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 [24], 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 [6]. 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 [14]. 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 [22] - 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 us 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.

```
1  void crash(char* buf) {
2      if (buf[0] == 'F') {
3          if (buf[1] == 'U') {
4              if (buf[2] == 'Z') {
5                  if (buf[3] == 'Z') {
6                      *(int*)NULL = 0x1337;
7                  }
8              }
9          }
10     }
11 }
```

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++ [7] 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 [23], Gramatron [19], 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 [17] or kAFL [16] are fuzzers designed to work on a hypervisor level allowing to fuzz OS kernels, 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 [11], 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.

Algorithm 1: Fuzzing loop

```
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
```

The algorithm presented in Algorithm 1 provides a simplified representation of the fuzzing process that allows us to concentrate on specific components of the fuzzer.

The natural modularity of the fuzzing process has proven to be beneficial, as shown by the example of LibAFL [8]. This fuzzer has taken advantage of this modular design by enabling users to create their custom implementations of individual components, thereby allowing greater flexibility and customization of the fuzzing process to tackle specific challenges or meet particular requirements.

2.2.3 Individual Fuzzer Components

To further understand the different techniques used by fuzzers, let's take a look at some papers that focus on individual components of the fuzzing process.

One important component is the mutation engine used to generate new inputs from existing ones. In the paper [2], the authors propose a new mutation strategy called Redqueen, which utilizes feedback from previous executions to build input-to-state correspondence. This allows Redqueen to solve simple comparison-based constraints, such as the one in the Listing 2, assuming the input-to-state mapping is one-to-one.

Another important component is the input selection strategy. In the paper [18], the authors propose a new seed selection strategy called *K-Scheduler*, which

```
1 if (strcmp(buf, "FuZzing1sCOOL") == 0) {
2     *(int*)NULL = 0x1337;
3 }
```

Listing 2: Example solvable by Redqueen

uses graph centrality analysis to select seeds that are more likely to increase feature coverage. The authors show that this strategy outperforms other seed selection strategies, such as *Entropic*, or next-best AFL-based seed scheduler *RarePath* by 25.89% and 4.21%, respectively.

2.2.4 Challenges

In conclusion, fuzzing has become one of the best techniques to find bugs in software. Through extensive research, various techniques have been developed and applied to different components of the fuzzing process, such as mutation engines, input selection strategies, and others. However, there are many challenges that haven't been solved yet. Ranging from the scalability of fuzzing to the quality of the generated inputs, there are many areas that can be improved.

One particularly challenging problem is the generation of inputs that satisfy complex constraints. Even with the most advanced fuzzers, it is still difficult, if not impossible, to generate inputs that satisfy constraints such as the one in Listing 3. This happens because the constraints may involve complex arithmetic operations, or other hard-to-resolve dependencies between input values. As a result, traditional fuzzing techniques that rely on random or mutation-based input generation with coverage feedback are not sufficient to solve this problem.

```
1 void vuln(int key) {
2     if (key * 0xa9a57b == 0x1337beef) {
3         error();
4     }
5 }
```

Listing 3: Example solvable by symbolic execution

That is where another set of techniques called *Symbolic Interpretation* comes into play.

2.3 Symbolic Interpretation

Symbolic interpretation, also known as symbolic execution, aims to solve the problem of generating inputs that satisfy complex constraints, such as the one in Listing 3.

Essentially, symbolic execution is a powerful technique that enables us to run a program with symbolic inputs instead of concrete ones. By treating program states as sets of constraints on these inputs, we can systematically explore different paths through the code and generate new test cases that can reveal hidden bugs.

For example, the state of the program in Listing 3 can be defined by this equation: `key * 0xa9a57b = 0x1337beef`. Depending on whether this equation is satisfied or not, we either take the `true` or the `false` branch. By solving this equation, we can generate an input that would open up the `true` branch, and thus trigger the `error()` function. For this particular example, the input `0x1337beef / 0xa9a57b = 0x1d` would satisfy the equation and trigger the error. What is notable, for classical fuzzers, it would require exhaustively testing all possible inputs to find this one, as there is no feedback which would guide the fuzzer towards this input.

Now that we have covered the fundamentals of symbolic execution, let us delve deeper into the various components of the symbolic execution process.

2.3.1 Symbolic Representation

Symbolic representation is the initial stage of the symbolic execution process where program variables and inputs are represented as symbolic expressions that can be mathematically evaluated and manipulated.

To effectively build and update a program's symbolic state based on the instruction semantics, it is necessary to symbolically execute machine code instructions while simultaneously updating the symbolic state. A convenient approach is to use a dynamic binary analysis framework, such as Triton [15], which provides an API for symbolic execution and allows us to easily build symbolic expressions

from machine code instructions.

In the Listing 4, we can see an example of how Triton can be used to symbolically execute a program from Listing 3, and generate an equation for the conditional jump instruction.

```
1  from triton import *
2
3  >>> # Create the Triton context with a defined architecture
4  >>> ctx = TritonContext(ARCH.X86_64)
5
6  >>> # Symbolize data (optional)
7  >>> ctx.symbolizeRegister(ctx.registers.eax, 'sym_eax')
8
9  >>> # Execute instructions
10 >>> ctx.processing(Instruction(b"\xb9\x7b\xa5\xa9\x00")) # mov ecx,
   ↪  0xa9a57b
11 >>> ctx.processing(Instruction(b"\xf7\xe1"))             # mul ecx
12 >>> ctx.processing(Instruction(b"\x3d\xef\xbe\x37\x13")) # cmp eax,
   ↪  0x1337beef
13
14 >>> # Get the symbolic expression
15 >>> zf_expr = ctx.getSymbolicRegister(ctx.registers.zf)
16 >>> print(zf_expr)
17 (define-fun ref!14 () (_ BitVec 1) (ite (= ref!8 (_ bv0 32)) (_ bv1 1)
   ↪  (_ bv0 1))) ; Zero flag
```

Listing 4: Triton API example

Triton provides a powerful mechanism for interpreting machine code instructions and updating the symbolic state simultaneously. However, it may not be able to handle certain scenarios such as external library calls or complex OS-dependent instructions like `syscall`. In such cases, it may be necessary to actually run the program and symbolically execute as much as possible, while concretizing the remaining instructions that cannot be symbolically executed.

This approach is commonly used by various symbolic execution engines, such as QSym [25] and others. In the case of QSym, the symbolic execution engine simply concretizes the instructions that cannot be symbolically executed, and then continues with the symbolic execution. This approach is called *concolic execution* and is widely used in symbolic execution engines.

2.3.2 Dynamic Constraints Collection

A key component of concolic execution is dynamic constraints collection, which is performed using a concrete executor that runs a program with specific inputs and collects constraints on those inputs.

To accomplish this, dynamic binary instrumentation (DBI) frameworks are commonly employed. DBI frameworks allow for program instrumentation and constraint collection on-the-fly, providing a convenient solution to perform dynamic analysis. Popular DBI frameworks, such as Pin [10], DynamoRIO [5], and QEMU [4] can be used for this purpose.

Typically, per-instruction callbacks are used in DBI frameworks to collect constraints as the program is executed. When a callback is triggered, the corresponding instruction is examined, and the constraints on the input values are collected. These constraints are then combined to form a path condition that represents all the constraints encountered during execution.

With the constraints collected, we can now solve them and generate new inputs.

2.3.3 Constraints Solving

To solve the constraints collected during dynamic analysis, a constraint solver is needed. The solver takes the path condition generated from the collected constraints and generates new input values that satisfy the condition. To solve the constraints, a wide range of SMT solvers can be employed, such as Z3 [12], Bitwuzla [13], CVC5 [3], and others.

The efficiency and accuracy of the solver play a crucial role in the performance of concolic execution. In some cases, the solver may not be able to find a solution, or the solution may be too complex and time-consuming to compute. To address these issues, various techniques such as constraint simplification and pruning can be used to simplify the constraints and reduce the solution space.

Once the solver generates new input values, the program can be executed again

with the updated inputs, and the process of constraint collection and solving can be repeated. This iterative process continues until all paths have been explored, or until a specific goal, such as reaching a specific code location or triggering a specific vulnerability, is achieved.

To illustrate the process of constraint solving, we can use the example from Listing 4. In this example, we symbolically executed the `mul` instruction and generated a constraint on the `ZF` flag. We can now solve this constraint using the Triton API, as shown in Listing 5. The solution to the constraint is `sym_eax = 0x1d`, which is the value that would trigger the `error()` function.

```
1  >>> # Solve constraint
2  >>> ctx.getModel(zf_expr.getAst() == 0x1)
3  {0: sym_eax:32 = 0x1d}
4
5  >>> # 0x1d * 0xa9a57b is indeed equal to 0x1337beef
6  >>> hex(0x1d * 0xa9a57b)
7  '0x1337beef'
```

Listing 5: Triton getModel() API example

2.3.4 Benefits

With the ability to symbolically execute a program and generate new inputs that satisfy a specific condition, a few obvious benefits arise.

First, we can automatically generate inputs that would open up new paths in the program. This allows us to explore different program states and as a result, test different aspects of the program. This is called *path exploration* and is one of the main benefits that symbolic execution provides.

The second benefit is the ability to not only explore different paths, but also to check security invariants. For example, we can check if a specific code location is reachable, or if a specific condition can be satisfied. This is called *security invariant checking* and is another key benefit of symbolic execution. This technique allows us to check if a specific vulnerability can be triggered, for example, if it is possible to generate such an input that would trigger an out-of-bounds access. Consider the example from Listing 6.

```
1 void vuln(uint32_t index) {
2     char data[64];
3     if (index < 74)
4         data[index] = 0x37;
5 }
```

Listing 6: Security invariants checking example

The bug here is obvious – the `data` array is only 64 bytes long, but the condition `index < 74` checks that the index is less than 74. This means that any index greater than 63 would trigger an out-of-bounds write. With the automatic security invariant checking, we can check if it is possible to generate such an input that would trigger the vulnerability.

One particular example of a tool that implements this technique is Sydr. In the paper *Symbolic Security Predicates: Hunt Program Weaknesses* [21], the authors proposed a technique called *symbolic security predicates* that allows for automatic security invariant checking.

2.3.5 Challenges

While symbolic execution provides a lot of benefits, it also comes with a handful of challenges.

Symbolic Memory

One of the main challenges of symbolic execution is the ability to build a precise symbolic model of the program. While modeling simple program with scalar values is relatively easy, modeling memory with pointer operations introduces a new level of indirection that makes the process much more difficult. Additionally, performance drops significantly due to the substantial increase in formula size and complexity when compared to scalar-only modes.

Another crucial aspect of the challenge is determining approximate or exact boundaries for symbolic memory accesses. In general case, a symbolic memory dereference could access any memory location, which is infeasible to model. Therefore, we must find a way to limit memory access to a specific range. For instance,

in the paper *Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution* [9], the authors study this problem and explore various techniques to address it. One approach proposed involves the use of heuristics to determine the leftmost boundary by extracting the concrete portion from the abstract syntax tree of the address expression.

Unsolvable Constraints

Another challenge arises from the fact that modern SMT solvers are not perfect at solving all SMT problems efficiently. The SMT problem is known to be NP-complete, which means there is no known algorithm that can solve all SMT problems efficiently. As a result, although modern SMT solvers can handle many real-world problems, there are still scenarios where the solver may fail to find a solution or take an excessively long time to do so. This limitation can hamper the performance and effectiveness of the symbolic execution process.

Incomplete Symbolic Model

Additional obstacle of symbolic execution is the potential incompleteness of the symbolic model. It may not always be possible or practical to completely model a program symbolically, especially when dealing with interactions with the outside world, such as system calls. This can result in discrepancies between the symbolic program state and the real one, leading to unsolvable constraints that impede the effectiveness of the technique. However, some approaches, such as concolic testing, can help mitigate this challenge by combining symbolic and concrete execution.

Path Explosion

Finally, a significant challenge in various types of program analysis, including symbolic execution, fuzzing, and path-sensitive static analysis, is the path explosion problem. This issue arises because the number of control-flow paths in a program increases exponentially with the program's size. As a result, the symbolic execution engine may not be able to explore all the paths within a reasonable

timeframe. To overcome this challenge, researchers have proposed various techniques, such as path pruning and constraint prioritization, that aim to reduce the number of explored paths without compromising the completeness of the analysis. However, these techniques have their limitations, and achieving complete path coverage remains a challenging problem in program analysis.

2.4 Hybrid Fuzzing

One of the first attempts to combine symbolic execution was presented as a [20]

2.4.1 Examples

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. Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019).
3. Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9.
4. Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.
5. Derek L. Bruening and Saman Amarasinghe. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". AAI0807735. PhD thesis. USA, 2004.
6. *Chromium project memory safety report*. https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed: 2023-02-12.
7. Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.
8. Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 1051–1065. ISBN: 9781450394505. DOI: 10.

1145/3548606.3560602. URL: https://doi.org/10.1145/3548606.3560602.

9. Daniil O. Kuts. "Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution". In: *CoRR* abs/2109.03698 (2021). arXiv: 2109.03698. URL: https://arxiv.org/abs/2109.03698.

10. Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, 190–200. ISBN: 1595930566. DOI: 10.1145/1065010.1065034. URL: https://doi.org/10.1145/1065010.1065034.

11. Valentin J. M. Manes et al. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2019. arXiv: 1812.00140 [cs.CR].

12. Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

13. Aina Niemetz and Mathias Preiner. "Bitwuzla at the SMT-COMP 2020". In: *CoRR* abs/2006.01621 (2020). arXiv: 2006.01621. URL: https://arxiv.org/abs/2006.01621.

14. 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.

15. Florent Saudel and Jonathan Salwan. "Triton: A Dynamic Symbolic Execution Framework". In: *Symposium sur la sécurité des technologies de l'information*

et des communications. SSTIC. 2015, pp. 31–54. URL: https://triton.quarkslab.com/files/sstic2015_slide_en_saudel_salwan.pdf.

16. 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.

17. 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.

18. Dongdong She, Abhishek Shah, and Suman Jana. "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 2194–2211. DOI: 10.1109/SP46214.2022.9833761.

19. 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.

20. Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016, pp. 1–16.

21. Alexey Vishnyakov et al. "Symbolic Security Predicates: Hunt Program Weaknesses". In: *2021 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 2021. DOI: 10.1109/ispras53967.2021.00016. URL: https://doi.org/10.1109%2Fispras53967.2021.00016.

22. 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.

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

24. *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.

25. Insu Yun et al. “QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.