Federal State Autonomous Educational Institution for Higher Education
National Research University Higher School of Economics

Information Security

BACHELOR'S THESIS

Research project

"Hybrid Fuzzing of the PyTorch Framework"

Prepared by the student of group 191, 4th year of study,
Larionov-Trichkine Theodor Arsenij

Supervisor:
PhD, Petrenko Alexander Konstantinovich

Consultant:
Kuts Daniil Olegovich, ISP RAS

Moscow 2022

Contents

Annotation

As the number and complexity of software systems continue to increase at a rapid pace, an ever-growing number of these systems are becoming critical to our daily lives.

AI takes this trend to a whole new level by allowing software systems to make decisions that were previously reserved for humans. With these advances in the field of information technologies, it is more important than ever to ensure that critical systems are robust and secure against cyber threats.

In this thesis, we will take a look at the problem of software security and how it can be addressed using automated analysis techniques. We will also improve several aspects of the existing hybrid-fuzzing tools and apply them to the PyTorch framework to detect bugs and vulnerabilities in its code.

Аннотация

Keywords

Hybrid Fuzzing, Program Security, Dynamic Analysis, PyTorch, AI Frameworks Security

1 Introduction

Software security is a growing concern in the modern world. With the rapid development of information technologies, the number and complexity of software systems have increased drastically. This has led to an increase in the number of software vulnerabilities as well as an increasing need for secure software development practices.

1.1 Memory Safety Vulnerabilities

Memory safety vulnerabilities are a particularly significant concern in software security. They refer to programming errors that can cause a program to access memory in unintended ways, potentially leading to system crashes, data leaks, or even full system compromise. Memory safety vulnerabilities are especially prevalent in large codebases written in memory-unsafe languages such as C and C++.

According to [11], for codebases with more than one million lines of code, at least 65% of security vulnerabilities are caused by memory safety issues in C and C++. The Chromium project security team also highlights the same point in their report [2]. This alarming statistic underscores the importance of addressing memory safety vulnerabilities in software development. Especially, for critical software systems, such as operating systems, web browsers, machine learning frameworks, and beyond.

1.2 AI and Security

In recent years, AI (Artificial Intelligence) has emerged as a key technology in many domains, including banking, healthcare, transportation, and more. With the rise of AI-powered applications, there is an increasing need for secure AI models and software systems that can withstand cyber threats, as these systems are often used to make critical decisions that affect human lives.

Of particular interest is the security of AI frameworks. Often, these systems are the foundation of AI applications. As such, vulnerabilities in AI frameworks can have a significant impact on the security of applications built on top of them.

One of the most popular AI frameworks is PyTorch [5]. PyTorch is an open-source machine learning framework developed by Meta (formerly Facebook). It is used by many companies and organizations, including Microsoft, Uber, Twitter, and more. Despite its popularity, PyTorch is not immune to security vulnerabilities, especially given that it is written in C++, a memory-unsafe language.

Considering the importance of PyTorch in the AI ecosystem, it is crucial to ensure that PyTorch is secure and robust against cyber threats.

1.3 Objective

Our objective in this work is twofold: to perform a comprehensive security analysis of PyTorch using hybrid fuzzing techniques with the goal of detecting and addressing any memory safety vulnerabilities present in the framework, and to enhance sydr-fuzz [9] - a hybrid fuzzing tool developed by ISP RAS.

2 Software Security Analysis Techniques

As we have seen in the previous section, software security is a question of paramount importance in the modern world. Due to the increasing complexity of software systems, it is no longer feasible to rely only on manual code reviews and testing to ensure that they are secure. Instead, a variety of automated analysis techniques have been developed to help developers detect and address security vulnerabilities in their software.

The security analysis techniques can be broadly divided into two categories:

- Static Analysis
- Dynamic Analysis

In this section, we will provide an overview of static analysis and then delve into a detailed examination of dynamic analysis techniques.

2.1 Static Analysis

A set of techniques known as static analysis involves analyzing the source code of a program without executing it. This approach allows to detect a wide range of problems in the code, potentially examining all possible execution paths.

Although static analysis tends to be more exhaustive, it suffers a lot from false positives as well as false negatives. Furthermore, static analysis tends to be very slow and resource-intensive, especially for large codebases.

To mitigate these concerns, dynamic analysis is frequently employed in conjunction with static analysis. Although it may not be as comprehensive as static analysis, it allows identifying issues that static analysis may miss.

2.2 Dynamic Analysis

Dynamic analysis, also known as fuzzing is one of the most popular techniques for finding bugs and vulnerabilities in software. It involves running a program with various inputs and monitoring its behavior. The goal of fuzzing is to detect error

conditions in the program by observing its behavior under different inputs.

Consider example 1. This program takes a string as an input and checks if the first four characters are equal to "FUZZ". If they are, the program crashes. Otherwise, it does nothing.

```
void crash(char* buf) {
    if (buf[0] == 'F') {
        if (buf[1] == 'U') {
            if (buf[2] == 'Z') {
                if (buf[3] == 'Z') {
                    *(int*)NULL = 0x1337;
                }
            }
        }
    }
}
```

Listing 1: Fuzzing example

The goal of a generic fuzzer would be to automatically find an input that would cause the program to crash.

The simplest way to do so would be to exhaustively test all possible inputs. While this works well in theory and is guaranteed to find the bug, it is not feasible in practice, as the number of possible inputs grows exponentially with the size of the input. For a program that processes a string of 10 characters, where each character can be any of the 127 ASCII characters, the total number of possible inputs is $127^{10} \approx 1.0915 \times 10^{21}$. This number is far too large to be tested in a reasonable amount of time. Instead, a smarter approach is required.

2.2.1 Fuzzers Overview

To compensate for the exponential growth of the input space, fuzzers use various techniques to guide the input generation. For example, state-of-the-art,

general-purpose fuzzer AFL++ [3] uses a technique called *coverage-guided fuzzing* to generate inputs that are more likely to trigger bugs. This technique involves instrumenting the program to collect code coverage information and then using this information to guide the generation of inputs towards unexplored parts of the program.

Another example of input generation techniques used by fuzzers is *grammar-based fuzzing*. This technique involves defining a grammar that describes the structure and syntax of valid inputs for a given program. The fuzzer then generates inputs that conform to this grammar, exploring different paths through the grammar to generate diverse inputs. This technique is used by various fuzzers, including Nautilus [1], Superion [10], Gramatron [8], and others.

Besides different approaches to input generation, fuzzers are also distinguished by the type of target they are designed to test. For example, Nyx [7] or kAFL [6] are fuzzers designed to work on a hypervisor level allowing to fuzz OS kernel, drivers, and other hard-to-test components. On the other hand, AFL++ or LibFuzzer are examples of general-purpose fuzzers.

2.2.2 Fuzz Testing Algorithm

While fuzzers might look very different on the surface, they all share the same basic structure and follow a similar algorithm. In the paper [4], the authors present a high-level overview of the fuzzing process.

Omitting some details, the fuzzing process can be summarized as follows:

1 Preprocessing - prepare a corpus of inputs, instrument the program to collect coverage information, etc.

2 Scheduling - select fuzzing strategies, etc.

3 Input generation - select an input from the corpus, mutate the input, generate new inputs, etc.

4 Input evaluation - run the program with the input, collect feedback (e.g. coverage information, crashes, etc.)

5 Continue fuzzing until a stopping condition is met (e.g. a timeout)

To implement the fuzzing process described above, a fuzzing loop can be used as shown in Algorithm 1.

```
queue ← construct_queue()
while should fuzz do
    input ← select_input(queue)
    input ← mutate(input)
    feedback ← run_program(input)
    if feedback is crash then
        report_bug(input)
    end
    if feedback is interesting then
        queue.push(input)
    end
end
```

Algorithm 1: Fuzzing loop

The fuzzing loop described in Algorithm 1 abstracts away details of the fuzzing process and allows us to focus on the individual components of the fuzzer. Each of these components can be customized to

2.3 Symbolic Interpretation

2.4 Hybrid Fuzzing

3 PyTorch Fuzzing

We now describe our PyTorch fuzzing methodology. We begin by describing our approach to analyzing PyTorch's attack surface

3.1 Attack Surface Mapping

3.2 Fuzzing Harness Development

4 Hybrid Fuzzer Improvements

4.1 Scheduling Symbolic Pointers Modelling

4.2 Utilizing Debug Information to Improve sydr-fuzz

5 Results

5.1 PyTorch Bugs

5.2 1 in 25

5.3 Annotate

6 Conclusion

References

1. Cornelius Aschermann et al. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019).
2. *Chromium project memory safety report*. https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed: 2023-02-12.
3. Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.
4. Valentin J. M. Manes et al. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2019. arXiv: 1812.00140 [cs.CR].
5. Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.
6. Sergej Schumilo et al. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.
7. Sergej Schumilo et al. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2597–2614. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo.

8. Prashast Srivastava and Mathias Payer. "Gramatron: Effective Grammar-Aware Fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, 244–256. ISBN: 9781450384599. DOI: 10.1145/3460319.3464814. URL: https://doi.org/10.1145/3460319.3464814.

9. Alexey V. Vishnyakov et al. "Sydr: Cutting Edge Dynamic Symbolic Execution". In: *CoRR* abs/2011.09269 (2020). arXiv: 2011.09269. URL: https://arxiv.org/abs/2011.09269.

10. Junjie Wang et al. *Superion: Grammar-Aware Greybox Fuzzing*. 2018. DOI: 10.48550/ARXIV.1812.01197. URL: https://arxiv.org/abs/1812.01197.

11. *What science can tell us about C and C++'s security*. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/. Accessed: 2023-03-14.