# Scalable Fuzzing of Program Binaries with E9AFL

Xiang Gao[§]
*National University of Singapore*
Singapore
gaoxiang@comp.nus.edu.sg

Gregory J. Duck[§]
*National University of Singapore*
Singapore
gregory@comp.nus.edu.sg

Abhik Roychoudhury
*National University of Singapore*
Singapore
abhik@comp.nus.edu.sg

***Abstract*—Greybox fuzzing is an effective method for software testing. Greybox fuzzers, e.g., AFL, use instrumentation to collect path coverage information to guide the test generation. The instrumentation is usually inserted by a modified compiler toolchain, meaning that the program must be recompiled in order to be compatible with greybox fuzzing. When source code is unavailable, or for projects with complex build systems, recompilation is not always feasible. In this paper, we present E9AFL, a fast and scalable tool that inserts AFL instrumentation to program binaries. E9AFL is built on top of the E9Patch static binary rewriting tool. To combat the overhead caused by binary instrumentation, E9AFL develops a set of optimization strategies. Evaluation results show that E9AFL outperforms existing binary instrumentation tools and achieves comparable performance with the compile time instrumentation.**

**Open source tool: https://github.com/GJDuck/e9afl**
**Tool demo video: https://youtu.be/bVyADCGZlnw**

***Index Terms*—Fuzzing, binary rewriting**

## I. Introduction

Greybox fuzzing is a proven and effective method for software testing. Popular greybox fuzzing tools, such as AFL [1], work by inserting instrumentation into the program in order to collect path coverage information for guiding the fuzzing process. The instrumentation can be inserted at compile-time using a specially modified compiler (e.g., `afl-gcc`). However, in scenarios where the source code is unavailable, compile-time instrumentation is not possible, meaning that the instrumentation must be inserted directly into binary code. Existing techniques instrument binaries either dynamically, such as with AFL-qemu [2], or statically using binary rewriting, such as AFL-dyninst [3]. However, each approach has its drawbacks. For example, dynamic binary instrumentation incurs high overheads due to the run-time translation—up to $5\times$ slower for AFL-qemu. Static binary instrumentation may also be inaccurate and introduce errors, which can manifest as false positives during the fuzzing process. Either way, binary fuzzing tends not to scale to very large/complex software.

To address the above limitations, we propose a fast and scalable binary AFL instrumentation tool E9AFL, which is based on the E9Patch [4] static binary rewriting system. Unlike other static binary rewriting tools, E9Patch can rewrite stripped binaries without heuristics, meaning that E9Patch can scale to very large/complex software, such as Google Chrome [5] and Firefox. Specifically, E9Patch uses a *trampoline*-based rewriting methodology combined with *instruction punning* [4], [6] and extensions to insert instrumentation into arbitrary locations. However, this methodology may incur significant fuzzing overheads since trampolines break the code contiguity and this can generate excess page faults. To alleviate the overhead, E9AFL introduces three core optimization strategies: *Trampoline ordering*, *Instruction selection* and *Bad block elimination* that help mitigate fuzzing overhead. We evaluate E9AFL on a set of subjects from the Fuzzbench benchmark. Our experimental results show that E9AFL outperforms the existing AFL-qemu and AFL-dyninst tools in terms of speed and coverage. Furthermore, we show that E9AFL runs at 77.0% of the speed of `afl-gcc` with comparable code coverage. Finally, we demonstrate the scalability of E9AFL by fuzzing the Google Chrome binary.

[§]Equal contribution

## II. Tool Usage

The E9AFL tool is designed for ease-of-use. Most binaries can be instrumented using a simple command:

```
$ e9afl program
```

This will generate an instrumented `program.afl` binary that can be fuzzed using the standard AFL toolchain. For example, we can instrument and fuzz the `readelf` binary as follows:

```
$ e9afl readelf
$ afl-fuzz -i in/ -o out/ -- ./readelf.afl -a @@
```

No other additional step or special set up is required.

We note the ease-of-use compared to AFL's source-level instrumentation. To instrument a program at source-level, it is necessary to create a correct compiling environment and build it using a specially modified compiler (e.g., `afl-gcc`). For software with complex build systems, e.g., Chrome, creating a compilation environment and modifying the build system can be a non-trivial task. In contrast, E9AFL is "push-button", and can instrument pre-built binaries via a single command.

## III. Tool Design

E9AFL is designed to be an easy-to-use tool that automatically inserts AFL instrumentation into existing binary code.

*Static Binary Rewriting:* E9AFL is built on top of the E9Patch [4] static rewriting system for Linux `x86_64` binaries. E9Patch uses a *trampoline-based* rewriting methodology, meaning that selected instructions are replaced by jumps to "trampolines" that implement the AFL instrumentation. An example is illustrated in Figure 1. Here, selected instructions that are highlighted in Figure 1 (a) are replaced by jump instructions that divert control to the trampolines illustrated in Figure 1 (b). The trampolines implement the instrumentation (as well as displaced instruction), before returning control-flow back to the main program. For short instructions, E9Patch will use *instruction punning* [4] to insert the jump. The trampoline-based methodology of E9Patch has shown to be general purpose and highly scalable, and can be used to rewrite very large/complex software, including the Chrome/Firefox web browsers [4] with a $>100MB$ binary size.

The main disadvantage of trampoline-based rewriting is performance, since the extra jumps to/from trampolines incur additional runtime overheads. An alternative is "inline" binary rewriting, which attempts to insert the instrumentation directly into existing the instruction stream [7], [8]. However, the inline rewriting has known limitations, such as requiring offsets to be corrected in the rewritten binary—which is an undecidable problem in the general case.

E9AFL is implemented as a plugin to the E9Patch frontend. The E9AFL plugin takes as input a disassembly of the input binary (from the frontend), and outputs an AFL instrumentation trampoline template, the AFL runtime, and a set of instrumentation locations. This information is then passed to E9Patch, which then generates the AFL instrumented binary.

*Trampoline template:* The basic trampoline template used by E9AFL is as follows:

```
 ...                                # Save state
mov  prev_loc,%eax                  # Load prev_loc
xor  $cur_loc,%eax
incb AREA(%eax)                     # AREA[cur_loc^prev_loc]++
movl $cur_loc>>1,prev_loc           # Set prev_loc
 ...                                # Restore state
```

The trampoline template implements the classic AFL instrumentation (i.e., `AREA[cur_loc ^ prev_loc]++`). Here: `AREA` is the AFL shared trace map, `prev_loc` is the previous location (*Thread Local Storage*), and `cur_loc` is the current location (a trampoline-specific constant). The trampoline also includes some boiler-plate code for saving/restoring the CPU flags and the `%rax` register, which are used by the instrumentation. This boilerplate is similar to code inserted by `afl-gcc` for the same purpose. The instrumentation itself is essentially counting edge transitions (`prev_loc`, `cur_loc`), which are stored in a memory region (`AREA`) shared with the `afl-fuzz` tool. This allows AFL to detect *path coverage* necessary for greybox fuzzing.

*Runtime injection:* The AFL runtime, including the fork server and `AREA` initialization, must also be injected into the instrumented binary. For this, we use an E9Patch feature that allows for user code to be injected into the rewritten binary during program initialization (i.e., before `main()` is called). The injected runtime is derived from the standard AFL runtime, with some minor modifications.

*Determining instrumentation locations:* As per source-level AFL, the instrumentation ought to be inserted once per basic-block. To find the set of basic-blocks, E9AFL implements a lightweight control-flow recovery analysis that finds all likely

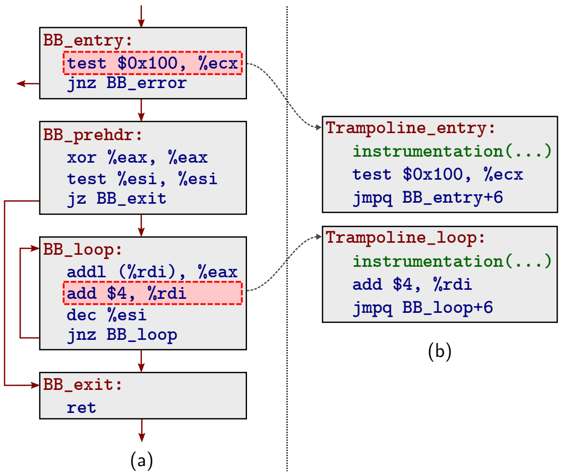

Fig. 1. Example of trampoline-based binary rewriting (a)original Control Flow Graph, and (b) *trampolines* that instrument selected instructions (highlighted).

jump targets from the input binary, including all direct targets, and all likely indirect targets by analyzing the data segments for *jump tables* and *code pointers*.

The accurate recovery of control-flow information is an undecidable problem in the general case. However, for the application of AFL instrumentation, the recovered control-flow information need not be perfectly accurate. In the case of an *overapproximation* (i.e., superfluous jump targets), this may result in more instrumentation than is strictly needed, resulting in higher overheads but is otherwise harmless. In the case of an *underapproximation* (i.e., jump targets missed), this may result in less accurate coverage information being passed to the fuzzer, meaning that `afl-fuzz` may not detect some new paths that could be detected otherwise. Nevertheless, the fuzzing process can still generate useful results even with a slight loss of accuracy. E9AFL uses a heuristic-based control-flow recovery that is reasonably accurate for most programs compiled using standard compilers (e.g. `gcc`). In the case of *Position Independent Executables* (PIEs), the accuracy is further improved, as the analysis can use ELF *relocations* to accurately identify code pointers in data segments.

Once all jump targets are identified, the set of basic-blocks can be derived. The instrumentation is inserted into each basic-block entry by issuing E9Patch instruction patching commands with the AFL instrumentation template. E9Patch completes the rewriting process and outputs an instrumented binary.

## IV. Tool Optimization

The basic E9AFL design will suffer from poor fuzzing throughput (execs/sec), meaning that some optimization is necessary. We use insights from the recent FuZZan [9] work, namely, that fuzzing performance is dominated by startup/teardown costs after `fork()`, and optimize accordingly.

*Startup/teardown costs:* In normal operation, fuzzers such as AFL create multiple instance executions of the target program, one for each generated input. To do so, AFL uses a *fork server*, which is essentially a *Remote Procedure Call*

(RPC) loop that is injected into the target program. Whenever a new input is ready for testing, `afl-fuzz` will instruct the target program (via a RPC) to make a copy of itself using the `fork()` system call. Here, `fork()` essentially duplicates the calling process into a *parent* and *child* process.[1] The child process executes the test case, collects coverage information via AFL instrumentation, and either exits normally or abnormally (i.e. crashes). The parent process will wait for the child to complete, and then send the resulting exit status back to the main `afl-fuzz` process before waiting for the next RPC.

The `fork()` system call is a relatively slow operation and this one of the main bottlenecks for overall fuzzer throughput (execs/sec). Modern versions of Linux attempt to optimize `fork()` by avoiding the copying of memory, including page table entries, as much as possible. For example, any page table entry corresponding to a file mapping will not be explicitly copied during a `fork()` operation. Rather, only if the child process actually accesses the mapping, a *page fault* will be generated, allowing the kernel to set up the corresponding page table entries lazily. In the context of fuzz testing, these page faults are a major contributing factor to the *startup costs* of the child process, and this can be a dominant factor in overall fuzzer performance [9].

*Trampolines and page faults:* With these insights, we can optimize the basic E9AFL design. One of the major sources of page faults in the rewritten binary are the trampolines used to implement the AFL instrumentation. Thus, to minimize page faults, our strategy will be to (1) make trampoline memory as contiguous as possible, and (2) remove trampolines if possible. To do so, we implement three main optimizations:

1) *Trampoline ordering*: allocate trampolines in order;
2) *Instruction selection*: select instructions which allow for better trampoline ordering; and
3) *Bad block elimination*: attempt to eliminate redundant instrumentation that will likely cause page faults.

*Trampoline ordering:* The idea of trampoline ordering is very simple: we contiguously allocate trampolines in the same order as the corresponding patched instructions. Thus, the same code regions will be mapped to the same trampoline memory, minimizing page faults.

However, for short instructions $<5$ bytes (the size of a `jmpq` instruction), E9Patch uses *instruction punning* to insert the trampoline. This means that E9Patch does not have complete control over trampoline placement, meaning that some trampoline fragmentation will still occur. To mitigate this, we can optimize which instructions are selected for instrumentation.

*Instruction selection:* Traditionally, the AFL instrumentation is inserted at the start of each basic block. However, the instrumentation can also be inserted elsewhere in the basic block and preserve the same functionality. E9AFL applies a simple *instruction selection* algorithm to choose an instruction with size $\geq 5$ bytes if available, allowing the *trampoline ordering* optimization to be applied to more basic blocks.

[1]See the manpage of `fork`.

Note, however, that not all basic blocks will have an instruction with size $\geq 5$ bytes. We define these to be *bad blocks*, since the *trampoline ordering* optimization cannot be applied, meaning that the corresponding trampoline is more likely to generate a page fault and slow down fuzzing. To mitigate this, we can attempt to eliminate the trampolines for bad blocks altogether.

*Bad block elimination:* Sometimes the AFL instrumentation for a given basic-block is *redundant*, meaning that the instrumentation can safely be eliminated without affecting path coverage. For example, suppose that all paths through block $A$ must pass through block $B$, and vice versa. Then only one of block $A$ or $B$ needs to be instrumented, since the path through one implies the path through the other. This can be generalized to the *path differentiation problem*, i.e., what is the minimum number of *Control Flow Graph* (CFG) vertices (i.e. blocks) that need to be marked (i.e. instrumented) such that all paths through the CFG can still be differentiated? Tools such as InsTrim [10] use this idea to optimize fuzzing by removing as much instrumentation as possible (as much as 80% can be removed on average). However, in the E9AFL context, our main insight is that "quality" is more important than "quantity". Specifically, we should preferentially eliminate the instrumentation for *bad* blocks only, since these blocks are the main source of additional page faults that slow down fuzzing.

The *bad block elimination* optimization uses an algorithm with similar aims to that of InsTrim [10]. First, the algorithm builds the CFG from the recovered set of basic blocks. Next, the algorithm marks each block as either *optimized* or *unoptimized*, where *optimized* means that the block should not be instrumented. Initially, the blocks will be marked as follows:

1) Good blocks are initially marked as *unoptimized*.
2) Bad blocks that are potential indirect jump/call targets are also initially marked as *unoptimized*.
3) All other bad blocks are initially marked as *optimized*.

Here, 2) is a simplification that removes the need for paths to be traced over indirect jumps/calls. The remainder of the algorithm attempts to find a solution to the path differentiation problem, and works by constructing all sub-paths $\sigma = \langle A \to \ \dots \ \to B\rangle$ though the CFG such that (1) $A$ and $B$ are *unoptimized*, and (2) all intermediate blocks between $A$ and $B$ are *optimized*. Note that loops ($A{=}B$) are allowed, and $B$ can be considered an *unoptimized* pseudo-block in the case where the last edge is an indirect call/jump, as per 2) from above. The path differentiation property is violated if there exists two (or more) distinct sub-paths:

$$\sigma_1 = \langle A \to \ \dots \ \to B\rangle \qquad \text{and} \qquad \sigma_2 = \langle A \to \ \dots \ \to B\rangle$$

for the same $(A, B)$ pair. To restore the property, the algorithm will mark an intermediate *optimized* block from $\sigma_1/\sigma_2$ as *unoptimized*. This process is repeated until no such sub-path pairs $(\sigma_1, \sigma_2)$ exist.

*Example:* An example of optimized code is illustrated in Figure 1. Here, `BB_entry` is a good block (since the `test`

TABLE I
THE EXPERIMENTAL RESULTS. HERE speed IS THE FUZZING SPEED (EXECUTIONS PER SECOND) AND cov (%) IS THE LINE COVERAGE.

| Subject | AFL-gcc | | AFL-qemu | | AFL-dyninst | | E9AFL-`O0` | | **E9AFL** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | speed | cov | speed | cov | speed | cov | speed | cov | **speed** | **cov** |
| FreeType | 1148 | 35.6 | 279 | 31.3 | 647 | 34.8 | 71 | 30.2 | **745** | **35.8** |
| `libjpeg` | 1444 | 5.3 | 455 | 4.8 | *n.r.* | *n.r.* | 420 | 4.9 | **1273** | **5.1** |
| `libpng` | 1465 | 27.4 | 302 | 23.4 | 885 | 23.4 | 495 | 22.8 | **1262** | **23.4** |
| `libxml2` | 958 | 9.9 | 76 | 7.8 | *n.r.* | *n.r.* | 152 | 5.7 | **601** | **9.9** |
| Vorbis | 1032 | 30.3 | 252 | 27.7 | 582 | 30.1 | 300 | 29.8 | **905** | **30.2** |
| G.Mean | 100% | 17.3 | 19.8% | 15.0 | - | - | 19.5% | 14.2 | **77.0%** | **16.6** |
| Chrome | n/a | n/a | *n.r.* | *n.r.* | *n.r.* | *n.r.* | 0.12 | n/a | **0.51** | **n/a** |

instruction is $\geq$5 bytes) and is marked as *unoptimized*. The remaining `BB_prehdr`/`BB_loop`/`BB_exit` blocks are bad (since all other instructions are $<$5 bytes) and are initially marked as *optimized*. However, the path differentiation property is violated by the following sub-paths:

$$\sigma_1 = \langle \texttt{BB\_entry} \rightarrow \texttt{BB\_prehdr} \rightarrow \texttt{BB\_exit} \rangle$$
$$\sigma_2 = \langle \texttt{BB\_entry} \rightarrow \texttt{BB\_prehdr} \rightarrow \texttt{BB\_loop} \rightarrow \texttt{BB\_exit} \rangle$$

Path differentiation will be restored by the algorithm by marking `BB_loop` as *unoptimized*. The optimized Figure 1 code only uses two trampolines, whereas four (non-contiguous) trampolines would be required under the basic unoptimized design (one for each basic block).

## V. EXPERIMENTS

We evaluate the efficiency and effectiveness of E9AFL against five subjects (FreeType, `libjpeg`, `libpng`, `libxml` and Vorbis) from FuzzBench [11]. To test scalability, we also evaluate E9AFL against the Google Chrome binary [5]. Our evaluation considers three comparable techniques: AFL-gcc, the original compile time AFL instrumentation; AFL-qemu [2], a binary AFL implementation based on the QEMU emulator [12]; and AFL-dyninst [3], a binary AFL instrumentation based on the Dyninst [13] binary rewriting tool. In addition to E9AFL with full optimization enabled, we also evaluate E9AFL-`O0` with optimization disabled. Except for AFL-gcc, which requires source code, all other tools directly instrument binaries. All experiments are run on an Intel Xeon CPU E5-2660 2.00GHz processor with 64GB of memory, and each experiment uses a timeout of 24 hours.

Our evaluation is shown in Table I with the main result highlighted in **bold**. Here, *speed* is the number of executions per second, *cov* (%) is the line coverage, and *n.r.* (*no result*) means that the corresponding instrumentation/tool failed.

*a) Fuzzing Efficiency:* Overall we see that AFL-qemu is quite slow, and only runs at 19.2% of the speed (execs/s) of the baseline AFL-gcc. In contrast, with full optimization enabled, E9AFL runs at 77.0% of the speed. This means that E9AFL achieves nearly the same performance as AFL-gcc without the need for the program to be recompiled from source code. Finally, AFL-dyninst proved to be less reliable, with two test subjects failing (AFL-dyninst fails to instrument `libxml2`, and the instrumented `libjpeg` would crash on some benign inputs). Regardless, E9AFL still achieves a better performance than AFL-dyninst for the non-failing test subjects.

*b) Fuzzing Effectiveness:* In terms of effectiveness, AFL-gcc, AFL-qemu, and E9AFL achieve an overall 17.3%, 15.0%, and 16.6% line coverage, respectively. Our results show that E9AFL can generate comparable code coverage to that of AFL-gcc. Compared with the other binary-only instrumentation tools, including the non-failing AFL-dyninst subjects, E9AFL achieves the overall best code coverage.

*c) Scalability:* For scalability, we find that E9AFL is the only tool that can successfully fuzz Chrome. Afl-gcc is not applicable since the source code of Chrome is unavailable (Chrome is closed source), AFL-qemu fails since the QEMU emulator does not support some syscalls used by Chrome, and AFL-dyninst fails because Dyninst fails to correctly disassemble the Chrome binary. E9AFL can successfully instrument the Chrome binary, which can then be fuzzed using AFL under *headless mode* (i.e., no user interface):

```
$ afl-fuzz ... -- ./chrome.afl --headless @@
```

We remark that Chrome is a large multi-threaded binary with high startup overheads, so it does not make an ideal fuzz target. Nevertheless, `afl-fuzz` can fuzz the instrumented Chrome with appropriate memory (`-m`) and timeout (`-t`) limits.

*d) Optimization:* Finally, we evaluate the fuzzing performance before/after the optimization has been applied. Here, the E9AFL-`O0` column from Table I represents the results before optimization. Overall, we see that E9AFL-`O0` runs at a mere 19.5% of the speed of AFL-gcc, which essentially replicates the slow performance of AFL-qemu. In contrast, the fully optimized E9AFL runs at 77.0% of the speed of AFL-gcc, or nearly $4\times$ the performance of the unoptimized version. These results show that the optimization is not only effective, but is also essential for achieving a good fuzzing performance and code coverage result.

## VI. CONCLUSION

Several AFL binary instrumentation tools have been proposed. Some tools are slow [2], use modified toolchains [14], have limited binary support (e.g., *position-independent-executable* only) [7], or may introduce false positives (or other binary rewriting errors) [3], [8]. In this paper we presented E9AFL—a new tool for automatically inserting AFL instrumentation into existing binary code. We show that E9AFL achieves 77.0% of the performance of AFL-gcc with comparable code coverage. However, unlike AFL-gcc, E9AFL does not require recompilation nor assume the availability of source code. Compared to other binary AFL solutions, E9AFL significantly improves the speed and code coverage, does not introduce false positives or other rewriting errors, and can scale to very large programs such as Chrome [5].

ACKNOWLEDGEMENTS

This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by the National Research Foundation (NRF) Singapore under National Cybersecurity R&D (NCR) programme.

## References

[1] M. Zalewski. (2021) American Fuzzy Lop. [Online]. Available: https://lcamtuf.coredump.cx/afl

[2] A. Griffiths and M. Zalewski. (2021) AFL QEMU Mode. [Online]. Available: https://github.com/google/AFL/blob/master/qemu_mode/README.qemu

[3] Cisco Talos. (2021) AFL Dyninst. [Online]. Available: https://github.com/talos-vulndev/afl-dyninst

[4] G. Duck, G. Xiang, and A. Roychoudhury, "Binary Rewriting without Control Flow Recovery," in *Programming Language Design and Implementation*. ACM, 2020.

[5] Google. (2021) Google Chrome Web Browser. [Online]. Available: https://www.google.com/chrome

[6] B. Chamith, B. Svensson, L. Dalessandro, and R. Newton, "Instruction Punning: Lightweight Instrumentation for x86-64," in *Program Design and Implementation*. ACM, 2017.

[7] S. Dinesh, N. Burow, D. Xu, , and M. Payer, "RetroWrite : Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *Security and Privacy*. IEEE, 2020.

[8] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing," in *Security Symposium*. USENIX, 2021.

[9] Y. Jeon, W. Han, N. Burow, and M. Payer, "FuZZan: Efficient Sanitizer Metadata Design for Fuzzing," in *Annual Technical Conference*. USENIX, 2020.

[10] C. Hsu, C. Wu, H. Hsiao, and S. Huang, "Instrim: Lightweight instrumentation for coverage-guided fuzzing," in *NDSS, Workshop on Binary Analysis Research*. Internet Society, 2018.

[11] Google. (2021) FuzzBench. [Online]. Available: https://github.com/google/fuzzbench

[12] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Annual Technical Conference*. USENIX, 2005.

[13] A. Bernat and B. Miller, "Anywhere, Any-time Binary Instrumentation," in *Workshop on Program Analysis for Software Tools*, 2011.

[14] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting," in *Security and Privacy*. IEEE, 2021.