CS 4264
Principles of Computer Security

November 20, 2024
Project 5: Fuzzing

Project 5: Fuzzing

This project is due on **December 11, 2024** at **11:59 p.m.** and counts for 8% of your course grade. Late submissions will be penalized by 10% plus an additional 10% every 5 hours until received. Late work will not be accepted after 24 hours past the deadline. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is a group project; you will work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If you have trouble forming a team, post to Piazza's partner search forum.

The code and other answers your group submits must be entirely your own work, and you are bound by the Honor Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via Canvas, following the submission checklist below. Please coordinate carefully with your partner to make sure at least one of you submits on time.

Introduction

This project will introduce you to automated vulnerability discovery in software through coverage-guided fuzz testing, a.k.a. fuzzing. The project starts by having you implement the world's simplest fuzzer to help you understand the basic operation of a fuzzer, then progressively tasks you with making focused improvements that increase the performance and effectiveness of your fuzzer. The result of this project is a fuzzer that is among the world's fastest, most effective fuzzers.

Objectives

- Understand the fundamentals of how coverage-guided fuzzers work.
- Understand what a fuzzer is good at and what it struggles with.
- Understand the sources of overhead and opportunities for improvement in a fuzzer's design.
- Gain familiarity with the state-of-research in high-performance fuzzing.

Read this First

This project asks you to find bugs in software targets that we provide you. While it is acceptable—in fact, encouraged—to test other pieces of software with your fuzzer, using the discovered security vulnerabilities to attack others' systems without authorization is prohibited by law and university policies and may result in *fines, expulsion, and jail time.* **You must not attack anyone else's system without authorization!** Per the course ethics policy, you are required to respect the privacy and property rights of others at all times, *or else you will fail the course.* See the "Ethics, Law, and University Policies" section on the course website. Note that many companies have bug bounty programs and using your fuzzer to find bugs in such programs to claim such bug bounties is encouraged as it makes society safer.

Objective 1: My First Fuzzer

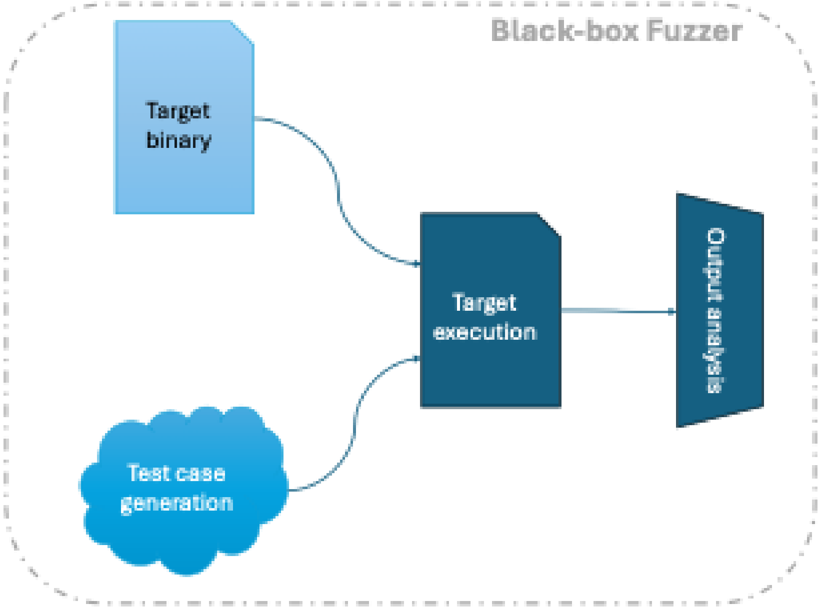

Figure 1: Basic black-box fuzzer.

Fuzzing is the least academic way to find bugs in programs: a fuzzer generates random test cases (i.e., inputs to some target program under test), executes them, and sees if the test case causes a crash during execution. Figure 1 shows a predecessor to fuzzing: black-box, randomized testing. In black-box testing, we assume that we know nothing about the internal operation of the target program during execution. Thus, the fuzzer can only generate test cases, execute the target with those test cases, and see what outputs the target produces; the ideal output being a crash, that is then sent to a human to triage.

In the first objective of this assignment, you will start the design of your fuzzer with a black-box fuzzer that randomly generates test cases. Since you can't assume anything about the target (because this is black-box testing) you will have to make sure that your fuzzer will generate all possible bytes (except ‘ and the null terminator). The target program that you will be testing takes a character string as input and crashes when the passed character string matches the secret value. Your fuzzer's goal is to find the secret value by causing the target program to crash. As you progress your way through the other assignment objectives, you will discover that black-box testing with randomly generated test cases is horribly ineffective.

Here are some hints to guide your development:

- Start with one-character strings and increase the length of the strings systematically.
- Use `rand()` to generate random numbers in C and `srand()` to seed the random number generator.
- You can use single quotes to pass bytes as input to a program via the command line. ’“When using single quotes, any characters enclosed within them are treated literally, which means that variable expansion and command substitution do not occur. This makes single quotes useful when you want to preserve the exact value of a variable or a command within a string.”
- You can pass a binary information to a program's standard in via `<` redirection using a binary file as the source.
- You can use a pipe (via `|`) to pass binary information between programs via standard in.
- For timing, you can use the C functions `time()` and `difftime()`, the `time` command, or some other tool to time your fuzzer. The only requirement is seconds granularity.
- If you worry about your fuzzer not working with the target binary, you can download a proof-of-life program (https://courses.cs.vt.edu/~cs4264/static/fuzzing/pol.c) and binary (mac: https://courses.cs.vt.edu/~cs4264/static/fuzzing/mac/pol and linux: https://courses.cs.vt.edu/~cs4264/static/fuzzing/linux/pol) to test your fuzzer. The proof-of-life program requires just a single byte of input and provides outputs that might be useful for debugging your fuzzer. Note that the target binary is more complex.

Note: The pre- and post-objective questions (throughout the assignment) are designed to get you to think about key fuzzing challenges and tradeoffs. Thus, your response will be graded on effort—not on correctness.

Pre-objective questions:

1. How do you generate test cases without needing to track what you've already generated?
2. Does it matter if you generate duplicate test cases? Roughly, how likely is it that your fuzzer will generate duplicate test cases?
3. How do you run another program from inside your program and supply it with arbitrary bytes?

Objective Steps:

1. Download the target binary for your system:
    (a) OSX: https://courses.cs.vt.edu/~cs4264/static/fuzzing/mac/target

    (b) Linux: https://courses.cs.vt.edu/~cs4264/static/fuzzing/linux/target
2. Run the target by hand to observe its behavior
    (a) ./target myInput123
    (b) The program outputs an error message with malformed input, nothing with correctly formatted input, and will crash when you find the secret input.
3. Build your black-box fuzzer using C by solving three challenges
    (a) Target execution: you must use system()
    (b) Test case generation: see the hint above
    (c) Output analysis: handle when the target crashes and when it doesn't
4. Use your fuzzer to find a crash-inducing test case
    (a) Note: this may take over an hour to discover the crash, depending on how fast your machine is.
    (b) Verify that the crash-inducing test case actually causes a crash outside of fuzzing by running it independently of the fuzzer
5. Update your fuzzer so that it reports the crash-inducing test case, the number of test cases it generated to find the crash, and the wall clock time it took to find the crash as your answer for Objective 1

Post-objective questions:
1. What is the fundamental weakness of black-box fuzzing?
2. How would you eliminate this weakness?
3. How would you improve your fuzzer?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet). Include your source file(s) along with your plain text file in the submitted zip file.
- Crash-inducing test case: ABC123
- Number of test cases: 1,234,567
- Wall clock time: 2700 seconds
- Pre question 1: A sentence or two.
- Pre question 2: A sentence or two.
- Pre question 3: A sentence or two.
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Post question 3: A sentence or two.
- Source code: include in zip bundle

Objective 2: Statistical Analysis

Fuzzing is an inherently random process due to the way that it generates test cases. When dealing with randomness in experiments, it is critical to perform many trials to capture a more complete distribution of behaviors to determine a truly representative outcome. Thus, in Objective 2, you will repeat the crash finding step of Objective 1 **3 times** to form a distribution of your fuzzer's performance. For each trial, record the number of test cases the fuzzer generated to find a crash-inducing test case and the wall clock time that the fuzzer took to uncover the crash. Report the average and standard deviation for both fuzzer performance metrics. These results form the baseline results that we will compare all future fuzzer incarnations against.

Pre-objective question:

1. Do you expect there to be a low or a high degree of variation between fuzzer runs and why?

Objective Steps:

1. Augment your fuzzer, write a shell script, or use Python such that it performs 3 fuzzing trials of the target.
2. Verify that the crash-inducing test case matches for every trial.
3. Calculate the mean and standard deviation for the number of test cases produced and wall clock time for each trial.

Post-objective questions:

1. Did you see a little or a lot of inter-run variation and why?
2. What is the impact of making claims about fuzzer performance and effectiveness using only a single run?
3. What tools or techniques would you use to assess if the number of experimental trials collected is sufficient to make comparative claims?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet).

- Crash-inducing test case: ABC123
- Average number of test cases: 1,234,567
- Std. Dev. number of test cases: 123,456
- Average wall clock time: 2700 seconds
- Std. Dev. wall clock time: 234 seconds
- Pre question 1: A sentence or two.
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Post question 3: A sentence or two.

Objective 3: Grammar-based Fuzzing

The limiting factor with the fuzzer you built in Objective 1 is that the fuzzer has no clue what types of input the target program takes besides it being a single array of bytes. In practice, reverse engineers looking to fuzz an opaque binary will have some information about the target program and the input format that the target expects (e.g., javascript to a web browser or a PDF file to Adobe Acrobat). Even in cases where such information is unknown, reverse engineers will disassemble or decompile the binary and perform manual analysis to get an idea of the target's input format. Spending time learning about the input the fuzzer expects is time well spent as it greatly increases the effectiveness of the fuzzer. This is because most real-world programs do a good job of dispatching test cases with malformed (i.e., syntactically incorrect) inputs. Thus, blindly throwing randomly generated bytes at a program only serves to test the existence of input format checking logic. Contrary to this, most bugs discovered by fuzzing involved unexpected, but well-formed inputs (e.g., a packet with zero length).

I will save you the effort of running odjdump and let you know the target's input format:

- Valid inputs are three characters long
- Valid characters come from the set $[0-9a-zA-Z]$
- Complete regex: $[0-9a-zA-Z]\{3\}$

If you are going to ever work in writing a language processor, translator, or compiler, you should know what Backus–Naur Form (BNF) is. BNF describes the syntax of programming languages or other formally-defined languages. BNF encodes a context-free grammar. BNF can be used to describe document formats, instruction sets, programming languages, and communication protocols.

Here is a BNF grammar that encodes this input format:

```
<input> ::= <char><char><char>
<char> ::= any lowercase character | any uppercase character | any decimal digit
```

Now that you know the grammar used to construct valid inputs to the target program, you will update the test case generation portion of your fuzzer to leverage that information to improve its effectiveness.

Pre-objective questions:

1. How do you expect grammar-based fuzzing to impact the number of test cases generated to discover the crash-inducing test case?
2. How do you expect grammar-based fuzzing to impact the wall clock time your fuzzer takes to discover the crash-inducing input?

Objective Steps:

1. Update your fuzzer's mutation engine such that it only generates syntactically valid program inputs given the provided grammar.
2. Perform **5 trials** of fuzzing the target program with the grammar-based test case generation engine.
3. Verify that the resulting crash-inducing test cases match what you found in Objective 1 and that all trials match each other.
4. Calculate the average and standard deviation of the number of test cases executed and the wall clock time.

Post-objective questions:

1. Explain why your results (compared to the results in Objective 2) make sense.
2. What are the tradeoffs in terms of performance and bug-finding ability for grammar-based fuzzing?
3. How would you improve your fuzzer's performance?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet). Include your source file(s) along with your plain text file in the submitted zip file.

- Crash-inducing test case: ABC123
- Average number of test cases: 1,234,567
- Average number of test cases relative to Objective 2: 0.54 times
- Std. Dev. In number of test cases: 123,456
- Std. dev. in number of test cases relative to Objective 2: 0.54 times
- Average wall clock time: 2700 seconds
- Average wall clock time relative to Objective 2: 0.54 times
- Std. Dev. in wall clock time: 234 seconds
- Std. Dev. in wall clock time relative to Objective 2: 0.54 times
- Pre question 1: A sentence or two.
- Pre question 2: A sentence or two.
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Post question 3: A sentence or two.
- Source code: include in zip bundle

Objective 4: My First Coverage-guided Fuzzer

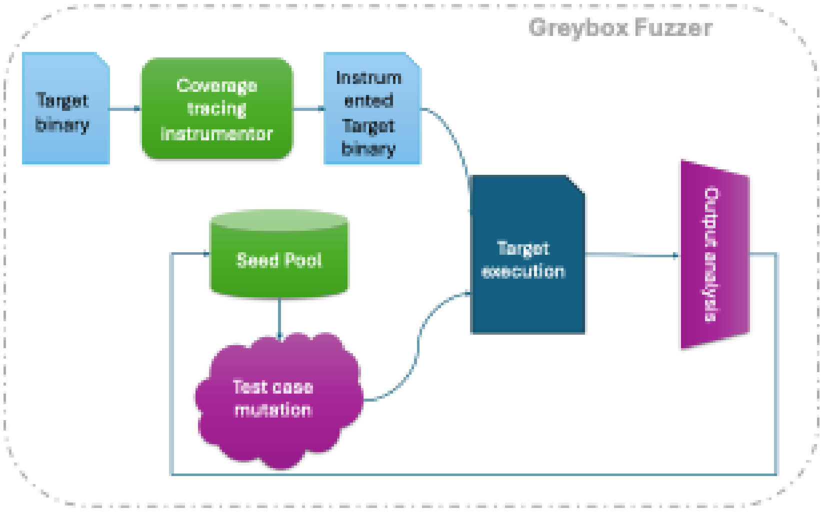

Figure 2: Basic coverage-guided fuzzer.

Even when fuzzers have access to a grammar that describes a target program's input format, modern programs are so complex that randomly generating a crash-inducing test case is incredibly unlikely. Your improvements in this objective will reveal that having a grammar alone does not yield an effective fuzzer—even with a relatively simple target program. The greatest advancement in fuzzing is the introduction of code coverage to guide the many decisions a fuzzer must make (instead of relying on pure randomness). At a high level, code coverage extends the information about test case execution beyond whether it produces a crash or not. The fuzzer uses this more detailed information to systematically explore a target program's behavior by gradually building new test cases upon past test cases; eventually leading to a crash as opposed to happening to magically derive a crash-inducing test case from thin air. The source of this extra information is code coverage.

By looking at the parts of a target's code that are covered or touched during execution, the fuzzer can determine if a test case is 'interesting' with respect to all previous test cases. This is done by seeing if the current test case executes some previously unexecuted region of the target's code or executes a previously executed region in a new way. The fuzzer keeps all interesting test cases (called seeds) around in a seed pool; the fuzzer discards all test cases that are both uninteresting and non-crashing. The fuzzer creates new test cases by mutating a seed selected from the seed pool (usually at random or using a power schedule) as opposed to generating one from scratch (as in Objective 3). By collecting and analyzing the set of interesting test cases, the fuzzer can make more targeted (but still randomly influenced) decisions during test case generation that allow the fuzzer to systematically work towards finding a crash-inducing test case.

Code coverage comes in many flavors with the most popular code coverage metrics in software fuzzing being basic block, edge, and path coverage. Basic block coverage (also referred to as statement or line coverage) represents specific instructions that contribute to the target's execution of a given test case. Note that a basic block is a set of sequential (in location) instructions terminated with a control flow instruction. Thus, a basic block is either executed entirely or not at all (because control never passes to the first instruction in the basic block). It is possible to break every program into a graph of connected basic blocks. Edge coverage is a superset of basic block coverage as it refers to the possible flows between a pair of basic blocks (due to navigating control flow instructions that direct control between basic blocks). For example, an if/else statement produces two edges: one that leads to the if basic block(s) and one that leads to the else basic block(s). Thus, edge coverage is a superset of basic block coverage, as every basic block is preceded by at least one edge, but possibly more. Lastly, path coverage represents an ordered traversal of edges through a program. For two test cases to have the exact same path coverage, they must execute the exact same instructions, in the exact same order as each other—only the data values differ. Thus, basic block coverage represents the coarsest grain metric and path coverage represents the finest grain metric. Research suggests that there is no one superior code coverage metric as too fine grain of a coverage metric tends to report effectively identical test cases as interesting, polluting the seed pool (which reduces mutation effectiveness) and too coarse grain of a coverage metric tends to force the fuzzer to make too big of jumps between a seed and a new, interesting test case (making a mutational fuzzer preform closer to a generational fuzzer).

In this objective, you will transform your fuzzer from a grammar-based, generational, black-box fuzzer to a grammar-based, mutational, coverage-guided, greybox fuzzer. See Figure 2 for a high-level depiction of a grammar-based, mutational, coverage-guided, greybox fuzzer. This type of fuzzer is the foundation for the most prevalent and successful academic and industrial fuzzers. Accomplishing this goal requires solving two technical challenges: (1) how to trace the code coverage during test case execution and (2) how to mutate seed test cases to form new test cases. Due to the time limit of the course, we will tell you the easiest path to code coverage tracing: compiler-based instrumentation using LLVMs existing compile-time options. You will use LLVM's SanitizerCoverage built-in code coverage instrumentation to trace a test case's edge coverage during execution. Once execution completes, you coverage will be dumped to a file. You will write code to process the coverage information in the file to determine if the most recently executed test case increased coverage with respect to all previous test cases. If it does, you will add that test case to the seed pool and continue the next fuzzing iteration by selecting a seed from the seed pool.

Once you've selected a seed, you must mutate that seed to create a new test case. Unfortunately, there is not a wealth of knowledge about the tradeoffs at play during mutation. The literature does tell us that having some seeds to mutate from is better than attempting to generate a test case from scratch (especially without a grammar), because the fuzzer wastes much of its time generating the first syntactically correct test case (most programs do a good job filtering out obviously syntactically incorrect inputs). The literature also highlights the power of using a dictionary of program-specific symbols for mutation. For example, a common way to build a dictionary is to run the target through the strings utility, which reports all constant strings in the target. The mutator can pull from this dictionary when creating a new test case. Due to time constraints, we will not explore the impact of having a dictionary for this assignment, but if you wanted to explore the impact of a dictionary on your own, I recommend you write a script that extracts all constants that the target uses for comparisons from the binary and constrict your grammar to those values. Outside of having seeds and using a dictionary or grammar to perform smarter mutations, we leave the mutation strategy up to you. If you are curious of a simple yet effective mutation strategy, we recommend a baseline mutation algorithm that first selects a byte of the seed to change, then to swaps in a random byte to replace the existing byte (leveraging the grammar provided by the last objective, of course).

Pre-objective questions:

1. What are viable mutation strategies?
2. What are the tradeoffs of mutating a lot of the seed versus mutating a little of the seed?
3. What are the possible ways to collect code coverage from a program and the pros and cons of each approach?
4. What do you expect the impact on number of test cases and run time to be compared to the previous objective and why?

Objective Steps:

1. Transform your fuzzer into a grammar-based, mutational, coverage-guided, greybox fuzzer

    (a) Add a seed pool that holds all previously seen coverage-increasing test cases
        i. Assume that you will only need to hold up to 100 seeds to lower the complexity of your implementation.
    (b) Add a seed selector that chooses a seed from the seed pool to mutate
        i. An ordered walk through the seed pool or purely random selection are both sufficient strategies.
    (c) Generate new test cases by mutating seeds (using the grammar from Objective 3 to ensure that you create only syntactically correct test cases.)
    (d) Instrument the target that it reports edge code coverage
        i. Use LLVM's SanitizerCoverage built-in code coverage instrumentation. Specifically, use `-fsanitize-coverage=trace-pc-guard`
        ii. Reference https://clang.llvm.org/docs/SanitizerCoverage.html to help you code your implementation.
    (e) Add logic to your execution analysis code to determine if the test case increased code coverage given all previously executed test cases.
2. Download the source code for a new target program at https://courses.cs.vt.edu/~cs4264/static/fuzzing/target.c, which you will need to compile the target with coverage tracing support. You will use this target for all remaining objectives.
3. Perform 10 trials of fuzzing the target program with your coverage-guided fuzzer.
    (a) Verify that the resulting crash-inducing test cases match across all trials.
4. Calculate the average and standard deviation of the number of test cases executed and the wall clock time.

Post-objective questions:

1. How do the results from this objective compare to those of the previous objective in terms of number of test cases and execution time and why do they make sense?
2. How could you change the fuzzer to improve performance or effectiveness?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet). Include your source file(s) along with your plain text file in the submitted zip file.

- Crash-inducing test case: ABC123
- Average number of test cases: 1,234,567
- Average number of test cases relative to Objective 3: 0.54 times
- Std. Dev. In number of test cases: 123,456
- Std. dev. in number of test cases relative to Objective 3: 0.54 times
- Average wall clock time: 2700 seconds
- Average wall clock time relative to Objective 3: 0.54 times
- Std. Dev. in wall clock time: 234 seconds
- Std. Dev. in wall clock time relative to Objective 3: 0.54 times
- Pre question 1: A sentence or two.
- Pre question 2: A sentence or two.
- Pre question 3: A sentence or two.
- Pre question 4: A sentence or two.

- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Source code: include in zip bundle

Objective 5: Extending Code Coverage to Loops

As mentioned in the previous objective, the most common code coverage metrics employed by commercial and academic fuzzers are basic block, edge, and path coverage. There is actually an enhancement to basic block and edge coverage metrics that attempt to bridge the gap between them and path coverage. This coverage enhancement is called hit counts. Instead of the binary notion of coverage (i.e., either covered or not) used by basic block and edge coverage, they can be extended to record how many times a basic block or edge was executed by a test case, i.e., its hit count. Recording hit counts is useful for assessing the progress of loops. If a test cases executes more iterations of a loop than all previous test cases, then this could be an indicator that we are getting closer to seeing new, potentially buggy, behavior. Consider a buffer overflow as an example.

Thus, in this objective, you will replace the edge coverage tracing in your fuzzer with edge hit count tracing. You will still use LLVM's SanitizerCoverage for this but will leverage the inline 8-bit counters instrumentation, as opposed to the trace-pc-guard instrumentation. This will give your fuzzer to accurately track up to 255 iterations of any loops in the target.

Pre-objective questions:

1. What do you expect the impact on number of test cases and run time to be compared to the previous objective and why?
2. How does the implementation of trace-pc-guard and inline 8-bit counters compare? Which do you expect to be lower overhead and why?
    - **Hint:** use objdump to examine the disassembly of each instrumented target.

Objective Steps:

1. Replace edge coverage tracing with edge hit count coverage tracing
    (a) Instrument the target that it reports edge hit count code coverage
        i. Use LLVM's SanitizerCoverage built-in code coverage instrumentation.
        ii. Specifically, use `-fsanitize-coverage=inline-8bit-counters`
        iii. Reference https://clang.llvm.org/docs/SanitizerCoverage.html to help you code your implementation.
    (b) Update the code coverage analysis logic to determine if the test case increased the hit count on any edge given all previously executed test cases.
2. Perform 10 trials of fuzzing the target program with your coverage-guided fuzzer.
    (a) Verify that the resulting crash-inducing test cases match what you found in Objective 4 and that all trials match each other.
3. Calculate the average and standard deviation of the number of test cases executed and the wall clock time.

Post-objective questions:

1. How do the results from this objective compare to those of the previous objective in terms of number of test cases and execution time and why do they make sense?
2. How could you change the fuzzer to improve performance or effectiveness?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet). Include your source file(s) along with your plain text file in the submitted zip file.

- Crash-inducing test case: ABC123
- Average number of test cases: 1,234,567
- Average number of test cases relative to Objective 4: 0.54 times
- Std. Dev. In number of test cases: 123,456
- Std. dev. in number of test cases relative to Objective 4: 0.54 times
- Average wall clock time: 2700 seconds
- Average wall clock time relative to Objective 4: 0.54 times
- Std. Dev. in wall clock time: 234 seconds
- Std. Dev. in wall clock time relative to Objective 4: 0.54 times
- Pre question 1: A sentence or two.
- Pre question 2: A sentence or two.
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Source code: include in zip bundle

Objective 6: Better Process Management 1: Fork/Exec

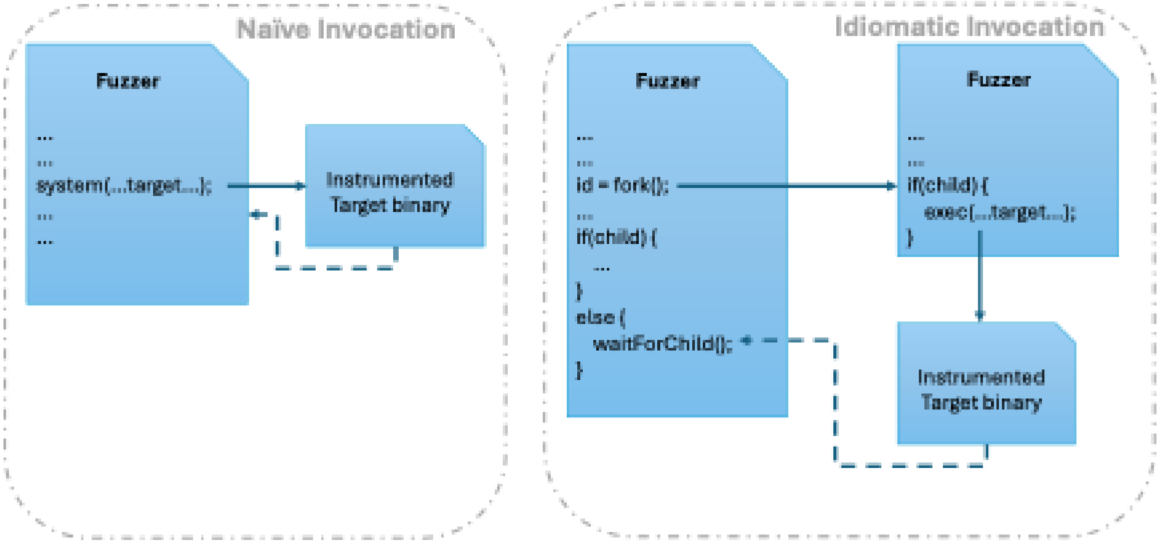

Figure 3: Fresh process creation options for target process management.

After Objective 5, you now have a fully-functioning coverage-guided, greybox fuzzer that will help you automatically find bugs in arbitrary programs. There are two ways that your fuzzer can improve its performance at this point: (1) better process management and (2) lower-overhead code coverage tracing. This objective focuses on increasing fuzzer performance by improving process management. Given that fuzzing revolves around executing a massive amount of test cases, the quicker it can execute them, the quicker it will find bugs. Where the fuzzer currently stands, for each test case, the fuzzer creates a new process and initializes it with the target program, executes the test case, and reaps the process, freeing its resources. It turns out that for the short running programs commonly fuzzed, the time taken to create, initialize, and reap a process is the same order of magnitude as executing the test case. This process management work is also invariant of the specific test case provided to the program. At the same time, process management is important as it is essential for correctness that each test case executes with the same starting program state (which starting with a fresh process is an easy path to).

Thus, to dramatically increase the speed of your fuzzer, you will explore mechanisms to eliminate duplicated process management. The first step in that direction is to create target processes in a more idiomatic way (at least on UNIX systems), as shown in Figure 3. When you run a program from a shell on a UNIX-based operating system, the shell creates a clone of itself using the fork() system call, then the clone process replaces the shell program with the code and data of the callee by using one of the exec() series of system calls. In this objective, you will replace you invocation of the target program using system() with a call to fork, followed by a call to exec.

Pre-objective question:

1. What do you expect the impact on number of test cases and run time to be compared to the previous objective and why?

Objective Steps:

1. Transform your generational coverage-guided, mutational, greybox fuzzer into one that use fork/exec
    (a) Replace `system()` with `fork()/exec()`
2. Perform 10 trials of fuzzing the target program with your coverage-guided fuzzer.
    (a) Verify that the resulting crash-inducing test cases match what you found in Objective 5 and that all trials match each other.
3. Calculate the average and standard deviation of the number of test cases executed and the wall clock time.

Post-objective questions:

1. How do the results from this objective compare to those of the previous objective in terms of number of test cases and execution time and why do they make sense?
2. How could you change the fuzzer's process management to improve performance?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet). Include your source file(s) along with your plain text file in the submitted zip file.

- Crash-inducing test case: ABC123
- Average number of test cases: 1,234,567
- Average number of test cases relative to Objective 5: 0.54 times
- Std. Dev. In number of test cases: 123,456
- Std. dev. in number of test cases relative to Objective 5: 0.54 times
- Average wall clock time: 2700 seconds
- Average wall clock time relative to Objective 5: 0.54 times
- Std. Dev. in wall clock time: 234 seconds
- Std. Dev. in wall clock time relative to Objective 5: 0.54 times
- Pre question 1: A sentence or two.
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Source code: include in zip bundle

[Extra Credit: 15 pts] Objective 7: Better Process Management 2: Forkserver

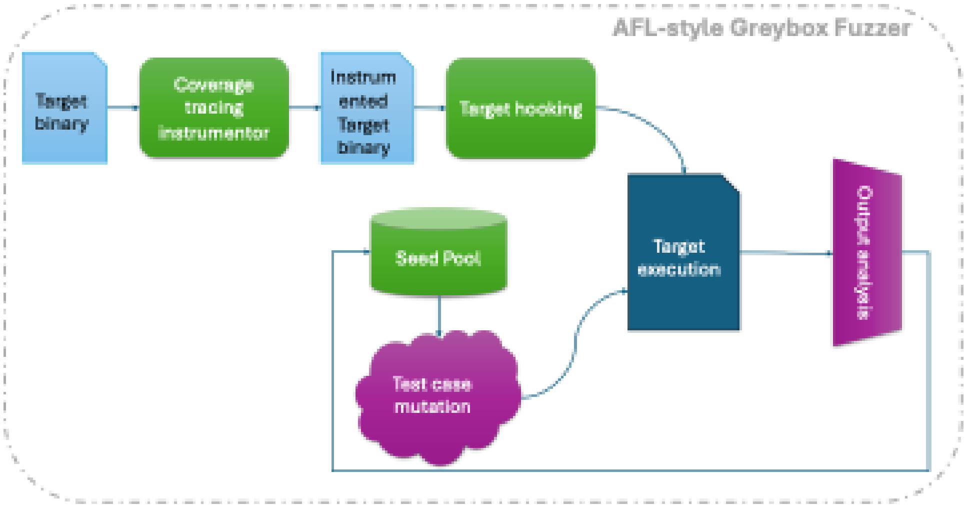

Figure 4: AFL-style greybox fuzzer that relies on hooking the target.

To deal with the high overhead of target process initialization, AFL-based Linux fuzzers employ a forkserver. Forkserver-based fresh process creation represents a significant improvement over the more simple `fork`/`exec`-based process creation in the previous objective. The forkserver reduces process management overhead by reducing some process initialization costs by duplicating the target process for each test case, as opposed to duplicating the fuzzer process and then replacing it with the target program. Cloning a process is low overhead due to the operating system using a technique called copy-on-write. Copy-on-write only actually

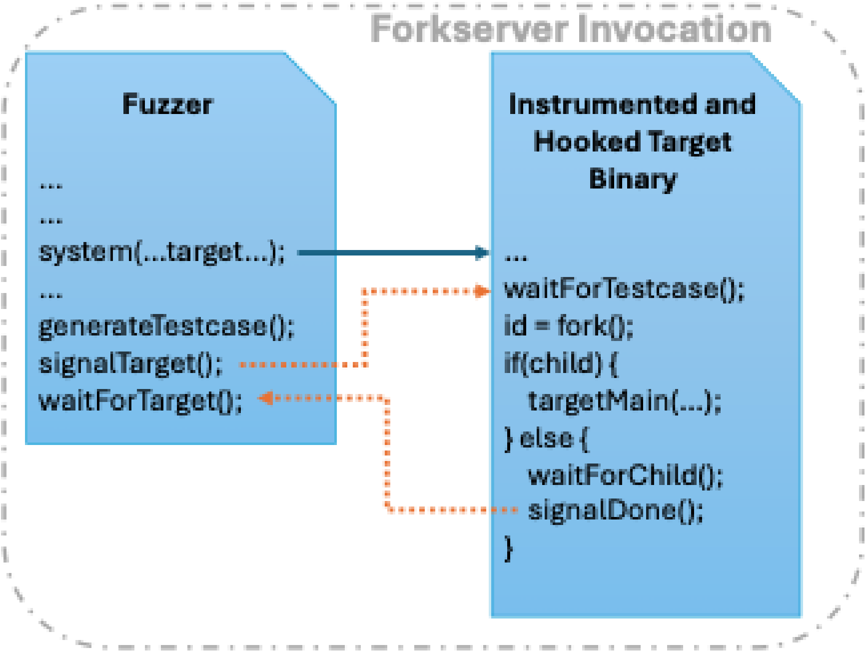

Figure 5: Fresh process creation via target-embedded forkserver.

creates duplicate memory pages (i.e., creates an independent copy of a memory page for a child, from a parent process) when a child or parent process writes to a cloned page. Put more simply, as long as the parent and child pages continue to be identical, they will share, but once they diverge, the operating system will pause execution to create a private copy for the child. The gambit at play is that most of the target's pages remain unchanged due to test case execution.

By cloning the parent process and leveraging copy-on-write, the forkserver is more fine-grain than than creating a fresh process as it ensures that the target process initialization steps are done—only once—and provides a transparent, page-level rollback of test-case-induced state changes via copy-on-write. Even though a forkserver does away with process loading costs, it still suffers from page-level duplication/management costs, process tear-down costs, as well as the kernel-level cost of process duplication. You will observe in future objectives that this technique is still relatively coarse-grain as it works at the page level and also relies on creating many processes.

A forkserver works by including some fuzzer-related code in the target program (called hooking). The fuzzer initially forks and loads the, now hooked, target binary into memory (just once though). The target pauses execution at the beginning of the inserted `main` function. The forkserver then waits for a new test case from the fuzzer. When it is received, the forkserver forks itself (i.e., creates a copy of the target's process at that point in execution) and starts executing the target's original `main` using the test case provided by the fuzzer process. The fuzzer code inserted into the target during hooking monitors the child's execution, reporting back the results to the fuzzer process. Because the fuzzer now has parts in the target and in its own process, Inter-Process Communication (IPC) mechanisms are used to coordinate and communicate across processes. In this objective, you will transform your fuzzer such that it uses a forkserver.

Pre-objective question:

1. What tradeoffs does a forkserver make?
2. What do you expect the effect of moving to a forkserver to be in terms of average test case run time and the average number of test cases executed until a crash is found and why?

Objective Steps:

1. Transform your generational coverage-guided, mutational, greybox fuzzer into one that uses a forkserver
   (a) Hook the target
      i. Interpose on the target's main
         A. Hint: an easy way to do this is via a preprocessor directive that creates macro the renames the target's main `targetMain()`
         B. Hint: a slightly more complex way to do this is to leverage LD_PRELOAD
      ii. Use fork()/exec() in your target hook code that spawns a child of the target and has the child execute the test case generated by the fuzzer process.
      iii. The parent process waits for child completion and checks for crashes.
      iv. The target hook code communicates test case execution status to the fuzzer process.
      v. The target hook code then waits for a new test case from the fuzzer process.
      vi. Hint: pipes and named pipes (aka FIFOs) are a great Inter-Process Communication (IPC) mechanism to solve the synchronization and communication challenges of this objective.
   (b) Modify the fuzzer.
      i. Invoke the hooked target from the fuzzer.
         A. You can use either `system()` or `fork()/exec()` for this.
      ii. Add matching IPC to synchronize and communicate with the target hook code in the target.
      iii. Add code to detect when the target crashes.
2. Use your fuzzer to find a crash-inducing test case for the target.
3. Perform 10 trials of fuzzing the target program with your coverage-guided fuzzer.
   (a) Verify that the resulting crash-inducing test cases match each other across trials.
4. Calculate the average and standard deviation of the number of test cases executed and the wall clock time.

Post-objective questions:

1. Why did you choose the IPC mechanism(s) that you did? What are the tradeoffs at play?
2. Why is it unimportant whether you use `system()` or `fork()/exec()` to start the hooked target?
3. Explain why the test case throughput results make sense.

4. What can be done to increase fuzzer performance in terms of reducing process management overhead?

What to submit: submit a plain text file containing your answers to the objective questions formatted as shown below (without the preceding bullet). Include your source file(s) along with your plain text file in the submitted zip file.

- Crash-inducing test case: ABC123
- Average number of test cases: 1,234,567
- Std. Dev. In number of test cases: 123,456
- Average wall clock time: 2700 seconds
- Test case throughput (i.e., test cases per second): 1234
- Test case throughput relative to Objective 6: 1.54 times
- Pre question 1: A sentence or two.
- Pre question 2: A sentence or two.
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Post question 3: A sentence or two.
- Post question 4: A sentence or two.
- Source code: include in zip bundle

Submission Checklist

Upload to Canvas a gzipped tar file named `project5.`*`pid1.pid2`*`.tar.gz` that contains only the files listed below. **These will be autograded, so make sure you have the proper filenames, formats, and behaviors**. Failure to work with the autograder—for any reason—will result in a 5% deduction from the maximum possible points. You can generate the tarball at the shell using this command:

```
tar -zcf project5.pid1.pid2.tar.gz obj[123456].txt obj[123456].c edgeTrace.c hitTrace.c
```

The tarball should contain only the files below:

```
obj1.txt
obj1.c
obj2.txt
obj3.txt
obj3.c
obj4.txt
obj4.c
edgeTrace.c
obj5.txt
obj5.c
hitTrace.c
obj6.txt
obj6.c
```

`obj7_target.c` [Optional extra credit.]
`obj7_fuzzer.c` [Optional extra credit.]

Your files can make use of standard C libraries and the provided on the most recent MacOS or Ubuntu Linux operating systems but they must be otherwise self-contained. Do not include any generated files with your submission. Be sure to test that your solutions work correctly in described environment—you don't have to test across platforms.