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

С каждым днем увеличивается количество и сложность информационных систем. Вместе с этим, все больше систем становится критически важным для нашей повседневной жизни.

С появлением ИИ эта тенденция приобретает совершенно новый масштаб, позволяя информационным системам принимать решения, которые раньше оставались за человеком. С такими достижениями в области информационных технологий как никогда важно обеспечить надежность и защищенность критически важных систем от киберугроз.

В этой работе мы рассмотрим проблему безопасности программного обеспечения и то, как ее можно решить с помощью методов автоматизированного анализа. Мы также усовершенствуем некоторые аспекты существующих инструментов гибридного фаззинга и применим их к фреймворку PyTorch для обнаружения ошибок и уязвимостей в его коде.

Keywords

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

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 [35], 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 [9]. 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 [20]. 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 [32] - 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 input and checks if the first four characters are equal to "**FUZZ**". If they are, the program crashes. Otherwise, it does nothing.

```c
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++ [10] 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 [34], Gramatron [29], 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 [26] or kAFL [25] 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 [16], 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 [11]. 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 us 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 [28], the authors propose a new seed selection strategy called *K-Scheduler*, which

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

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, many challenges have not been solved yet. Ranging from the scalability of fuzzing to the quality of the generated inputs, many areas 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.

```c
void vuln(int key) {
    if (key * 0xa9a57b == 0x1337beef) {
        error();
    }
}
```

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 that would guide the fuzzer toward 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 [23], which provides an API for symbolic execution and allows us to easily build symbolic expressions from machine code instructions.

In 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.

```
from triton import *

>>> # Create the Triton context with a defined architecture
>>> ctx = TritonContext(ARCH.X86_64)

>>> # Symbolize data (optional)
>>> ctx.symbolizeRegister(ctx.registers.eax, 'sym_eax')

>>> # Execute instructions
>>> ctx.processing(Instruction(b"\xb9\x7b\xa5\xa9\x00")) # mov ecx, 0xa9a57b
>>> ctx.processing(Instruction(b"\xf7\xe1"))             # mul ecx
>>> ctx.processing(Instruction(b"\x3d\xef\xbe\x37\x13")) # cmp eax, 0x1337beef

>>> # Get the symbolic expression
>>> zf_expr = ctx.getSymbolicRegister(ctx.registers.zf)
>>> print(zf_expr)
(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 [36] 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 [15], DynamoRIO [8], and QEMU [5] 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 [18], Bitwuzla [19], CVC5 [4], 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.

```
>>> # Solve constraint
>>> ctx.getModel(zf_expr.getAst() == 0x1)
{0: sym_eax:32 = 0x1d}

>>> # 0x1d * 0xa9a57b is indeed equal to 0x1337beef
>>> hex(0x1d * 0xa9a57b)
'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 of symbolic execution is the ability to not only explore various execution paths but also to verify security invariants. With this technique, we can analyze whether a particular code location is reachable and if a given condition can be met. By employing this approach, we can identify potential vulnerabilities and evaluate if a particular input can trigger them. For instance, we can examine whether it is possible to generate an input that triggers an out-of-bounds access, as demonstrated in the code example presented in Listing 6.

```c
void vuln(uint32_t index) {
    char data[64];
    if (index < 74)
        data[index] = 0x37;
}
```

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 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* [33], 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 a 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 the 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* [14], 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

An 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

As we have seen in the previous sections (2.2.4, 2.3.5), both fuzzing and symbolic execution have their strengths and weaknesses. One of the main problems of fuzzing is the inability to generate complex inputs, while symbolic execution suffers from the path explosion problem and execution speed.

To overcome the limitations of both techniques, researchers have proposed a hybrid approach called *hybrid fuzzing*. Hybrid fuzzing combines fuzzing and symbolic execution to take advantage of their strengths and mitigate their weaknesses. In this approach, the fuzzer generates inputs and feeds them into the symbolic execution engine. The symbolic execution engine then explores the different paths in the program, thus helping the fuzzer explore new code paths.

The primary benefit of hybrid fuzzing is that it is no longer limited by the fuzzer's inability to generate complex inputs. With the help of symbolic execution, the fuzzer can generate inputs that could pass complex checks and open up new paths in the program, leading to better code coverage and more thorough testing. Additionally, the symbolic execution engine can also help identify potential vulnerabilities by checking security invariants.

2.4.1 Approaches

One of the first tools to implement this approach was SAGE [13], which was later improved upon by Driller. In the paper *Driller: Augmenting Fuzzing Through Selective Symbolic Execution* [30], the authors introduced a technique called *selective symbolic execution*. This technique enables the exploration of only the paths that are important to the fuzzer and generates inputs for conditions that are challenging for the fuzzer to solve independently.

The approach employed by Driller is relatively straightforward. A symbolic execution engine is executed only if the fuzzer is unable to produce new code coverage for a given period of time. While this approach is effective to some extent, recent studies have revealed that the optimal approach is to generate inputs concurrently with the fuzzer. This concurrent input generation approach is currently the standard in modern hybrid fuzzers, including QSYM, SymQEMU, and Sydr.

QSym

QSym was the first "modern", binary-only hybrid fuzzer that showed significant improvements over previous approaches. In the paper *QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing* [36], the authors described the design and implementation of QSym. One particularly interesting aspect of QSym is that it uses pintool-based instrumentation to collect the necessary information for the symbolic execution engine. While this approach has some limitations (e.g. only x86 is supported), it allows for a much more efficient implementation because there is no need for the IR translation step.

In addition to using pintool-based instrumentation for efficiency, QSym also incorporates several novel techniques to improve the hybrid fuzzing process. One of these techniques is optimistic constraint solving, which allows QSym to combat the problem of over-constrained paths by solving only part of the constraints.

Another technique implemented in QSym is unrelated constraint elimination. This technique removes constraints that are deemed unrelated to the current path condition. This approach is particularly useful for reducing the number of constraints that need to be solved by the SMT solver, thus improving performance.

The combination of these techniques and the concurrent input generation approach has made the duo of AFL and QSym a highly effective hybrid fuzzer. Nevertheless, the progress in this field did not stop there, and several other approaches have been developed since then.

SymCC

SymCC presented a compilation-based approach to symbolic execution that aims to address the issue of speed, which has been a major hurdle to practical symbolic execution. In the paper *Symbolic execution with SymCC: Don't interpret, compile!* [21], the authors describe how they implemented SymCC, an LLVM-based C and C++ compiler that incorporates concolic execution into the binary. By integrating concolic execution into the resulting binary, SymCC can achieve much better performance than previous approaches, such as QSym.

While this method greatly improved the performance of symbolic execution, it also introduced some limitations. The most obvious one is that it requires recompiling the program with SymCC, which is not always possible. Nevertheless, the combination of SymCC and AFL has proven to be highly effective in practice.

SymQEMU

SymQEMU is another compilation-based symbolic execution tool for binaries, which is similar to SymCC. However, SymQEMU uses dynamic binary translation to instrument the binary, whereas SymCC uses LLVM-based compilation. This means that SymQEMU does not require recompiling the program like SymCC, making it more practical for use on pre-existing binaries.

In the paper *SymQEMU: Compilation-based symbolic execution for binaries* [22], the authors describe how they implemented SymQEMU and evaluated its performance as a hybrid fuzzer in combination with AFL. They found that SymQEMU, when combined with AFL, outperforms previous approaches to hybrid fuzzing, including SymCC + AFL and QEMU-based fuzzers such as Driller and QSYM.

Fuzzolic

At the same time, as SymQEMU paper was published, another paper *FUZZOLIC: Mixing Fuzzing and Concolic Execution* [7] was also published which proposed a hybrid fuzzer called Fuzzolic. Fuzzolic is built on top of the binary translator QEMU, offering significant benefits in terms of performance and versatility compared to the QSYM concolic executor. The authors also proposed an approximate solver called FUZZY-SAT [6], which borrows techniques from the fuzzing domain and provides an alternative to accurate but expensive SMT solving techniques.

Besides the approximate solver, Fuzzolic is also the first to change the classic architecture of concolic engines. Instead of implementing a tracer and the solver as a single component, Fuzzolic decouples them into two separate components. This allows Fuzzolic to overcome one of the major problems affecting QSYM, which is the inability to use external libraries such as SMT solvers due to limitations of recent releases of most dynamic binary translation frameworks. By separating the tracer and solver components into distinct processes, Fuzzolic can avoid this issue and ensure compatibility with future changes in DBT frameworks.

Currently, Fuzzolic is one of the most advanced hybrid fuzzers in terms of performance and versatility. However, it is still in its early stages of development. Nevertheless, it is a promising approach that could potentially become the state-of-the-art solution for hybrid fuzzing.

Sydr

Lastly, Sydr [32] is a binary-only dynamic symbolic execution (DSE) tool that employs dynamic binary instrumentation, like SymQEMU and Fuzzolic. However, unlike the SymQEMU or Fuzzolic it instruments the binary using DynamoRIO instead of QEMU. What is also notable is that Sydr separates the concrete executor (tracer) and the symbolic executor (solver) into two separate components, which is similar to Fuzzolic.

Sydr employs a range of state-of-the-art techniques to enhance the performance of the symbolic execution engine, such as path predicate slicing, optimistic constraint solving, non-symbolic instruction skipping, and other methods.

An interesting feature of Sydr is the integration of security invariant checking into its symbolic execution engine, known as "security predicates". This enables Sydr to identify possible security vulnerabilities in a program by checking for violations of security invariants during symbolic execution.

Sydr is one of the key components of the Sydr-fuzz [31] dynamic analysis tool that enables the combination of Sydr with other tools such as AFL++ or LibFuzzer, making hybrid fuzzing possible.

In this thesis, we will focus on the Sydr and Sydr-fuzz tools as targets for our proposed improvements.

3 PyTorch Fuzzing

PyTorch is a popular open-source machine-learning framework that has gained immense popularity in recent years. Developed by Meta (formerly Facebook), PyTorch has emerged as one of the most widely used machine learning frameworks due to its ease of use, flexibility, and dynamic computational graph, making it a popular choice for researchers and developers alike.

PyTorch is a critical component of many applications across various industries such as banking, healthcare, insurance, and many others. It is used for natural language processing, image classification, speech recognition, and other tasks. In banking, PyTorch is used to develop fraud detection systems, while in healthcare, it is used to diagnose diseases and predict patient outcomes. The insurance industry uses PyTorch to analyze risk and predict losses. The flexibility of PyTorch enables it to be used in many other domains as well. PyTorch has been used to build many state-of-the-art machine learning models and is a vital tool in the field of deep learning.

Despite its popularity and usefulness, PyTorch has several challenges that must be addressed to ensure its reliability and security. PyTorch has multiple dependencies, and it includes a considerable amount of C/C++ code which implies that it is susceptible to memory safety vulnerabilities. Moreover, PyTorch is an interesting target for adversaries since it is used in critical applications. Therefore, it is crucial to ensure that PyTorch is secure and free from vulnerabilities. Fuzzing is a valuable technique that can help identify bugs and vulnerabilities in PyTorch, making it more robust and secure. By fuzzing PyTorch, we can ensure that it can withstand attacks and continue to operate correctly in real-world applications.

In this chapter, we will explore the concept of PyTorch fuzzing and its significance in improving the reliability and security of PyTorch. We will begin by describing its attack surface. We will then develop fuzzing harnesses for the interesting parts of the codebase. Finally, we will describe the fuzzing methodology and the results of our work.

3.1 Attack Surface Mapping

To begin our fuzzing efforts, we must first identify the attack surface of PyTorch. The attack surface refers to all the entry points through which an attacker can potentially interact with the system and launch an attack. In the case of PyTorch, its attack surface includes various modules, libraries, and dependencies that it uses.

To identify interesting parts of the codebase that are relevant to the attack surface, we have performed manual code analysis. Our analysis has highlighted several modules that are particularly interesting to fuzz, including the model loading and RPC communications modules.

3.1.1 Model Loading

The process of loading pre-trained models is a crucial entry point that attackers can exploit to gain access to the system. This process is typically handled by the model loading module, which can be accessed via the `torch.load()` function.

During the loading process, the `torch.load()` function goes through several deserialization steps, also known as unpickling, to recreate the original object from the byte stream. Deserialization is a common source of vulnerabilities in many applications, as it can be difficult to implement correctly. Unfortunately, PyTorch is not immune to this issue. Additionally, since the implementation is written in C++, it is even more susceptible to memory safety vulnerabilities.

The code responsible for model loading and parsing is mostly located in these files:

- jit/serialization/import.cpp
- jit/ir/irparser.cpp
- jit/serialization/unpickler.cpp
- jit/runtime/interpreter.cpp
- jit/frontend/schema_type_parser.cpp

3.1.2 Remote Communications (RPC)

Besides model loading, PyTorch has another interesting mechanism that attackers can exploit - the RPC communications module.

The RPC module in PyTorch is a complex system that opens up new, remotely accessible attack vectors. PyTorch uses the RPC module to implement distributed training and inference that allows users to train and execute models across multiple machines. This feature is essential for large-scale applications that require high computational power. However, it increases the security risks by creating additional entry points for attackers.

PyTorch uses various types of RPCs such as `RRef` (Remote Reference), `ScriptCall`, and others to interact with remote machines. Before sending RPCs, they are serialized into pickled objects using the `torch::jit::pickle` function. The RPCs are then sent using different backends like TensorPipe, Gloo, and MPI. Once received, the RPCs are deserialized using the `torch::jit::unpickle` function, and the target `Message`'s class `fromMessage()` method is called. This leaves the receiver with a plain message that can be further processed.

Unfortunately, the complexity of this system makes it prone to bugs and vulnerabilities. Multiple serializations and deserializations of messages can introduce bugs, and the fact that the RPC protocol is implemented in C++ makes it an attractive target to look for memory safety vulnerabilities. Moreover, given the memory-unsafe nature of the RPC protocol, a single bug could potentially allow an attacker to execute code remotely on a target machine. As a result, fuzzing the RPC module is highly recommended to identify and address potential vulnerabilities.

With that in mind, we have identified the following files as the most interesting targets for security research:

- distributed/rpc/*.cpp
- jit/serialization/unpickler.cpp

3.1.3 Finding Fuzz-Targets

Now that we have identified different parts of the PyTorch codebase that are relevant to the attack surface, we can proceed to the second part of the attack surface mapping - identifying specific functions and methods to fuzz.

To achieve this, we have used two different approaches:

1 **Manual code review** - we have performed a manual code review of the PyTorch codebase to identify relevant functions that are confined to the defined attack surface.

2 **CodeQL** - we have used CodeQL [3] to broadly search for interesting functions and methods that perform some kind of deserialization or parsing.

The first approach is straightforward and does not require any additional tools. However, it is time-consuming and requires a lot of manual work. Nevertheless, it yields the best results since it allows us to precisely identify the functions that might be interesting to fuzz.

The second approach lacks "precision" but can be automated and scaled to a large codebase. It allows us to quickly identify a narrowed-down set of functions that are worth looking into. However, it is not as precise as the first approach since it relies on heuristics and does not "understand" the code. As a result, it can miss some relevant functions. Nevertheless, it is a good starting point for fuzzing since it can help identify interesting functions that can be further analyzed manually.

To begin with, we employed the second approach to pinpoint some specific functions that are worth looking into. We developed a CodeQL query **??** that searches for functions that have two parameters:

1 The first parameter is a pointer to data of "parsable" types. For example - `char*`, `byte*`, and others.

2 The second parameter is an integer that represents the size of the data. For example - `int`, `size_t`, and others.

With that in place, we added a few more heuristics to filter out irrelevant functions. Finally, we used *Cyclomatic Complexity* [12] to rank the results and identify the most complex functions. Some results of the query are shown in Table 3.1.

Complexity	**Function**
13	`rpc::parseWireSections`
6	`Unpickler::readSlowWithBuffer`
5	`TokenTrie::insert`
4	`rpc::wireDeserialize`

Table 3.1: CodeQL query results

These results gave us a good starting point. From here, we manually reviewed the functions and started to study the codebase, employing the first approach.

Finally, we have compiled a list of the most interesting functions that are worth fuzzing. The list is shown in Table 3.2.

Function
`jit::parseIR`
`jit::load`
`rpc::deserializeResponse`
`rpc::deserializeRequest`

Table 3.2: Fuzz targets

The first two functions are related to the JIT module and are responsible for parsing and loading the pickled data. Some examples of such data are: `saved models`, `serialized request`, and others.

The last two functions are related to the RPC module and are responsible for deserializing RPC requests and responses. These functions are interesting because they are directly processing untrusted data that is received from the network.

Besides that, we have also identified another interesting function - `jit::preoptimizeGraph`, which is a part of the JIT module. We decided to target it with differential fuzzing to find bugs related to the optimization of the JIT graphs.

3.2 Preparing PyTorch for Fuzzing

3.2.1 Fuzzing Harness Development

With the fuzz targets identified, we can proceed to the next step - developing a fuzzing harness. The goal of the fuzzing harness is to provide a way to feed the fuzz target with data and collect the results of the execution. For our purposes, we have developed a *LibFuzzer*-compatible [27] targets for each of the functions listed in Table 3.2.

Each libFuzzer-based target shares the same structure. The structure is shown in Listing 7.

```
int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    // 1. Parse the input data
    // 2. Call the fuzz target with the parsed data
    // 3. Return 0
}
```

Listing 7: Fuzz target structure

The goal of a generic fuzzing harness is to pass the data generated by the fuzzer to the fuzz target and clean up the resources after the execution is finished. The fuzzing harness is also responsible for handling the exceptions that might be thrown by the fuzz target.

The fuzzing harnesses that we have developed for PyTorch are similar to the generic one. However, they also perform some additional steps. For example, they handle some PyTorch-specific exceptions.

The developed fuzzing harnesses are listed below:

- `irparser_fuzz.cc` - fuzzes `jit::parseIR`
- `load_fuzz.cc` - fuzzes `jit::load`
- `message_deserialize_fuzz.cc` - fuzzes `rpc::deserializeResponse` and `rpc::deserializeRequest`
- `jit_differential_fuzz.cc` - fuzzes three related methods - `jit::parseIR`, `jit::preoptimizeGraph`, and `jit::InterpreterState(code).run()` with differential fuzzing

However, the development of fuzzing harnesses is only one more step on the road toward hybrid fuzzing. The next step is to compile all the necessary build targets and prepare the environment.

3.2.2 Build Targets Compilation

To meet the requirements of hybrid fuzzing, we need to prepare a few different types of build targets.

LibFuzzer Target

The first one is the fuzz target itself. It is the main entry point for the fuzzer. This target is compiled using the *clang* compiler with the *libFuzzer* library linked. To achieve this, we have added the following compilation flags to the build script: `-fPIC -g -fsanitize=fuzzer,address,bounds`. To easily distinguish the fuzz target from other targets, we have also added the `_fuzz` suffix to the name of the produced binary.

Sydr Target

The second type of build target is the plain binary, which reads the data directly from the file and passes it to the fuzz target, without involving the fuzzer. It allows the symbolic execution engine to execute the fuzz target with concrete data. This target is also compiled with the help *clang* compiler, however, the flags are different: `-fPIC -g`. We have also added the `_sydr` suffix to the name of the binary.

Coverage Target

And the last type of build target is the binary that is used to collect the coverage information. It is compiled with the *clang* compiler and the following flags: `-fPIC -g -fprofile-instr-generate -fcoverage-mapping`. As for the previous targets, we have also added the `_cov` suffix to the name of the binary.

3.3 PyTorch Hybrid Fuzzing

With all the artifacts produced, we can now proceed to the next step - hybrid fuzzing. The goal of this step is to prepare the environment, perform the fuzzing, and analyze the results. To perform hybrid fuzzing, we have used the *sydr-fuzz*, a tool developed by ISP RAS. *Sydr-fuzz* is a flexible framework for hybrid fuzzing, which allows us to easily perform the fuzzing of the target program with the help of the symbolic execution engine and state-of-the-art fuzzers such as *AFL++* and *libFuzzer*.

3.3.1 Preparing the Environment

Before we can start the fuzzing, we need to prepare the environment. The preparation includes many steps, however, in our case, we only need to perform a single step - to generate the corpus for the fuzzer. The corpus is a set of inputs that are used by the fuzzer to generate new inputs.

As most of our targets are aimed at fuzzing the unpickling functionality, we have collected the corpus by extracting test models from the PyTorch repository. Besides unpickling, we also had a few targets that were aimed at fuzzing the IR-parsing functionality. For these targets, we have searched for tests that use the `jit::parseIR` method and extracted the corresponding IRs. An example of such a testcase is shown in Listing 8.

```
graph(%a : Tensor):
      %b : Tensor = aten::mul(%a, %a)
      %c : Tensor = aten::mul(%b, %b)
      %d : Tensor = aten::mul(%c, %c)
      %c_size : int[] = aten::size(%c)
      %c_alias : Tensor = aten::view(%c, %c_size)
      %e : Tensor = aten::mul(%b, %d)
      %f : Tensor = aten::mul(%c_alias, %c_alias)
      %output : Tensor = aten::mul(%e, %f)
      return (%output)
```

Listing 8: IR program extracted from the PyTorch repository

3.3.2 Dynamic Analysis Pipeline

With everything prepared, we can now proceed to the fuzzing itself. The fuzzing is performed in a form of a dynamic analysis pipeline, which is illustrated in Diagram 3.1.

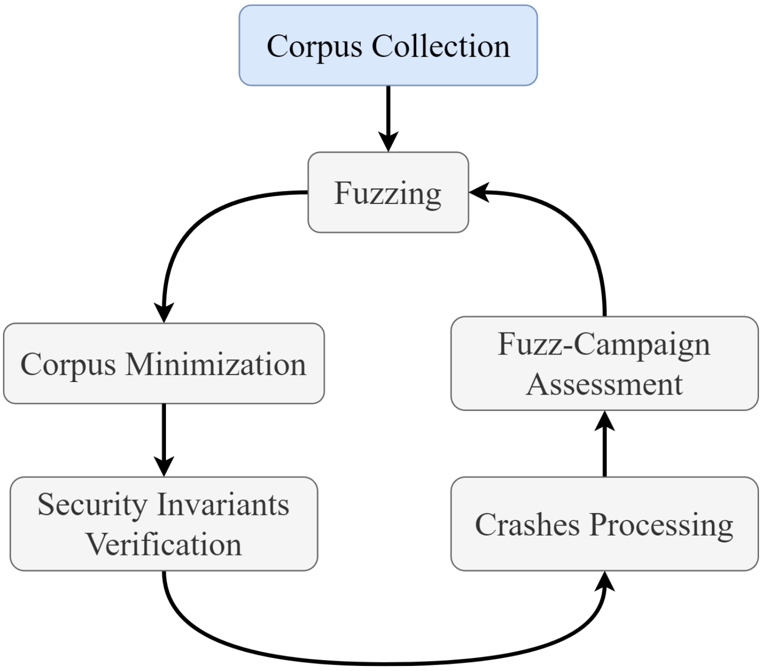

Figure 3.1: Dynamic analysis pipeline

Hybrid Fuzzing

The first step of the pipeline is hybrid fuzzing. It is performed with the help of the *sydr-fuzz* tool. The tool takes a toml configuration file as input, which specifies the targets to be fuzzed and other parameters. For example, the configuration file for the `jit_differential` target is shown in Listing 9.

This configuration file contains three sections: `sydr`, `libfuzzer`, and `cov`. The `sydr` section contains information about the target program and how to execute it under the symbolic execution engine. The `libfuzzer` section contains libfuzzer-related information, such as the path to the fuzz target binary and the arguments that should be passed to it. Finally, the `cov` section contains information about the coverage target.

With the configuration file prepared, we can now start the fuzzing campaign. To do so, we need to execute the *sydr-fuzz* binary as follows: `sydr-fuzz -c`

```
[sydr]
args = "-l debug"
target = "/jit_differential_sydr @@"

[libfuzzer]
path = "/jit_differential_fuzz"
args = "-detect_leaks=0 -rss_limit_mb=30720
↪ -timeout=300 -report_slow_units=350
↪ /ir_corpus"

[cov]
target = "/jit_differential_cov @@"
```

Listing 9: Configuration file for the `jit_differential` target

`jit_differential.toml run`. The command will start the fuzzing process and will print the results to the console.

Corpus Minimization

After running the fuzzer for some time, we can proceed to the next step of the pipeline - corpus minimization. The goal of this step is to reduce the size of the corpus by removing the redundant inputs. The corpus minimization is performed using the *sydr-fuzz* tool in the *cmin* mode.

This step is optional, however, it is highly recommended, because it greatly improves the performance of the next step - security invariants verification. Without corpus minimization, the security invariants verification is less likely to find any bugs, as it will potentially re-execute almost the same inputs multiple times.

Security Invariants Verification

The next step of the pipeline is security invariants verification. The idea behind this step is to execute the *sydr* symbolic execution engine in the *security* mode, with inputs from the minimized corpus. The *security* mode is a special mode that is designed to check for violation of various security invariants, such as integer overflows, null pointer dereferences, buffer overflows, and others. If any of the invariants are violated, the symbolic execution engine will report the corresponding bug.

Running the symbolic execution engine in the *security* mode is a very expensive operation. For this reason, we have worked on the optimization of this step. The results are discussed in chapter 4.2.

Crashes Processing

Near the end, we have the crashes processing stage. During previous steps, *sydr-fuzz* may have found some crashes. At this stage, those crashes need to be processed. To do so, *sydr-fuzz* employs the *casr* tool [24]. This tool performs automatic crashes processing, which includes deduplication, and clusterization.

Executing *sydr-fuzz* in this mode, using the following command: `sydr-fuzz -c jit_differential.toml casr`, will produce a set processed crashes.

Fuzz-Campaign Assessment

Finally, after all the steps are completed, we can assess the results of the fuzzing campaign. The assessment is performed manually by the user, and includes the following steps:

1 Bug triaging and reporting

2 Code coverage analysis

During the bug triaging and reporting step, the user needs to analyze the found bugs and decide whether they are real bugs or not. If the bug is proven to be real, the user needs to report it to the developers of the target program.

The code coverage analysis step is performed to assess the effectiveness of the fuzzing campaign. The goal of this step is to help the user to understand whether the fuzzing campaign was effective and whether it is worth continuing it. The code coverage analysis is performed using the *sydr-fuzz* tool in the *cov* mode.

This concludes the description of the dynamic analysis pipeline, which has been used to fuzz the PyTorch framework. In the next section, we will discuss some of the findings.

3.4 Results Overview

As a result of our work multiple bugs were found in different parts of the PyTorch framework. Most of the bugs were found in the module responsible for unpickling, however, we have also identified at least one remotely-accessible bug in the RPC module.

We reported all discovered bugs to the PyTorch framework developers, who confirmed their validity and subsequently merged the corresponding pull requests into the PyTorch repository. The pull requests related to the bugs are listed below:

- #94300: Add size check before calling stack_.at(dict_pos) in unpickler.cpp
- #94298: Add stack emptiness checks inside interpreter.cpp
- #94297: Add size check before calling .back() in rpc/script_call.cpp
- #94295: Add exception handlers for stoll in schema_type_parser.cpp
- #91401: Add out-of-bounds checks inside irparser.cpp and unpickler.cpp

In section 5.1, we will take a closer look at the bugs we found and the pull requests we created.

4 Hybrid Fuzzer Improvements

During the fuzzing of the PyTorch, we identified several drawbacks of our hybrid fuzzing solution. In this section, we aim to address these issues to improve the overall performance of the *sydr-fuzz*.

4.1 Scheduling Symbolic Pointers Modeling

Almost any program written in C/C++ uses pointer operations, thus if we want to have a precise symbolic model of the program, we need to model pointers as well. However, as we have already discussed in section 2.3.5, modeling symbolic pointers is a very computationally expensive process. Nevertheless, it is required to find new paths in the program, that would otherwise be unreachable.

The same applies to PyTorch. It uses pointers extensively, and thus we need to model them to achieve maximum testing completeness. In fact, we have noticed that symbolic pointer modeling is required for almost any project under test.

4.1.1 Problem Statement

With that being said, we can not simply disable symbolic pointer modeling, as it would prevent us from finding new paths in the program, and thus degrade the overall testing completeness. On the other hand, we can not enable it all the time, as it would significantly slow down the fuzzing process.

So, we would like to develop a scheduling strategy, that would allow us to achieve the maximum possible code coverage, without sacrificing performance.

A scheduling strategy in this context is a mechanism that should allow us to run sydr with memory modeling enabled only when we would benefit from it the most.

4.1.2 Proposed Solution

To do so we have implemented a simple, yet effective scheduling strategy, which executes sydr with memory modeling enabled once in N runs.

This strategy "exploits" how caching mechanism works for already traversed branches. By allowing sydr to execute multiple times before running it with memory modeling enabled, we allow the cache to saturate with the branches that are not dependent on the symbolic pointers. This way, when we finally run sydr with memory modeling enabled, it will be able to reuse most of the results from previous runs, having to perform the expensive symbolic pointers modeling only for a small portion of new branches. This significantly improves the performance of the symbolic pointer modeling, allowing us to find new paths in the program, while still maintaining a reasonable execution time.

4.1.3 Experimental Evaluation

After we have implemented the scheduling strategy, we had to find the optimal value of N. The value of N should be large enough to not significantly degrade the overall performance of the hybrid fuzzer, and at the same time allow the cache to saturate. But it should not be too large, as it would prevent us from finding new paths in the program.

To find the best value of N, we have conducted multiple experiments using the *FuzzBench* benchmarking platform [17]. The results of these experiments are discussed in section 5.2.

Based on the results of these experiments, we have decided to use N=25 as the default value for the scheduling strategy implemented in *sydr-fuzz*.

We have also introduced a new configuration parameter called `symaddr`, which enables users to customize the number of runs performed between symbolic pointer modeling runs. This parameter can be used to fine-tune the performance of the *sydr-fuzz* for a specific target.

4.2 Enhancing Security Invariants Checking Mechanism

The next issue we have identified during the fuzzing of the PyTorch is related to the security predicates mechanism. As we have mentioned in section 2.4.1, *sydr* has a mechanism that allows it to check for various security invariants violations, such as buffer overflows, null pointer dereferences, integer overflows, and others. Unfortunately, security predicates often tend to produce false positives. That is why *sydr-fuzz* implements automatic verification of security predicates results.

4.2.1 Security Predicates - Violations Verification

To verify that the bug is real, *sydr-fuzz* executes the target program built with sanitizers, on the input generated by *security predicates* mechanism. When a sanitizer reports an error in the same location that was identified as the error source by security predicates, the bug is considered to be verified. In other words, if the error is detected in the expected location, it confirms that the security predicates accurately captured the potential vulnerability and that the input seed triggered the vulnerability.

To perform the validation, *sydr-fuzz* needs to symbolize *sydr*'s log file, which contains the information about the location of the error. Originally, *sydr-fuzz* used a Python script that relied on *addr2line* and *objdump* tools to perform this operation. However, we have found that this step can be time-consuming and negatively impact the overall performance of the hybrid fuzzer. That is why we have decided to optimize the performance of this step by reimplementing it in Rust. This allowed us to significantly improve the performance of the verification process.

4.2.2 Utilizing Debug Information to Improve Annotation Speed

Before we could start rewriting the annotation script, we had to understand how the debug information is stored and how we could work with it from Rust.

DWARF Debug Information Format

The DWARF debugging information format is a standard way to store debugging information in object files. It is used by many compilers and debuggers, including *GCC*, *Clang*, and others. The DWARF format is designed to be extensible, efficient, and language-independent. It uses a series of data structures to describe the program in a way that is both human-readable and machine-readable. Debug information can be used to partially reconstruct the source code, even when only the binary file is available. By leveraging this information, it is possible to recover variable names, file names, function names, and other important entities that were present in the original code.

That is precisely the information that is required for the annotation process. With its help, we can recover the function names and line numbers from the addresses listed in the *sydr*'s log file. This allows us to compare the output of the sanitized binary with the information from the *sydr*'s log file, and thus verify the violation.

Rust Implementation

To implement the annotation process in Rust, we have developed a *dbginfo* module, which is responsible for parsing the DWARF debug information using the *gimli* and *addr2line* crates. The *dbginfo* is then used by the *annotate* module, which is responsible for the annotation process itself.

To make the annotation process efficient, we have implemented an in-memory caching mechanism that avoids parsing the debug information for the same binary multiple times. This caching approach significantly improves the performance of the annotation process by reusing parsed debug information across multiple calls to the `annotate_log(log_path, annotated_path)` function.

4.2.3 Experimental Evaluation

After we have implemented a new solution we executed multiple benchmarks to evaluate its performance. In Section 5.3, we discuss the findings of these experiments and provide insights into the effectiveness and efficiency of our approach.

Our measurements showed that the annotation process for *branch traces* was up to 99.95% faster, while the annotation speed for *instruction traces* showed a comparable improvement of up to 99.13%. These performance gains demonstrate the effectiveness of our optimization approach in accelerating the analysis of both types of traces.

5 Results

5.1 PyTorch Bugs

Based on the findings during this project we've applied to talk at the Positive Hack Days (PHD) 2023 conference. The talk was accepted, and we presented our findings at the conference.

5.1.1 #91999: Case study of a bug in PyTorch

5.1.2 #91999: Case study of a bug in PyTorch

5.2 Experimental Results for the Scheduling Strategy

To assess the quality of proposed changes we have conducted multiple experiments.

The first one: symptr-15-1

	sydr_aflplusplus	sydr_symptr
FuzzerMedian	94.81	95.18
FuzzerMean	94.69	94.67
freetype2-2017	94.81	95.18
harfbuzz-1.3.2	93.91	94.20
lcms-2017-03-21	94.81	94.52
libjpeg-turbo-07-2017	98.40	97.49
libpng-1.2.56	98.41	98.32
openthread-2019-12-23	85.04	84.94
sqlite3_ossfuzz	97.48	98.01

Figure 5.1: Median relative code coverage on each benchmark (N=15)

The second one: symptr-25-1

By avg. score

	average normalized score
fuzzer	
sydr_aflplusplus	99.82
sydr_symptr	99.80

By avg. rank

	average rank
fuzzer	
sydr_aflplusplus	1.43
sydr_symptr	1.57

Figure 5.2: Average score and average rank (N=15)

	sydr_symptr	sydr_aflplusplus
FuzzerMedian	95.10	93.68
FuzzerMean	95.96	93.82
freetype2-2017	93.71	93.59
harfbuzz-1.3.2	93.78	93.68
lcms-2017-03-21	95.10	93.92
libjpeg-turbo-07-2017	98.30	97.85
libpng-1.2.56	98.24	98.02
openthread-2019-12-23	98.11	86.07
sqlite3_ossfuzz	94.44	93.65

Figure 5.3: Median relative code coverage on each benchmark (N=25)

And, finally, the third one: symptr-25-vs-35-2

5.3 Annotate

By avg. score

	average normalized score
fuzzer	
sydr_symptr	100.00
sydr_aflplusplus	97.82

By avg. rank

	average rank
fuzzer	
sydr_symptr	1.0
sydr_aflplusplus	2.0

Figure 5.4: Average score and average rank (N=25)

	sydr_symptr	sydr_alter_symptr
FuzzerMedian	95.66	94.62
FuzzerMean	94.80	94.40
freetype2-2017	95.01	94.14
harfbuzz-1.3.2	94.38	94.43
lcms-2017-03-21	94.60	94.81
libjpeg-turbo-07-2017	97.68	98.85
libpng-1.2.56	97.88	97.66
openthread-2019-12-23	83.18	83.86
re2-2014-12-09	99.38	99.32
sqlite3_ossfuzz	96.31	92.14

Figure 5.5: Median relative code coverage on each benchmark (N=25 vs N=35)

By avg. score

	average normalized score
fuzzer	
sydr_symptr	99.72
sydr_alter_symptr	99.31

By avg. rank

	average rank
fuzzer	
sydr_alter_symptr	1.5
sydr_symptr	1.5

Figure 5.6: Average score and average rank (N=25 vs N=35)

Benchmark	Default (ms)		Rust (ms)		Performance diff
	Mean	Std	Mean	Std	
cjson	292.25	±66.1337	1.08	±0.06	-99.63 %
libjpeg	931.96	±55.3113	1.75	±0.19	-99.81 %
libpng	924.03	±567.277	1.64	±0.26	-99.82 %
libxml2	38308.54	±796.162	12.27	±0.33	-99.97 %
minigzip	14567.67	±121.165	7.51	±0.48	-99.94 %
pcre2	14562.59	±1419.99	7.26	±0.58	-99.95 %
readelf	19486.86	±1015.72	7.02	±0.36	-99.96 %
rizin	31061.09	±1800.72	12.77	±0.22	-99.96 %
yices	8936.02	±111.98	3.98	±0.3	-99.96 %
yodl	16907.58	±377.37	7.97	±0.31	-99.95 %

Table 5.1: Mean execution time to annotate 10 branch traces with the default and Rust annotator.

Benchmark	Size (mb)	Default (s)		Rust (s)		Performance diff
		Mean	Std	Mean	Std	
cjson	1.21	20.74	±0.05	0.23	±0.005	-98.87 %
libjpeg	6.69	176.9	±0.60	2.32	±0.027	-98.69 %
libpng	2.08	57.6	±0.36	0.75	±0.019	-98.69 %
libxml2	50.3	timeout	timeout	21.27	±1.016	XXX %
minigzip	0.75	timeout	timeout	166.59	±1.863	XXX %
pcre2	6.62	172.77	±0.69	2.8	±0.048	-98.38 %
readelf	9.19	454.11	±40.68	3.94	±0.144	-99.13 %
rizin	34.04	timeout	timeout	15.15	±0.925	XXX %
yices	10.48	294	±1.1	4.69	±0.123	-98.4 %
yodl	22.53	timeout	timeout	10.3	±1.014	XXX %

Table 5.2: Mean execution time (over 3 runs) to annotate instruction trace with the default and Rust annotator.

6 Conclusion

6.1 Summary

6.2 Future Work

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. Pavel Avgustinov et al. “QL: Object-oriented Queries on Relational Data”. en. In: *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany* (2016). DOI: 10.4230/LIPICS.ECOOP.2016.2. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6096/.

4. 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.

5. 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.

6. Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. “Fuzzing Symbolic Expressions”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 711–722. DOI: 10.1109/ICSE43902.2021.00071.

7. Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. “FUZZOLIC: Mixing Fuzzing and Concolic Execution”. In: *Comput. Secur.* 108.C (2021). ISSN: 0167-4048. DOI: 10.1016/j.cose.2021.102368. URL: https://doi.org/10.1016/j.cose.2021.102368.

8. Derek L. Bruening and Saman Amarasinghe. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". AAI0807735. PhD thesis. USA, 2004.

9. *Chromium project memory safety report.* https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed: 2023-02-12.

10. Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

11. 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.

12. G.K. Gill and C.F. Kemerer. "Cyclomatic complexity density and software maintenance productivity". In: *IEEE Transactions on Software Engineering* 17.12 (1991), pp. 1284–1288. DOI: 10.1109/32.106988.

13. Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft." In: *Queue* 10.1 (2012), 20–27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. URL: https://doi.org/10.1145/2090147.2094081.

14. 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.

15. 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.

16. Valentin J. M. Manes et al. *The Art, Science, and Engineering of Fuzzing: A Survey*. 2019. arXiv: 1812.00140 [cs.CR].
17. Jonathan Metzman et al. "FuzzBench: An Open Fuzzer Benchmarking Platform and Service". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, 1393–1403. ISBN: 9781450385626. DOI: 10.1145/3468264.3473932. URL: https://doi.org/10.1145/3468264.3473932.
18. 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.
19. 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.
20. 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.
21. Sebastian Poeplau and Aurélien Francillon. "Symbolic execution with SymCC: Don't interpret, compile!" In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

22. Sebastian Poeplau and Aurélien Francillon. "SymQEMU: Compilation-based symbolic execution for binaries". In: *Proceedings 2021 Network and Distributed System Security Symposium* (2021).

23. 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.

24. Georgy Savidov and Andrey Fedotov. "Casr-Cluster: Crash Clustering for Linux Applications". In: *2021 Ivannikov Ispras Open Conference (ISPRAS)*. 2021, pp. 47–51. DOI: 10.1109/ISPRAS53967.2021.00012.

25. 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.

26. 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.

27. Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 157–157. DOI: 10.1109/SecDev.2016.043.

28. 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.

29. 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.

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

31. Alexey Vishnyakov et al. "Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle". In: *2022 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 2022. DOI: 10.1109/ispras57371.2022.10076861. URL: https://doi.org/10.1109%2Fispras57371.2022.10076861.

32. 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.

33. Alexey V. Vishnyakov et al. "Symbolic Security Predicates: Hunt Program Weaknesses". In: *CoRR* abs/2111.05770 (2021). arXiv: 2111.05770. URL: https://arxiv.org/abs/2111.05770.

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

35. *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.

36. 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.