CS161 Computer Security
Spring 2008

Final Design Document

Eric Chang
Mike Hughes
Manu Srivastava
Adarsh Uppula

An Overview of a PCFG

We are using a Probabilistic Context-Free Grammar (PCFG) in order to generate valid PostScript input.

Example of a PCFG for a simple calculator:

```
//Non Terminal Symbols
Start → Block (p = 1.0)
Block → Statement NewLine Block (p = 0.8) | Statement NewLine (p = 0.2)
Statement → Number Operation Number (p = 1.0)
Operation → Addition (p = 0.25) | Subtraction (p = 0.25) | Multiplication (p
= 0.25) | Division (p = 0.25)

//Terminal Symbols
Number → [0-9]+
NewLine → '\n'
Addition → '+'
Subtraction → '-'
Multiplication → '*'
Division → '/'
```

To randomly generate valid input data to our simple calculator, we start at the "Start" symbol, randomly choose a production out of all the possible productions, and then recurse. Each production eventually boils down to Terminal symbols, which represent the actual bytes that constitute a valid input.

As an example run:
Currently at "Start":
Result: Because there is only one possible production, we return the result of evaluating 'Block.'

Currently at "Block":
Result: Output Generator chooses production "Statement Block" (w/80% chance). We evaluate "Statement" first and "Block" second.

Currently at "Statement":
Result: Because there is only one possible production, we return the result of evaluating 'Number' Operation' Number.' These recursive values return 2 * 6 (multiplication had a 25% chance of being selected from the Operation production)

Currently at "Block":
Result: Output Generator chooses production "Statement" (w/20% chance). We evaluate "Statement" and eventually get 5 + 4 (addition had a 25% chance of being selected from the Operation production)

Thus, our valid output is:

```
2 * 2
5 + 4
```

Implementation

Writing the Output Generator

The output generator will parse a passed-in grammar file and then generate *mostly legal* random output. We occasionally generate illegal outputs to test for the error-handling capabilities of the program. The biggest challenges are in writing the parser for the grammar file and implementing the appropriate data structures to aid random output generation. Because we want our fuzzer to work with any type of input file, we design a simple grammar that is used to specify another grammar. For example, our PostScript specification is written in a grammar that our Output Generator understands. To use this same fuzzer for different file formats, we just have to create new grammar files for those new formats.

The output generator will also be responsible for "driving" the test application. In other words, it will have the task of feeding the randomly-generated output as input to the process we are testing and then recording the results.

Writing the PostScript Grammar

PostScript is a full-fledged (albeit antiquated), Turing complete language, which means that a lot of effort will go into precisely and carefully defining this grammar file. In order to find all the bugs in pstotext, we will handle in our grammar and generator all the potentially bug related Postscript operators which are given to us. By the end of the project, we can expect to have found all 20 bugs.

Our biggest challenge in this phase is defining a thorough (and correct) specification that fleshes out all possible corner cases for the grammar. The second biggest challenge is attaching the appropriate probabilities to each production. For example, in the calculator grammar listed above, assigning uniform probabilities to the two possible 'Block' productions means that our expected output length is two lines. If we favor a bigger output file, then we must skew the probability in favor of the first production.

Once the format of the grammar file has been decided, both phases can make progress in parallel.

Pseudorandom Number Generation

By utilizing the random module in **python**, we will be able to take advantage of the MersenneTwister algorithm that is built-in. The MersenneTwister is purportedly one of the most extensively tested random number generators in existence. It is also a deterministic pseudorandom number generator, so by saving our seeds, we'll be able to replay sessions, meeting one of our major design requirements. We don't need it be cryptographically secure, just uniform in distribution of random number outputs. However, to ensure that the seeds that we generate for the MersenneTwister are themselves random, we utilize the python SystemRandom class, which uses the underlying operating system's cryptographically secure, non-deterministic PRNG to create a positive 64-bit integer, which has more than enough variablility to satisfy our needs.

Testing

The main part of the testing will be on whether or not the grammar we generate is valid postscript. Since pstotext does not output anything if it fails, we needed to find external tools that can output information even when our postscript file was invalid. To test our program, we will open up the generated file in gsnd. Since this program use the same ghostscript backend as pstotext, they let us simulate running postscript in pstotext while giving us useful debug information. Testing in gsnd allows us to manually step through the stack to see what is on the stack at various stages of execution, and it also allows us to go directly to the point in our postscript file that causes the parser to fail. Essentially, we are using ghostscript to directly test our postscript's validity. Since ghostscript is a command-line program, it will be easy to test all generated files for syntactical correctness (assuming, of course, that ghostscript is bug-free itself). To automate testing, we can write a script that opens up all our generated random input files and records any errors.

Web Scraper

In addition to our fuzzer, we also created a web scraper as a backup solution. While designing the fuzzer using a PCFG allows us to tweak our grammar to output theoretically any combinations of output that we want, there was still the problem that we did not potentially cover the entire range of possibilities that postscript allows. We could have been following Adobe's specifications too closely, or simply not imagined a production necessary to induce a bug.
The web scraper avoids crawling the entire web for postscript files (and needlessly downloading irrelevant HTML in the process) by querying Google for the PostScript filetype and iterating over those results, feeding them into psttotext. Each file's URL, MD5 sum, as well as the exit code and bug number generated (if any) is logged. If a bug number is encountered, it is added to the ignore list. The Google query can utilize any of the standard and advanced features, so if, for example, we wanted to try to feed it Japanese characters, we could add the command line option --query 'site:.jp'. The scraper is flexible in that it allows the filetype and test command to be changed from the command line, so that we could have just as easily been scraping .doc files and feeding them to Microsoft Word. Consult the README for more information.

Delta Debugger

It was our original intention with the web scraper to reverse-engineer any additional bugs found by it so that we could understand why our grammar lacked the productions that induced those bugs. In order to reduce input files found on the web to a reasonable size so that their inspection was tractable, we created a delta debugger. The framework that it utilizes was created Andreas Zeller, the originator of the delta debugging technique, but we made

significant modifications to it and created a front-end for it for use with our project. The delta debugger has significant memory requirements for caching results, and even after increasing memory on one of our systems to 4GB, it was still prone to running out of resources. Additionally, the time to complete a full delta debugging session could take upwards of 12 hours, likely due to Python's garbage collector running over gigabytes of memory. However, results from it were quite good, and we reverse-engineered a few of the bugs with it. The most impressive input minimization was for bug 15, which was reduced to two characters.

Additional Fuzzing Policies

Because our grammar was simply not generating all of the possible bugs, we added an additional policy that cuts up the input from the grammar, reorders it, and feeds it into the given command. This proved to get us "out-of-the-box" and found additional bugs. We also read that this is a technique employed by

Frequently Asked Questions

"In general, what form does the specification for a testing input take?"

Specification for the input is accomplished via a BNF grammar specified by the end user. If one wishes to generate a particular type of input (e.g. Postscript, html, etc.), then they simply describe the input in a grammar file and feed it to the fuzzer to generate specifically tailored random input (see the example at the beginning of this document)

"How is that specification expanded into a test input with a PRNG seed?"

When defining the grammar, a particular symbol can have multiple productions. For example, if I wish to output an arithmetic operation, the arithmetic operation could be add, subtract, divide, etc. The input generator will randomly select one of these operations and use that as part of the input; therefore, the seed influences the sequence of random numbers generated and hence the input that is generated. On a more general level, the seed also influences input from the entire grammar file such that generation with each seed produces substantially different output files.

"Within this framework, how are PostScript inputs specified?"

PostScript inputs are specified in a .grm (grammar) file using a BNF grammar. The fuzzer, based on the 'postscript' SPEC argument to –fuzz-file (or –fuzz-string) determines that the user wants to generate random postscript input and uses the corresponding grammar file to do so.

"What space of PostScript documents should this search / sample from?"

The space of our fuzzer includes all the postscript commands that John provided for us. In addition, our out space includes postscript primitives and objects such as numbers, strings, procs, and others that the postscipt commands require to execute successfully. In terms of our web scraper, the space of PostScript documents would be a significant portion of the Internet, as indexed by Google.

"What implementation-level issues arose, and how were they solved?"

The biggest implementation-level issue that we had arose when replaying seeds. When our fuzzer output the floating-point seed captured from search mode, it would automatically round our seed. This caused the fuzzer to print a different seed than the one used to generate the input file. We changed our seed format to long integers to avoid this issue. We also had problems with the delta debugger being killed on the instructional machines, and running out of memory on our home computers. We solved this by adding more RAM.

Labor Breakdown

Eric: Write up for Milestone 2. Final design doc. Set up repository and build system. Set up server for testing. Contributed to the grammar file. Write up for final design document.
Mike: Write up for Milestone 1. Coding the output generator as well as testing output generator. Contributed to the

grammar file. Created the web scraper for grabbing postscript files off of the internet. Wrote delta-debugger.
Adarsh: Write up for Milestone 2. Post Milestone 2 output generator implementation. Testing of skeleton code. Contributed to the grammar file.
Manu: Coding the skeleton code for fuzzer. Coding the main output generator and defining the grammar that will be used to specify the PostScript Grammar. Contributed to the grammar file.