Lecture 4: Dynamic Analysis and Fuzzing

Lecturer: Suman Jana Scribe: Jonas Guan

Feb 21, 2019

Presentation Logistics

Starting from the class on March 7th, students will begin to individually present research papers of their choice in class. Please email the paper you would like to present to Suman by February 28th, so that he can schedule related papers near each other. Each class will contain of 1 student presentation, but this may change depending on how much time we have left. Presentations should be approximately 45 minutes of material and 15 minutes of discussion.

Selecting a Research Paper / Research Project

The primary goal of this class is to explore how to use ML to solve program analysis problems, and this field contains many interesting subdomains you can explore for your presentation and research project. Try to concentrate your effort when selecting your topic: though not required, the paper you present should ideally be in the same subdomain as your research project. As this is a research course, your aim should be depth instead of breadth.

Some subdomains you can look at include:

1. **Improving static/dynamic/symbolic analysis with machine learning:** These are traditional fields in program analysis. Potential research topics include using ML to cut down false positives, an issue that plagues static analysis, or improve the efficiency of fuzzers. A sample paper on symbolic analysis is on the course website (Neuro-Symbolic Execution: The Feasibility of an Inductive Approach to Symbolic Execution); it investigates using neural networks to make symbolic execution more scalable.

2. **Program synthesis:** This field studies how to automatically construct programs given a task. For example, given a set of unsorted lists and their sorted counterparts, automatically learn a sorting algorithm. Traditionally this domain uses SMT solvers, but recent work has shown that neural networks achieve good results (DeepCoder: Learning to Write Programs). A related domain is latent representation, which is only interested in the correctness of the generated program; in contrast, program synthesis is interested both in correctness and interpretability of the program.

3. **Semantic similarity of binaries:** Given program binaries, this field studies how to automatically learn the semantic similarities between the programs. Syntactic similarity between programs is easy to learn, but semantic similarity is a difficult problem. For example, the x86 instructions *xor eax, eax* and *mov eax, 0* are semantically identical, but are represented as different instructions.

4. **Code deobfuscation:** This field studies how to automatically deobfuscate code into a readable format. Program code is often obfuscated, either purposefully by the coder (e.g. malware author trying to thwart analysis) or by a compiler (stripping identifiers for efficiency). Code deobfuscation aims to learn the semantic meaning of obfuscated code and represent it in an interpretable format without running the code. A paper on the course website (Predicting Program Properties from "Big Code") in this field studies how to take stripped JavaScript as input and produce JavaScript with meaningful variable names as output. A closely related field is code optimization using ML.

5. **Generic program representation:** In program analysis, programs are traditionally represented as control flow graphs, but this may not be a representation that is well suited for ML based approaches. This field investigates program representations that are more suitable for machine learning, and how to learn these representations. In some sense, this is the foundation for all research areas in program analysis with ML. A sample paper in this field (code2vec: Learning Distributed Representations of Code) looks at how to translate a program into meaningful values in a vector.

You may also choose papers from relevant subdomains not listed above. Since meaningful discussion is only possible if everyone is on the same page, try to make sure you read the papers before their presentations and come to class with questions.

A 1-page preliminary project proposal for your research project is due on March 15th.

Dynamic Analysis

Previously we studied static analysis and symbolic analysis and found that both have several disadvantages. Static analysis may never converge for large code bases, and even if the analysis converges it contains many false positives due to the info lost when merging paths. Symbolic analysis is highly dependent on the solver and cannot know if a path is feasible or not until the solver is called; converting a program to symbolic formula is also computationally expensive.

On the other hand, dynamic analysis has no false positives, because every path analyzed is actually executed and therefore must be feasible. The difficulty of dynamic analysis is achieving high code coverage.

There are two main approaches to dynamic analysis: verifying properties at runtime through **instrumentation**, and **path exploration** through fuzzing. The instrumentation approach is performed by inserting checks into the program code. For example, to check if the value of x is ever 0, we can add an if statement to check if x = 0 after each instruction, and if so terminate the program and report this. This example is naïve and inefficient but represents the basic idea. The fuzzing approach is performed by generating a set of inputs that achieve high code coverage and running the program with these inputs to observe buggy behavior. An idea related to fuzzing is

regression, which runs the program with a set of common inputs to test its robustness under normal use. In contrast, fuzzing tests for robustness against abnormal inputs.

Fuzzing

The concept of fuzzing was pioneered by Barton Miller at the University of Wisconsin in 1989, who was inspired by the observation that some of his failed attempts to login to his remote machine crashed the system. He became interested in whether programs are in general robust to random input, and conducted an experiment by sending random inputs to common programs and observing their behavior. The results showed that an overwhelming number of programs were not robust against such inputs and crashed. In security, the goal of fuzzing is often not to just crash a program, but also to find buffer overflow vulnerabilities.

Miller's experiments are an example of **blackbox fuzzing**, where the inputs are selected without regard to the analyzed program's logic. However, this primitive approach has the severe weakness that most inputs are filtered out at shallow branches. Modern programs commonly employ checks at shallow branches to ensure that the structure of the input is valid, and the probability that the fuzzer generates an input that adheres to the expected structure is low. For example, ELF parsers usually check that the first 4 bytes of an ELF file is its magic number and exits if this condition is not satisfied. Later we will look at enhancements to this fuzzing approach.

There are two components to fuzzers: an **input generator** that automatically generates test cases, and a **monitor** that detects errors in the program at runtime. There are multiple types of error detection. A simple error detection is to check if the program has crashed, and if so report the type of the crash. A more advanced detection would be to check for dynamic memory errors using an instrumentation tool such as Valgrind. This method will catch more bugs, but as tradeoff it is more expensive to run.

An enhancement to random blackbox fuzzing is **mutation-based fuzzing**. This enhancement aims to address the problem that random inputs are often filtered out at shallow branches. In this approach, the generator uses valid inputs that can reach deeper branches as seeds, and then mutate these valid inputs to generate test cases. The mutations can be performed randomly or based on some heuristics, for example selecting certain grammar rules in context-free grammar to violate. The disadvantage of this approach is that it lacks a feedback loop to improve mutations based on past performance. Furthermore, code coverage of mutation-based fuzzing largely depends on the coverage of the input seeds.

As an example, let's study how to fuzz a PDF parser using mutation-based fuzzing. We first collect a corpus of PDF files using google, then filter out a subset of interesting PDFs to use as seeds. The goal of filtering is to select a subset that maximizes code coverage. The metric for filtering can either be the distance between examples in the input space, or distance between examples in the program space, i.e. find examples that trigger paths that are far from each other by some

distance metric. The latter approach is usually preferred as it more directly increases branch coverage in the program, but is more costly as it requires execution. A critical problem could be that checksums will fail as we mutate valid input, which again causes programs to terminate at shallow branches. This problem also occurs in symbolic execution. To prevent this, we could disable checksum logic in the program before analysis.

A different enhancement to mutation-based fuzzing is **generation-based fuzzing**. Instead of mutating based on valid inputs, generation-based fuzzing generates inputs from scratch based on some description of valid input formats. This enhancement is better equipped to deal with complex dependencies such as checksums, but writing the generator is more difficult.

A key question in fuzzing is how much fuzzing is enough. Fuzzing will always hit saturation after a certain point, when it becomes prohibitively costly or impossible to find new paths. For mutation-based fuzzing, this occurs because mutations cannot reach paths that are far from any input seeds. For generation-based fuzzing, the test cases the generator can generate is finite and may not reach all paths.

Another question is how to evaluate fuzzer performance. Because the success of fuzzing is probabilistic in nature, we cannot measure performance solely by the number of bugs found. The locations of bugs are sparse, and some fuzzers may find more bugs than others just by being lucky. A better criterion for performance is **code coverage**. There are three methods to measure code coverage: line coverage, branch coverage and path coverage. Line coverage measures the number of lines in the source code that was executed; branch coverage measures the number of conditional jumps (branches) that was taken; and path coverage measures the number of paths that was taken. As a side note, because fuzzing is probabilistic, no fuzzer can provide formal guarantees of program robustness.

Using code coverage as the criterion for performance, we can further enhance fuzzing with **coverage-guided gray-box fuzzing**. This enhancement is based on mutation-based fuzzing, but also keeps track of mutations that perform better and bases future mutations on past successful mutations. More specifically, it uses a genetic algorithm that mimics the gene propagation process in nature. The algorithm's fitness function is defined using code coverage, and mutations that increases code coverage are kept for future mutations and those that don't die out. This is the fuzzing technique used by current state-of-the-art fuzzers such as AFL and libFuzzer.

Another method of fuzzing is **data-flow-guided fuzzing**. Previous discussed methods use the control flow of the program as guidance, but this fuzzing technique integrates the data flow of the program. The fuzzer instruments data flow and analyzes the inputs to comparisons at branches, then modifies test inputs and observe the effect on comparisons. The downside of this method is high overhead costs. The use of data-flow-guided fuzzing may also be limited: many programs like parsers consist mostly of logical rather than numerical comparisons, and thus the extra overhead incurred may not be worth it. Prototype implementations of data-flow-guided fuzzing can be found in libFuzzer and go-fuzz.

Two challenges for fuzzing are selecting seeds to efficiently achieve high code coverage and dealing with branches that are difficult to get past. Seed selection is a balance between coverage and efficiency. If the seeds don't cover enough different branches, many possible paths will not be tested. On the other hand, since seeds that cover similar branches causes redundancy, we must remove duplicate seeds to improve efficiency. Small seeds are preferred because they are usually faster for the program to process, which leads to a faster feedback loop for mutations.

Branches that have small sets of satisfying inputs is often difficult to get past. The following code snippet is an example:

```
void test (int n) {
   if (n == 0x12345678)
      crash();
}
```

In the worst case, we need 2^{32} attempts to guarantee we satisfy the conditional check. To combat this issue, we can either apply transformations to the checks, or remove the checks completely. However, we should be careful when modifying program logic to avoid false positives. Below is a transformation that greatly decreases attempts required to pass this branch by providing more granular feedback on each byte:

```
void test (int n) {
   int dummy = 0;
   char *p = (char *) &n;
   if (p[3] == 0x12) dummy++;
   if (p[2] == 0x34) dummy++;
   if (p[1] == 0x56) dummy++;
   if (p[0] == 0x78) dummy++;
   if (dummy == 4)
      crash();
}
```

As a quick recap, the two most important components of dynamic analysis is instrumentation and path exploration, and fuzzing is our method of conducting path exploration.

In the next class we will learn more about applying ML to fuzzing by studying a NN-based fuzzer developed by Suman's research group that is being published at a top security conference.