|  |  |
| --- | --- |
| **Slide 1** | Hi everyone.  Thank you for accepting this paper.  This research is done by a student of mine. Unfortunately, Maxim is away due to his family issue.  All this work was prepared by him with my little contribution.  One of the key problems with fuzzing is that this process is totally hidden for experts. For example, experts do not know what is happening with code now and they cannot control the fuzzing process without stopping it.  This work is focuses on way to solve the problem of identical code section analysis.  I hope that the results will be useful for you. |
| **Slide 2** |  |
| **Slide 3** | Today I am going to tell you about fuzzing and some popular types of fuzzers.  I'll show you main fuzzing strategy, which should be implemented to achieve good fuzzing results.  Then I'll present you a classical fuzzing scheme based on AFL.  I will show you, how to solve a problem of identical code analysis.  Finally, I’ll show you experimental results of the developed tool. |
| **Slide 4** | Fuzzing is a popular method of program analysis. It is a technique of automated testing when a program receives specially modified, incorrect data that can lead to its emergency state or undefined behavior. Of course, with the help of fuzzing, it is possible to identify a large number of errors and at least a large number of vulnerabilities. |
| **Slide 5** | Programs that implement the fuzzing process are called fuzzers. There are different types of fuzzers. (ENTER) For example, fuzzers associated with network protocols,(ENTER) the OS kernel, drivers, web apps, and localhost apps. This classification depends on fuzzing goal. (ENTER)  If we do have information about input data format, it is called white - box fuzzer. (ENTER) On the other hand, black - box fuzzers allow us to fuzz programs without having information about input data types.  Depending on the receipt of input data (ENTER), there are such fuzzers as (ENTER): **Generation** – the input is generated.  **mutational** --- individual bits of input words are changed. **combined** --- combining elements of generational and mutational fuzzing. |
| **Slide 6** | The fuzzing strategy includes the following main stages:  1. Static code analysis and determination of the attack surface. 2. Setting of the primary corpus that contains the input data that provides the greatest code coverage. These are diverse inputs. 3. Reducing the size of the primary housing so that the coverage does not change. 4. Conducting a standard fuzzing cycle.  5. Crash analysis that includes analysis of found bugs and identification of different types of vulnerabilities (). |
| **Slide 7** | Let's say a few words about AFL.  It stands for American Fuzzy Lop.  It is one of the most popular fuzzers.  It was developed by Mikhal Zalevsky at Google. This tool helps to discover a large number of vulnerabilities.  According to the classification, AFL refers to white-box source – based feedback fuzzers. This means that to analyze the program code using AFL, its source code and information about the specifics of the input data are required.  This is especially important for fuzzing with a dictionary, but we did not use a dictionary in our work. We also omit the QEMU mode, since the instrumentation of the program source code is important for us. |
| **Slide 8** | Let's look at such a simple model when, after instrumentation, the fuzzer starts a standard cycle. The fuzzer alternately extracts an element from the queue and mutates it, examining the execution traces. |
| **Slide 9** | Execution trace is a sequence of basic program blocks that have been visited due to a certain set of input data. This is implemented using edge instrumentation. This means that when moving from one base block to another, an ordered pair (A, B) of base blocks, called a tuple, is registered. |
| **Slide 10** | Let's go back to the model. It often happens when a fuzzer spends a lot of time analyzing identical sections of code without increasing coverage and without finding new bugs. This is registered when the fuzzer observes the same execution traces. At the same time, most of the code blocks may not be investigated or their research could be postponed. This is unacceptable, especially when the fuzzing process is quite long and can sometimes last days and weeks.  We called this problem as "identical code section analysis". |
| **Slide 11** | The slide shows a simplified diagram of the proposed fuzzing system.  It differs from the original AFL scheme by the adding the module that allows expert to manage or control in runtime the fuzzing process.  Expert can send control commands to the module.  Now, let's have a look at the process in more detail. |
| **Slide 12** | Receiving the run command from the user, |
| **Slide 13** | *fuzz.py* sends the control signals to the fuzzer,  starting the *afl\_fuzz* process with certain parameters and report table update time ~~(~~**~~T~~~~RT)~~** |
| **Slide 14** | During this time, the fuzzer performs an original AFL cycle: retrieving the next element from the queue, |
| **Slide 15** | mutating it, testing the program response |
| **Slide 16** | updating the queue and displaying a status window for the expert. |
| **Slide 17** | At the same time, the specially instrumented *has\_new\_bits()* function writes traces. |
| **Slide 18** | When time ~~(~~**~~T~~~~RT~~**~~)~~ is over, *fuzz.py* stops the process |
| **Slide 19** | reads the traces, and determines such characteristics as coverage productivity **~~part\_cov~~**~~,~~ **~~part\_exec, qcov, qrun~~**. |
| **Slide 20** | The results are presented to experts in the report table. |
| **Slide 21** | The expert analyzes the report table and sends the command to resume the process or redirect the flow. We can redirect the execution flow by selecting an element from the candidate list in the report table. The candidate list was defined by the module at the previous stage. These are queue elements that have not been tested yet or have comparatively better characteristics. At the same time, unexplored elements have a higher priority since they can potentially increase coverage. Having selected an element from the list, the operator sends a command to the module, |
| **Slide 22** | which modifies the queue order ~~(new element should be in the first place)~~ and makes appropriate changes for the fuzzer service variables to avoid conflict. Finally, file with traces is cleared, the afl\_fuzz process resumes, then the cycle repeats again |
| **Slide 23** | Now I want to show how to detect a trace. At the instrumentation stage, a random integer is generated in each base block of the program, called the base block label. When moving from one base block to another, the label of the directed tuple is formed by applying the XOR operation and the right shift. AFL uses a special array in which 1 corresponds to the address – label of the tuple that is in the track. |
| **Slide 24** | To detect the trace, it is enough to instrument *the has\_new\_bits()* function, which updates the *trace\_bits[]* array at each iteration, adding the record lines to the file immediately after the next array update. |
| **Slide 25** | After AFL process starts, the user receives feedback in the form of a status window and a report table *fuzz.py*.  The expert interacts with the module through the command interface. The main field of the table is **part\_cov**.  The value of this variable is calculated as the ratio of the number of new unique tuples found **qcov** to the number of runs on the current **qrun** element. This is called *coverage productivity*. **Part\_execs** is defined as the program runs percentage on the current element of the total number of the program runs. |
| **Slide 26** | The fuzzing results are shown on this slide.  The module was checked using the tiff library utilities, which are designed to work with .tiff files. It is known that many utilities of this library contain a lot of vulnerabilities. Therefore, this library is quite suitable for testing the module and comparing it with other AFL-like fuzzers. AFL and aflFast were chosen as such fuzzers.  The fuzzing time is ~~T~~~~f~~ ~~=~~ 12 hours. This is not enough to have a quality program fuzzing, but it is quite enough to test the module, since the utilities have already been tested by the developers. The report table update time is ~~T~~~~RT~~ ~~=~~ 30 minutes. For each utility and each fuzzer, the total number of **unique crashes** and **total coverage** were determined.  Having analyzed the results of the graph, **we can conclude** that the proposed fuzzing system in the most cases finds crashes faster than other fuzzers. |
| **Slide 27** | We plan to consider the scheme of using the proposed solution with static analyzers by adding a visualization module. This will significantly reduce the search space, as well as improving the interactivity of the fuzzing process.  We also strive to make the deployment procedure simple for different platforms. |
| **Slide 28** | Thank you! |