

Project 3: Fuzzing

This project is due on **October 30, 2026 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 final exam will cover project material, so you and your partner should collaborate on each part.

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. You may use AI for assistance provided you include all of the services and prompts that you used at the end of your writeup.

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 introduces you to automated vulnerability discovery in software through coverage-guided fuzz testing, a.k.a. fuzzing. You will start by implementing a simple baseline fuzzer to help you understand the fundamentals of fuzzer operation. Then you will progressively make enhancements to your fuzzer targeting performance and effectiveness limitations. By the end of this project, you will have a deep understanding of state-of-the-art fuzzer optimizations used in the world's fastest and most effective fuzzers.

Objectives

- Explain the fundamental operation and workflow of coverage-guided fuzzers.
- Analyze the types of bugs and vulnerabilities that fuzzers discover effectively—and where they tend to fall short.

- Identify key sources of performance overhead and propose design strategies to improve fuzzing efficiency.
- Gain familiarity with the state-of-research in high-performance fuzzing.

Read this First

This project asks you to find bugs in the software targets that we provide. Since you are developing a very real fuzzer, you may also test other software for learning or debugging purposes—but under no circumstances may you use discovered vulnerabilities to attack systems you do not own or for which you do not have explicit written authorization. Unauthorized testing or exploitation can violate criminal and civil law and University policy and may result in disciplinary action (including failing the course or expulsion), civil liability, and criminal prosecution. Bug-bounty participation is encouraged as a learning opportunity and as a way to improve security in the real world—but only when done within the bounty program’s rules and with explicit permission where required. See the “Ethics, Law, and University Policies” section on the course website for full details. If you’re unsure, ask.

Preliminaries

To simplify this assignment, everyone will use the Computer Science Department’s rlogin computing infrastructure. Rlogin is a cluster of shared, 64-bit Linux servers that provides a consistent environment for compiling and running your fuzzer and building the target. Rlogin consists of 32 compute nodes, each with two 16-core Intel Xeon processors and 384 GB of memory. These resources are shared with other students; they are not reserved for your experiments. Your home directory is shared across nodes, but running processes belong to the particular node on which you start them.

Start with the department’s rlogin access guide, which explains account setup, SSH access, and connecting to a specific node. You can create a CS account at <https://admin.cs.vt.edu/create/> and reset its password at <https://admin.cs.vt.edu/reset/>. With a CS account, connect from a terminal using `ssh yourusername@rlogin.cs.vt.edu`, replacing `yourusername` with your CS username; this login uses your CS password. For account or connection problems, contact techstaff@cs.vt.edu.

Fuzzing results depend on randomness: one lucky run can find a bug quickly, while another run of the same fuzzer may take much longer. A common practice in fuzzing evaluations is to run multiple independent 24-hour trials with different random-influenced test cases, giving each configuration the same time budget. Longer runs help reveal behavior beyond initial startup, and repeated runs help distinguish consistent improvements from chance. Follow each objective’s instructions for the measurements required in this assignment, and plan ahead for experiments that may take many hours.

Your evaluation must keep running when your laptop sleeps, your network connection drops, or you log out of SSH. Choose one of the following mechanisms and start it *on rlogin* before launching a long experiment. In the examples, replace `./fuzzer` with your executable and any required arguments, and use a different log filename for each trial.

- **nohup: a simple unattended run.** This command makes a program ignore the hangup signal normally associated with losing its terminal. For example:

```
nohup ./fuzzer > trial1.log 2>&1 < /dev/null &
```

The final `&` runs the command in the background; the redirections save standard output and errors to `trial1.log` and disconnect standard input from the terminal. This is useful for a batch experiment that needs no keyboard input. You can inspect its output with `tail -f trial1.log`, but `nohup` does not provide an interactive session to reattach to.

- **screen: a persistent terminal session.** Run `screen -S fuzzing`, then launch your fuzzer inside the new shell. Press `Ctrl-a`, release the keys, and press `d` to detach while the program continues running. After reconnecting to the same node, use `screen -r fuzzing` to return, or `screen -ls` to list your sessions. This is useful when you want to return to the same interactive terminal to inspect progress or stop the experiment.
- **tmux: multiple persistent terminals.** Run `tmux new -s fuzzing`, then launch your fuzzer inside it. Detach with `Ctrl-b`, followed by `d`; reconnect on the same node with `tmux attach -t fuzzing`. Use `tmux ls` to list sessions. Like `screen`, it preserves an interactive terminal; its windows and split panes are convenient for keeping a shell and a log viewer alongside your experiment.

Reconnect to the same node. Logging in through `rlogin.cs.vt.edu` may send you to a different server each time. Run `hostname` before starting your experiment and record the result. To return, connect with `ssh yourusername@portal.cs.vt.edu` and select that node from the menu, as described in the access guide. Shared files will still be visible on other nodes, but your running fuzzer and terminal session will remain on the original node. Before starting Objective 1, practice launching a long-running job, disconnecting, and reconnecting to verify your setup.

Follow the `rlogin` usage policies: keep the number of concurrent experiments modest, monitor disk usage, and stop unneeded jobs and sessions when finished.

Warning: Even if you develop on your own machine, your fuzzer must run correctly on `Rlogin` servers, as we will use them for grading.

Run trials concurrently. You can and should run multiple independent trials at the same time, on the same `rlogin` node or on different nodes. Give each trial its own working directory, log, and output files, so that trials do not overwrite one another's coverage data or results. Start more trials than the minimum required when resources permit, because some runs may take much longer than expected or fail unexpectedly. Use only complete, independent trials for your reported measurements.

Objective 1: My First Fuzzer

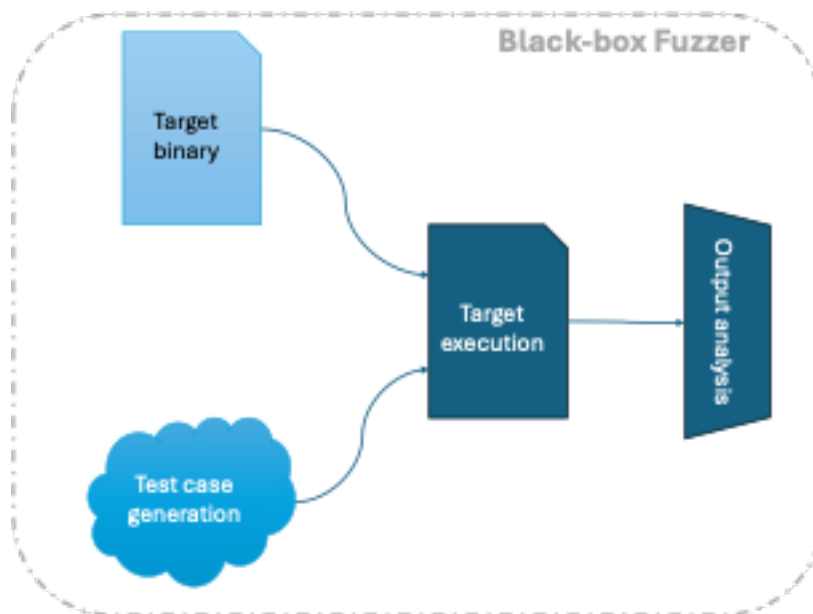


Figure 1: Basic black-box fuzzer.

Fuzzing is the least academic way to find bugs in programs: a fuzzer generates random-influenced test cases (i.e., inputs to some target Software Under Test (SUT)), executes them, and sees if the test case causes a crash during execution. Figure 1 shows a predecessor to fuzzing: black-box, completely-randomized testing. Black-box testing assumes no knowledge of the internal operation of the target program during execution. Thus, the testing system is limited to generating test cases randomly, executing them on the SUT, with the only observable information being whether the SUT crashed or not. Crashing test cases are sent to a human to triage. Alternatively, non-crashing test cases are ignored.

For the first objective, you will start the design of your fuzzer with a black-box fuzzer that randomly generates test cases. Since you cannot assume anything about the target program (because this is black-box testing) your fuzzer must generate all possible bytes (except ‘ and the null terminator characters, for simplicity). The SUT that you will be testing takes a byte string as input and crashes when the passed bytes match the secret value; the SUT will crash when you pass in the secret value. As you progress your way through the other assignment objectives, you will discover that black-box testing with completely-random test cases is horribly ineffective.

Here are some hints to guide your development:

- Start with one-byte strings and increase the input length systematically.
- Use `rand()` to generate random numbers in C and `srand()` to seed the random number generator.
- Supplying arbitrary bytes to programs: common methods:
 - **stdin from a file:** `echo $'\x41\x42\x00' > case.bin; ./target < case.bin`

- **pipe:** `echo $'\x41\x42' | ./target`
- **argv with bash-style escape:** `./target $'AB\x00'`.
- **Note:** the `$` is not needed when passing raw bytes.
- 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 <https://courses.cs.vt.edu/~cs4264/static/fuzzing/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 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 the SUT from inside your fuzzer and supply it with arbitrary bytes?

Objective Steps:

1. Download the SUT binary: <https://courses.cs.vt.edu/~cs4264/static/fuzzing/target>
2. Execute the SUT by hand to observe its behavior
 - (a) `./target myInput123`
 - (b) The SUT outputs:
 - an error message with malformed input
 - nothing with correctly formatted input
 - crashes when you find the secret input value
3. Build your black-box fuzzer using C by solving three core fuzzer challenges
 - (a) **SUT execution:** you must use `system()` to execute the SUT from within your fuzzer
 - (b) **Test case generation:** see the requirements and hint above
 - (c) **Execution 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 11 hours to discover the crash, depending on how much load is on rlogin.
 - (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 before finding the crash, and the wall clock time it took to find the crash-inducing test case.

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 project submission archive.

- 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 submission archive

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 (even beyond fuzzing), it is critical to perform multiple trials to capture a distribution of behaviors to determine a truly representative view of fuzzer performance and effectiveness. 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.

Diving Deeper: In an actual scientific experiment you must use statistics to guide when you have performed a sufficient number of trials to get a good idea of the real distribution of fuzzer behavior. I recommend reading the research paper “Evaluating Fuzz Testing” from Klees et al. from the University of Maryland in 2018. The paper provides rules of thumb for evaluating fuzzer performance and effectiveness as well as statistical analysis tools to make scientific claims comparing fuzzers. The most important statistical tools that the paper suggests fuzzing evaluations should employ are the Mann Whitney U-test to determine whether the outcomes of the trials in fuzzer A’s data sample are more likely to be larger than outcomes in fuzzer B’s and the Vargha and Delaney’s A_{12} test to determine the magnitude

of difference between fuzzers A and B. To see how this paper impacted 150 fuzzing papers published after it, read “SoK: Prudent Evaluation Practices for Fuzzing” from Schloegel et al. in 2024. The paper reveals the common practices used in the evaluations of fuzzing research papers published at the top venues across the Computer Security and Software Engineering fields.

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 across the trials.

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: informing the fuzzer about input structure

The fuzzer you built in Objective 1 treats inputs as opaque byte arrays. In practice, however, fuzzing is far more effective when the fuzzer understands (or approximates) the SUT’s input

format. Real-world software usually expects structured, syntactically correct inputs (for example, JavaScript to a browser, a PDF file to a viewer, or a well-formed network packet). Thus, programs typically filter out malformed inputs quickly, so purely random inputs often exercise only the format-checking paths.

Many bugs are triggered not by grossly malformed inputs but by unexpected yet well-formed inputs (for example, a field with length zero, unusual nesting depth, or a rarely used option). Reverse engineers sometimes recover input structure by reading source, examining documentation, or by disassembling/decompiling a binary to find parsing routines; other times, format structure is inferred from sample files or protocol traces. Investing time to learn or model the expected input format is usually worth the effort, because it lets the fuzzer reach deeper, semantically meaningful code paths where real bugs hide.

In this objective you will modify your fuzzer to exploit knowledge about the target SUT's input format. Possible approaches include grammar-based generation, template-based generation, or structure-preserving mutation. In this objective, you will leverage grammar-based generation.

I will save you the effort of running `objdump` (to reverse engineer the SUT) and tell you the SUT's input format:

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

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 the SUT's 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 for the SUT, update the test case generation portion of your fuzzer to leverage that information.

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 project submission archive.

- 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 submission archive

Objective 4: Coverage-guided Fuzzing: informing the fuzzer about details of SUT execution

Even with a grammar, randomly producing a crash-inducing input for realistic programs is extremely unlikely for SUTs of even moderate complexity. The major advance that makes modern fuzzers effective is the addition of SUT execution feedback—typically in the form of code coverage—to guide fuzzing decisions. Coverage gives the fuzzer information beyond “crashed or didn’t crash”: it tells the fuzzer what parts of the program were exercised. This more detailed execution feedback allows the fuzzer to identify interesting inputs that explore new SUT behavior.

By looking at the parts of a SUT’s code covered during execution, the fuzzer can determine if a test case is “interesting” with respect to all previously executed test cases. An

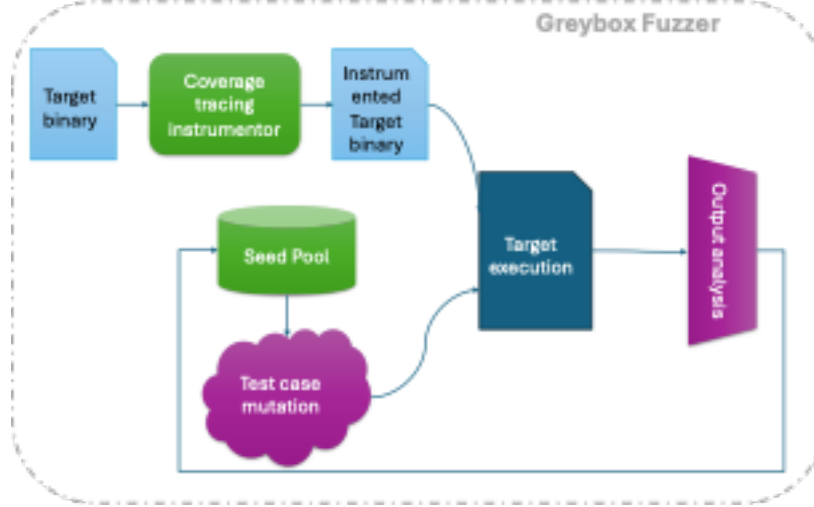


Figure 2: Basic coverage-guided fuzzer.

input is interesting if it increases coverage (e.g., executes a previously unseen edge or basic block). A coverage-guided fuzzer keeps a pool of interesting test cases (called seeds), while those that are uninteresting or result in timeouts are discarded. The fuzzer mutates seeds rather than generating inputs from scratch, focusing effort on regions of the input space that lead to new execution behavior. This guided, mutation-based approach is the foundation of the most successful academic and industrial fuzzers as it allows for systematic SUT behavior exploration.

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 SUT’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. 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 flow 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 the SUT. 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 perform 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. Code coverage tracing: due to the time limit of the course, we will tell you the easiest path to code coverage tracing: compiler-based instrumentation using LLVM's 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, code 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.
2. Mutation-based test case creation: 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 SUT through the `strings` utility, which reports all constant strings in the SUT. 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 SUT uses for comparisons from the binary and constrain 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 about a simple yet effective mutation strategy, we recommend a baseline mutation algorithm that first selects a byte of the seed to change, then 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 SUT such 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 SUT at <https://courses.cs.vt.edu/~cs4264/static/fuzzing/targetCov.c>, which you will need to compile the SUT with coverage tracing support.
3. Perform 10 trials of fuzzing the SUT 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 project submission archive.

- 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 submission archive

Objective 5: Extending Code Coverage to Loops

Basic block and edge coverage record whether a program region was executed; hit counts extend this binary notion of code coverage by recording how many times a block or edge executed during a run. Hit counts are particularly useful for loop-heavy code: a test case that executes more loop iterations than any previous test may be getting closer to triggering loop-related bugs (e.g., off-by-one overflows, size checks, or loop-carried memory corruption).

In this objective you will replace the edge-only coverage tracing used in Objective 4 with edge hit count tracing and use a new target with an input-dependent loop. Use the same three-character alphanumeric input grammar as before. The new target retains the crash-inducing input from Objective 4, but its loop runs a different number of times depending on the input. To keep the assignment practical, use LLVM’s SanitizerCoverage inline 8-bit counters instrumentation (compiler option) to record per-edge 8-bit counters. These counters let your fuzzer detect when a test causes more iterations of a loop (or more executions of a given edge) than all previous tests, up to a per-edge maximum of 255 counts. Use this information to decide whether a test case is “interesting” (i.e., it increases the known observed maximum hit count for at least one edge) and therefore should be added to the seed pool.

Pre-objective questions:

1. How might the new target and hit-count feedback each affect the number of test cases and run time compared to Objective 4?
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. Download `targetCovLoop.c` from <https://courses.cs.vt.edu/~cs4264/static/fuzzing/targetCovLoop.c> and compile it as the SUT for this objective. It replaces `targetCov.c` from Objective 4 and will also be used in Objective 6.
2. Replace edge coverage tracing with edge hit count coverage tracing
 - (a) Instrument the SUT such 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.

3. Check that two valid inputs can traverse the loop a different number of times and that your hit-count output distinguishes them. Record the inputs and their observed loop-edge hit counts.
4. Perform 10 trials of fuzzing the SUT 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.
5. 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 compare with Objective 4 in terms of number of test cases and execution time? Since both the target and coverage feedback changed, what can and cannot be attributed to hit counts?
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 project submission archive.

- Crash-inducing test case: ABC123
- Loop hit-count check: two inputs and their observed loop-edge hit counts
- 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 submission archive

Objective 6: Better Process Management 1: Fork/Exec

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

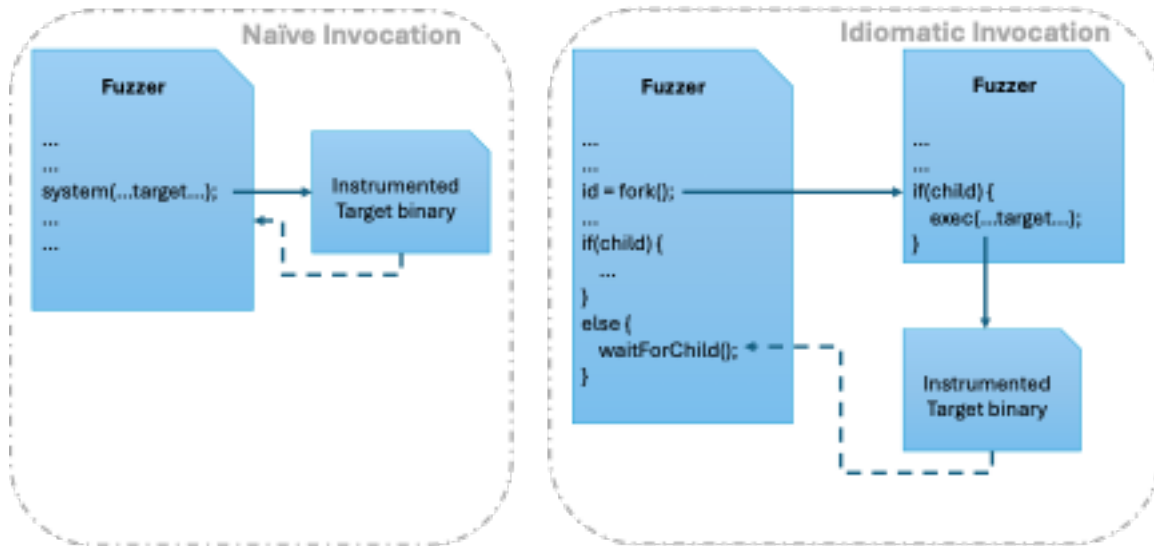


Figure 3: Fresh process creation options for target process management.

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 your 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. Continue using `targetCovLoop.c` from Objective 5 with edge hit count tracing.
2. Transform your generational coverage-guided, mutational, greybox fuzzer into one that uses `fork/exec`
 - (a) Replace `system()` with `fork()/exec()`
3. 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.
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'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 project submission archive.

- 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 submission archive

Objective 7: Handling Abnormal Target Termination

Up to this point, your fuzzer stops when it discovers a crash-inducing test case. Real programs may contain many bugs, and different test cases may trigger the same bug. To discover additional bugs, your fuzzer must save its findings and continue testing until the fuzzing expedition ends.

An input may also cause the SUT to run unusually slowly or enter an infinite loop. Waiting indefinitely for that execution to complete brings the entire expedition to a halt. A per-test-case timeout bounds how long the fuzzer waits before terminating the SUT and moving on. Exceeding a timeout does not prove that a program is stuck in an infinite loop: a slow execution or heavy load on a shared server may also cause a timeout.

In this objective, you will extend the fork/exec fuzzer from Objective 6 to handle multiple crashes and timeouts during a 24-hour experiment. Adding this functionality to your fuzzer brings it one step closer to working with real-world software.

Pre-objective questions:

1. Should crashing test cases be added to the seed pool? Why or why not?
2. Should timeout-inducing test cases be added to the seed pool? Why or why not?
3. When should you stop fuzzing and why?

Objective Steps:

1. Download the new SUT, which contains multiple crash conditions and an infinite loop:
<https://courses.cs.vt.edu/~cs4264/static/fuzzing/targetAbnormal.c>
 - (a) This replaces `targetCovLoop.c`, used in Objectives 5 and 6. It still takes a three-character alphanumeric string as a command-line argument, so retain your existing grammar and mutation strategy.
 - (b) Compile the new target with the edge hit count instrumentation from Objective 5 and continue using `fork/exec` to execute it.
2. Modify your fuzzer to recognize and report multiple crashes without stopping the fuzzer.
 - (a) Inspect the status returned by `waitpid()`. Use `WIFEXITED()` to check for normal termination, then `WEXITSTATUS()` to read the exit code. Use `WIFSIGNALED()` to check for signal termination, then `WTERMSIG()` to read the signal.
 - (b) For this target, count a signal-terminated execution as a crash unless your fuzzer terminated it because of a timeout. A nonzero exit code alone is not a crash. Detect and report failures to create or execute the target separately from target crashes.
 - (c) Preserve the exact crash-inducing input and its terminating signal, by writing them to the screen or to disk, then continue fuzzing.
3. Add a positive, user-provided command-line timeout, specified in milliseconds, for each SUT execution.
 - (a) Measure elapsed wall clock time using a monotonic clock. For example, use `clock_gettime()` with `CLOCK_MONOTONIC`. Include time spent waiting for the target; the fuzzer's CPU time alone is insufficient.
 - (b) Ensure that waiting for the child cannot block indefinitely. One approach is to poll `waitpid()` with `WNOHANG` and check the deadline between polls. Avoid busy-waiting, but recognize that long sleeps between polls lower throughput.
 - (c) If the target is still running when its time limit expires, terminate that child with `kill()` and `SIGKILL`, then reap it with `waitpid()`. Record that your fuzzer initiated the termination so that the resulting signal is classified as a timeout rather than a crash.
 - (d) Save the timeout-inducing input and continue fuzzing. Reap every child, whether it exits normally, crashes, or times out, so that long experiments do not accumulate zombie processes.
4. Keep abnormal findings separate from the seed pool and handle missing coverage output correctly.
 - (a) For this objective, add only normally terminating, coverage-increasing test cases to the seed pool. Save crashes and timeouts for reporting, but do not add them to the pool or use their coverage to update the global hit count maxima.

- (b) A crash or forced termination may bypass the code that writes coverage at normal exit. Remove the previous coverage file before each execution, and process only valid coverage from the current, normally terminating execution. Handle missing or incomplete output without crashing the fuzzer or reusing stale coverage.
 - (c) Track total crash and timeout events and distinct inputs in each category. Keep one record per distinct input and outcome, with its occurrence count and elapsed time of first discovery. Exact input comparison is sufficient; grouping different inputs by their underlying bug is not required.
5. Perform one 24-hour fuzzing trial on the new target.
- (a) First perform short checks that your fuzzer continues after a crash, terminates an execution that exceeds its timeout, and can execute another test case afterward. Verify that a nonzero normal exit is not reported as a crash.
 - (b) Choose a per-test-case timeout that allows ordinary target executions to finish on rlogin, and explain your choice. Keep this timeout fixed throughout the trial.
 - (c) Use the unattended-run setup from the Preliminaries. Start with an empty seed pool and coverage history, and record the random number generator seed and rlogin node.
 - (d) Stop launching new test cases after 86,400 seconds, even if you have found no new abnormal inputs recently. Allow any remaining execution to finish or reach its per-test-case timeout, reap it, and report the actual total elapsed time.
 - (e) Report the total number of executions, including crashes and timeouts. Calculate throughput as this count divided by actual elapsed seconds. Report all abnormal findings using the records described above.
 - (f) After the trial, replay each distinct abnormal input independently of the fuzzing loop. For timeouts, replay at the original limit and at a larger, finite limit; report whether they finish. Exclude replays from trial measurements.

Post-objective questions:

1. How do your total crash and timeout counts compare with your distinct input counts? Why might different crash-inducing inputs trigger the same bug?
2. Why did you choose your timeout value? What are the tradeoffs of a shorter or longer timeout on a shared rlogin node? What did replaying inputs with a larger timeout show?
3. How did you verify that your fuzzer continues correctly after crashes and timeouts, reaps target processes, and does not reuse stale coverage?
4. Does finding no new abnormal inputs near the end of the trial imply that you have found all bugs? Why is a single 24-hour trial insufficient to make a strong comparison of two fuzzers' effectiveness?

What to submit: submit `obj7.txt` with your measurements and answers formatted as shown below (without the preceding bullet). Replace example values with your results, using zero or “none” where appropriate. Repeat the abnormal-input lines for every distinct finding. Include `obj7.c`, any updated coverage tracing source, and instructions for compiling and running your fuzzer in the project submission archive.

- Per-test-case timeout: 100 milliseconds
- Random number generator seed: 12345
- Rlogin node: node name
- Number of test cases: 1,234,567
- Wall clock time: 86,400 seconds
- Test case throughput: 14.29 test cases per second
- Total crash events: 120
- Distinct crash-inducing inputs: 2
- Total timeout events: 30
- Distinct timeout-inducing inputs: 1
- Crash-inducing test case: A1b; signal: SIGSEGV; occurrences: 70; first found: 12 seconds; replay: same signal
- Crash-inducing test case: C3d; signal: SIGABRT; occurrences: 50; first found: 45 seconds; replay: same signal
- Timeout-inducing test case: X2y; occurrences: 30; first found: 90 seconds; replay at 100 milliseconds: timed out; replay at 1000 milliseconds: timed out
- 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.
- Post question 4: A sentence or two.
- Source code: include in submission archive

Objective 8: Loading and Saving Seeds

Your fuzzer can now continue testing after crashes and timeouts, but it still starts each run without an existing seed corpus. Starting with useful seeds can help a fuzzer reach interesting program behavior sooner, because it can mutate those inputs instead of first having to generate them from scratch. However, simply adding more seeds is not always helpful: inputs that exercise the same behavior can consume space and mutation effort without increasing coverage.

In this objective, you will reuse the same instrumented target from Objective 7 and add support for loading and saving seed files. The target still receives a three-character alphanumeric string as its command-line argument. Your fuzzer will read test cases from files, then pass their contents to the target using the existing fork/exec interface. This lets you explore the effect of an initial seed corpus while keeping the target and its input format the same.

Initialize the seed pool and coverage history by executing the test cases in a folder called **seeds**. Add only normally terminating, coverage-increasing inputs to the pool, following the policies from Objectives 5 and 7. Write those inputs, including useful initial seeds, to a folder called **interesting** as you discover them. Saving the corpus preserves useful test cases for inspection and future fuzzing runs.

Pre-objective questions:

1. What is the impact of saving non-distinct test cases to the seed pool?
2. How would you choose an initial seed corpus for this target's three-character input grammar?
3. Why is it important to write interesting test cases out to a file?

Objective Steps:

1. Reuse the instrumented SUT from Objective 7, with the same three-character alphanumeric command-line input, edge hit count tracing, and crash and timeout handling.
2. Modify your fuzzer to read and process initial test cases from the `seeds` folder.
 - (a) Create a small initial corpus of `.txt` files, each containing exactly one three-character alphanumeric test case without a trailing newline. Record the filenames and their contents in your report.
 - (b) Read and validate each file's contents, then pass the test case itself as the target's command-line argument. For example, a file containing `A1b` should cause your fuzzer to execute the target with `A1b` as its input argument.
 - (c) Execute every initial seed using the same timeout and outcome analysis as a generated test case. Update the seed pool and global hit count maxima only for normally terminating, coverage-increasing inputs. Record initial seeds that crash or time out separately, as in Objective 7.
3. Write each normally terminating, coverage-increasing test case to its own file in the `interesting` folder, including any initial seeds that meet this criterion. Preserve the exact input bytes without a trailing newline, and use distinct filenames so that new findings do not overwrite earlier ones.
4. Perform one 24-hour trial of fuzzing the target program with your coverage-guided fuzzer.
 - (a) Reset the seed pool and coverage history before loading the initial seeds. Choose a per-test-case timeout and keep it fixed throughout the trial. Save this trial's outputs separately from earlier experiments.
 - (b) Start the trial timer before executing the initial seeds. Continue fuzzing after crashes and timeouts, as in Objective 7. Stop launching new test cases after 86,400 seconds; allow any remaining execution to finish or reach its per-test-case timeout, reap it, and record the actual elapsed wall clock time.
 - (c) Use the unattended-run setup from the Preliminaries. Record the random number generator seed, rlogin node, total executions, elapsed time, and crash and timeout findings. Include initial seed executions, crashes, and timeouts in the execution count. Calculate the trial's throughput as total executions divided by actual elapsed seconds.
 - (d) Replay abnormal inputs independently after the trial to verify that they reproduce, following Objective 7. Exclude these replays from trial measurements. Compare findings with Objective 7; random variation means that the sets of discovered inputs need not match exactly.

Post-objective questions:

1. How do your results compare with Objective 7, and what role might the initial seed corpus have played?
2. How would you increase your fuzzer's performance?
3. Should test cases causing timeouts and crashes also be written to disk?
4. Should test cases causing timeouts and crashes be added to the seed pool? Explain your answer.

What to submit: submit `obj8.txt` containing your measurements and answers formatted as shown below (without the preceding bullet). List every initial seed file and its contents, and report crash and timeout findings in the format used in Objective 7. Report the measurements from this single trial; averages and standard deviations across trials are not required. The values below are examples. Include `obj8.c` in the project submission archive (see Submission Checklist for details).

- Number of trials: 1
- Initial seed file: `seed1.txt`; contents: `A1b`
- Initial seed file: `seed2.txt`; contents: `C3d`
- Per-test-case timeout: 100 milliseconds
- Random number generator seed: 12345
- Rlogin node: node name
- Number of test cases: 1,234,567
- Wall clock time: 86,400 seconds
- Test case throughput: 14.29 test cases per second
- 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.
- Post question 4: A sentence or two.
- Source code: include in submission archive

Objective 9: Fuzzing Real-world Software

It's time to show what your fuzzer can do on real-world software. In this objective, you will fuzz the program `pdftotext`.

Objective Steps:

1. Download `xpdf` source code (the application suite containing `pdftotext`): <https://www.xpdfreader.com/download.html>
2. Verify that you can build `xpdf`
 - (a) Follow the build instructions in the `INSTALL` document.
 - (b) In addition to what's stated in that document, I needed to install the following packages using my OS's package manager: `build-essential` and `libstdc++-14-dev`.

- (c) Note that you do not need `Qt` as we won't be testing the GUI.
- 3. Download seeds for your target: https://github.com/unifuzz/seeds/tree/master/general_evaluation/pdf and place them in the `seeds` folder that your fuzzer expects to find seeds in.
- 4. Modify your fuzzer to read and mutate binary PDF seed files, replacing Objective 8's restriction to three-character `.txt` seeds. Write each generated test case to a file and pass its filename to `pdftotext` as the input argument; preserve the exact binary contents of the test case.
- 5. Ensure your coverage instrumentation and analysis work with the larger coverage bitmap that `pdftotext` produces.
- 6. Rebuild `xpdf` with coverage instrumentation that tracks edges:
 - (a) You have to tell `CMake` to use the desired compilers: `clang` and `clang++`
 - (b) You have to tell the compiler to include the correct coverage instrumentation
 - (c) You have to tell the linker where to find your coverage tracing handlers
 - (d) Here is the command I use in my build directory: `cmake -DCMAKE_BUILD_TYPE=Release -D CMAKE_C_COMPILER=clang -D CMAKE_CXX_COMPILER=clang++ -D CMAKE_C_FLAGS="-fsanitize-coverage=inline-8bit-counters /Users/mdhicks2/objective8-pc-guard.o" -D CMAKE_CXX_FLAGS="-fsanitize-coverage=inline-8bit-counters /Users/mdhicks2/objective8/trace-pc-guard.o" ../`
 - (e) Verifying that the target was built with coverage instrumentation:
 - i. The compiler will display warning messages that there were unused files sent to the linker (only the object file with the `main` function needs to be linked to your coverage instrumentation handlers).
 - ii. Use `objdump -D target` to disassemble the binary and look for calls to coverage sanitizer functions.
- 7. Perform research-quality experiments:
 - (a) Use `rlogin` (compute services for student use provided by the Computer Science Department) to run your fuzzing trials.
 - (b) Perform three 24-hour trials.

Post-objective questions:

1. Why is the coverage bitmap for this target so much larger than previous targets?
2. What do you expect the challenges to be to adapt your fuzzer to work with a different real-world program?

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 project submission archive (see Submission Checklist for details).

- Total number of possible edges: 12,345
- Average number of edges covered: 12,345
- Average number of test cases executed: 1,234,567
- Std. Dev. of the number of test cases executed: 123,456

- Std. Dev. of the number of edges covered: 123,456
- Post question 1: A sentence or two.
- Post question 2: A sentence or two.
- Source code: include in submission archive

Submission Checklist

Upload to Canvas a gzipped tar file named `project3.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. Replace `pid1` and `pid2` with your team members' PIDs. Run the following command from the directory containing your submission files:

```
tar -zcf project3.pid1.pid2.tar.gz \
  obj[1-9].txt srcFiles[13456789] ai.txt
```

The tarball should contain only the files below:

- Objective 1: `obj1.txt` and `srcFiles1`
- Objective 2: `obj2.txt`
- Objective 3: `obj3.txt` and `srcFiles3`
- Objective 4: `obj4.txt` and `srcFiles4`
- Objective 5: `obj5.txt` and `srcFiles5`
- Objective 6: `obj6.txt` and `srcFiles6`
- Objective 7: `obj7.txt` and `srcFiles7`
- Objective 8: `obj8.txt` and `srcFiles8`
- Objective 9: `obj9.txt` and `srcFiles9`
- `ai.txt`: a text file listing the AI tools and prompts you used to complete the assignment

Each objective's plain text file must contain its required measurements and answers. For each programming objective, include in your `srcFiles` folder a `Makefile` that includes targets that compile and run your fuzzer.

Your files can make use of standard C libraries and the provided default utilities on the servers that make up `rlogin.cs.vt.edu` but your solutions must be otherwise self-contained. Do not include any generated files with your submission. Be sure to test that your solutions work correctly on `rlogin` servers.