanitizerCoverage — Clang 12 documentation. https://clang.llvm.org/docs/SanitizerCoverage.html, 2021. [Online; accessed 29-April-2021].

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.

[9] Foster Brereton. Binspector: Evolving a security tool. http://web.archive.org/web/20181020154300/https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html, 2015. [Online; accessed 15-May-2021].

[10] Mark J Buxton. Haswell new instruction descriptions now available! https://software.intel.com/content/www/us/en/develop/blogs/haswell-new-instruction-descriptions-now-available.html, 2020. [Online; accessed 15-May-2021].

[11] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[12] Intel Corporation. Intel® Intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/, XML file available at https://software.intel.com/sites/landingpage/IntrinsicsGuide/files/data-3.5.4.xml in 6,096,526 bytes, 2020. [Online; accessed 15-May-2021].

[13] Joe W. Duran and Simeon C. Ntafos. A report on random testing. In Seymour Jeffrey and Leon G. Stucki, editors, *5th International Conference on Software Engineering (ICSE)*, pages 179–183, 1981.

[14] Michael Eddington. Peach Fuzzer: Discover unknown vulnerabilities. http://web.archive.org/web/20210121202148/https://www.peach.tech/, 2015. [Online; accessed 15-May-2021].

[15] Jonathan Metzman et al. fuzzbench/fuzzer.py at master · google/fuzzbench. https://github.com/google/fuzzbench/blob/master/fuzzers/afl/fuzzer.py, 2020.

[16] Brandon Falk. Vectorized emulation: Hardware accelerated taint tracking at 2 trillion instructions per second | Gamozo Labs Blog. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html, 2018. [Online; accessed 29-April-2021].

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[18] Inc. Free Software Foundation. Labels as values (using the GNU compiler collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html, 2020. [Online; accessed 15-May-2021].

[19] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: data flow sensitive fuzzing. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium*, pages 2577–2594, 2020.

[20] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[21] Google. google/fuzzer-test-suite: Set of tests for fuzzing engines. https://github.com/google/fuzzer-test-suite, 2020. [Online; accessed 15-May-2021].

[22] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: A guided fuzzer for finding buffer overflow vulnerabilities. *login Usenix Mag.*, 38(6), 2013.

[23] Marc Heuse. AFLplusplus/coverage-64.h at stable · AFLplusplus/AFLplusplus. https://github.com/AFLplusplus/AFLplusplus/blob/stable/include/coverage-64.h, 2021. [Online; accessed 13-May-2021].

[24] Sam Hocevar. zzuf – Caca Labs. http://caca.zoy.org/wiki/zzuf, 2007. [Online; accessed 15-May-2021].

[25] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018.

[26] IEEE and The Open Group. setjmp.h - stack environment declarations. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/setjmp.h.html, 2018. [Online; accessed 15-May-2021].

[27] Cheick Keita, Marina Polishchuk, Patrice Godefroid, William Blum, Stas Tishkin, Dave Tamasi, and Marc Greisen. Microsoft security risk detection ("Project Springfield"). https://www.microsoft.com/en-us/research/project/project-springfield/, 2015. [Online; accessed 26-January-2018].

[28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium*, pages 1949–1966, 2019.

[30] Jonathan Metzman, Abhishek Arya, and Laszlo Szekeres. FuzzBench: Fuzzer benchmarking as a service. https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html, 2020. [Online; accessed 13-May-2021].

[31] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[32] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019.

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[34] James Reinders. Intel® AVX-512 instructions. https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html, 2017. [Online; accessed 15-May-2021].

[35] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., USA, 1st edition, 2008.

[36] Dmitry Vyukov. google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2015. [Online; accessed 15-May-2021].

[37] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[38] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2313–2328, 2017.

[39] Michal Zalewski. Fuzzing random programs without execve(). http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014. [Online; accessed 15-May-2021].

[40] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl, 2015. [Online; accessed 15-May-2021].

[41] Michal Zalewski. New in AFL: persistent mode. http://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html, 2015. [Online; accessed 15-May-2021].

[42] Michal Zalewski. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2015. [Online; accessed 15-May-2021].

[43] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 858–870, 2020.