Записки научных
семинаров ПОМИ
Том 530, 2023 г.

I. Yegorov, E. Kobrin, D. Parygina, A. Vishnyakov, A. Fedotov

PYTHON FUZZING FOR TRUSTWORTHY MACHINE LEARNING FRAMEWORKS

Abstract. Ensuring the security and reliability of machine learning frameworks is crucial for building trustworthy AI-based systems. Fuzzing, a popular technique in the secure software development lifecycle (SSDLC), can be used to develop secure and robust software. Popular machine learning frameworks such as PyTorch and TensorFlow are complex and written in multiple programming languages including C/C++ and Python. We propose a dynamic analysis pipeline for Python projects using the Sydr-Fuzz toolset. Our pipeline includes fuzzing, corpus minimization, crash triaging, and coverage collection. Crash triaging and severity estimation are important steps to ensure that the most critical vulnerabilities are addressed promptly. Furthermore, the proposed pipeline is integrated in GitLab CI. To identify the most vulnerable parts of the machine learning frameworks, we analyze their potential attack surfaces and develop fuzz targets for PyTorch, TensorFlow, and related projects such as h5py. Applying our dynamic analysis pipeline to these targets, we were able to discover 3 new bugs and propose fixes for them.

§1. Introduction

Artificial Intelligence (AI) has been the focus of significant attention in recent years due to its transformative potential across a wide range of industries. However, with the increasing prevalence of AI-based systems, security concerns have also emerged. Ensuring the reliability and safety of these systems is paramount, and the field of secure AI has emerged as a multi-disciplinary area of computer science dedicated to achieving this goal.

AI-based systems are typically built on top of machine learning (ML) frameworks such as TensorFlow [23] and PyTorch [19], which provide the

Key words and phrases: fuzzing, trustworthy AI, machine learning framework, TensorFlow, PyTorch, Python, artificial intelligence, crash triage, dynamic analysis, secure software development lifecycle, SSDLC, computer security.

necessary tools to implement and train machine learning models. The growing complexity and pervasiveness of these frameworks have made them a prime target for attackers seeking to exploit vulnerabilities for malicious purposes. The potential consequences of a successful attack on an AI-based system are significant, ranging from compromising the integrity and confidentiality of sensitive data to causing physical harm or financial loss.

Developing secure and trustworthy ML frameworks is a challenging task given the large and complex codebase of popular ML frameworks, which are developed in multiple programming languages such as C/C++ and Python. Secure software development life cycle (SSDLC) is commonly used to provide safety and code quality for open source and commercial projects. Dynamic analysis, such as fuzzing, is often applied in SSDLC.

Fuzzing is a method for discovering software vulnerabilities by feeding unexpected or malformed seeds to a program. By applying fuzzing to ML frameworks, it is possible to identify security vulnerabilities that may not be found through traditional testing methods. However, the effectiveness of fuzzing depends on several factors, such as the quality of the initial test cases, the coverage of the code, and the ability to reproduce and isolate the bugs that are discovered. Code coverage quality is also tied with fuzz targets development. During fuzz targets development the attack surface should be considered, which has some peculiarities for ML frameworks. Furthermore, after fuzzing, the resulting bugs need to be triaged and prioritized to ensure that the most critical vulnerabilities are addressed first.

In this work we make the following contributions:

- we implement a dynamic analysis pipeline (fuzzing, corpus minimization, crash triaging, and coverage collection) for Pythonic projects in the Sydr-Fuzz [33] tool;
- using the proposed dynamic analysis pipeline for Python, we have found 3 new bugs in ML frameworks and related projects.

The paper is organized as follows. In Section 2 we discuss Python fuzzing tools and approaches, ML framework specific fuzzer FreeFuzz [35], and current fuzz targets for TensorFlow [23] and PyTorch [19] projects. In Section 3 we describe proposed dynamic analysis pipeline for Python projects. Section 4 is dedicated to attack surface analysis of ML frameworks. Section 5 demonstrates our results of fuzzing ML frameworks such as TensorFlow, PyTorch, and related projects. Section 6 concludes the paper.

§2. Related work

2.1. Atheris. Atheris [1] is a state-of-the-art fuzzing engine designed to guide the testing process through code coverage analysis. Based on libFuzzer [31], Atheris supports fuzzing of both Python code and native extensions for CPython. When it comes to native code, Atheris can be used in conjunction with Address Sanitizer [30] (ASAN) or Undefined Behavior Sanitizer (UBSAN) [8] to uncover additional bugs. A failure criterion is triggered by an uncaught exception thrown from the Python code or an abort caused by the sanitizers.

To collect Python coverage, Atheris provides three different bytecode instrumentation options. The first option is to instrument the libraries that are imported. The second option is to instrument specific individual functions. The third option is to instrument every function currently loaded in the interpreter. The coverage information provided by Atheris is compatible with the popular *coverage.py* [3] utility. Coverage reports are generated when the fuzzer exits due to a Python exception, `sys.exit()`, or after the specified number of runs has been reached.

Fuzzing native extensions (C/C++) with Atheris is similar to fuzzing Python code [26]. The main challenge is instrumenting the extensions to perform effective analysis Moreover, Atheris supports custom mutators for structure-aware fuzzing. Additionally, Atheris implements `FuzzedDataProvider` [6], which facilitates the translation of input data from simple bytes to other simple-structured forms, such as characters, Unicode symbols, strings, integers, lists, and more. Atheris is fully supported by OSS-Fuzz [13, 32].

2.2. FreeFuzz. FreeFuzz [35] is a novel approach to deep learning (DL) library testing via their public APIs. This technique automates API-level fuzz testing and provides a general and systematic approach to test DL libraries using open source examples of API application.

Prior approaches [28, 34] to testing DL libraries relied on pre-existing models as input test cases, which could apply only a few model-level mutations. In contrast, FreeFuzz constructs a significantly large input space by collecting code snippets from library documentation, library developer tests, and various DL models in the wild. The collected code is then instrumented to trace dynamic execution information. Based on the traced data, FreeFuzz constructs the type space, API value space, and argument value space for the subsequent fuzzing stage.

During the main analysis stage, FreeFuzz performs mutation-based fuzzing of the target library using several mutation rules for type and value mutations. These rules allow to cover test cases for many API functions with a large number of parameter values and combinations.

To resolve the test oracle problem, FreeFuzz utilizes three techniques. First, it runs on different hardware configurations such as CPU, GPU without CuDNN, and GPU with CuDNN to detect wrong-computation results. Second, it employs metamorphic relations based on types precision to detect performance bugs. Finally, it filters incorrect program terminations to distinguish crashes from invalid seeds obtained as a result of numerous mutations.

2.3. C++ Fuzz Targets for Tensorflow and PyTorch. Fuzz testing is an important technique to ensure the reliability of AI frameworks. Some fuzz targets have already been presented for the C++ side of APIs in various frameworks, such as TensorFlow [23]. While TensorFlow had its own simple fuzz targets, they only covered a limited amount of code and were not very promising for fuzzing like Base64 encoding functions, functions for path processing, functions for checking some internal structures, etc. These fuzz targets were already added to OSS-Fuzz [13]. However, TensorFlow also had some interesting fuzz targets that covered more code, including functions for decoding and encoding different file formats such as images and audio. Unfortunately, these targets were not added to OSS-Fuzz due to some compilation problems. Nevertheless, we were able to build them and added them to our repository, OSS-Sydr-Fuzz [15], which is a repository for hybrid fuzzing of open source software with our own tool, Sydr-Fuzz [33]. As a result of hybrid fuzzing these targets, we found an error in the code responsible for decoding WAV file format [4].

On the other hand, PyTorch [19] did not contain any fuzz targets, and none were added to OSS-Fuzz [13]. To address this issue, we wrote our own C++ fuzz targets that covered code responsible for parsing PyTorch intermediate representation, message deserialization, loading JIT modules, and more. We also wrote fuzz targets for torchvision, a PyTorch subproject for processing audio and video data [25]. Our fuzz testing efforts on PyTorch and torchvision C++ fuzz targets revealed many errors in PyTorch, torchvision, and some of their dependencies. All Sydr-Fuzz trophies, including errors in PyTorch, can be found in the OSS-Sydr-Fuzz trophy list [22].

2.4. Hypothesis. Hypothesis [11] is a Python library designed to automate code testing. It utilizes modern property-based testing, which tests target code on a range of input classes instead of concrete inputs.

With simple unit tests, software is typically tested on a limited number of test cases based on the programmer's imagination. Hypothesis simplifies the process of writing unit tests and makes them more powerful by identifying edge cases for test code that may not be reached through human efforts.

Using Hypothesis to test code does not require significant modifications to the unit test code. Instead, one simply needs to describe the structure of function arguments in a way that is compatible with Hypothesis and set constraints on them (e.g. testing code that only works with positive integer values). The test can then be run as before, with Hypothesis generating a set of inputs based on the given input structure and constraints. This feature enables the discovery of corner cases that can cause the tested code to fail or behave incorrectly.

Hypothesis tests are easy to turn into fuzz targets. Along with its ability to describe the structure, Hypothesis becomes a handy structure-aware fuzzing tool.

§3. Python dynamic analysis pipeline

Sydr-Fuzz [33] is a powerful tool that offers a comprehensive pipeline for dynamic analysis of Python code. This pipeline includes fuzzing, corpus minimization, coverage collection, and crash triaging. Each component of the pipeline is implemented as a separate command, ensuring that users can easily access the functionalities they need.

3.1. Fuzzing and Corpus Minimization. Dynamic analysis is a critical component of software testing, and fuzzing is one of the most popular methods used to carry it out. Fuzzing involves generating a large number of inputs to a program and monitoring its behavior for unexpected crashes or other errors. Sydr-Fuzz implements fuzzing via `sydr-fuzz run` command and uses Atheris [1] fuzzing engine, which is a coverage-guided Python fuzzing engine that supports native extensions.

Before the fuzzing session can begin, seed corpus minimization is performed to ensure that the fuzzer initializes faster and easier with a smaller corpus. Once the minimization is complete, the fuzzing session is launched. It continues until the coverage stops growing for some specified period or a

predetermined number of crashes are discovered. The `exit-on-time` and `jobs` options are responsible for managing these behaviors.

After the fuzzing session is complete, corpus minimization is the next step. This step is crucial because the time required for all subsequent steps depends on the size of the corpus. As such, the corpus must always be minimized to ensure optimal performance. Additionally, minimizing the corpus may be beneficial for reuse in subsequent launches. The `sydr-fuzz cmin` command provides this functionality.

3.2. Coverage Collection. Corpus coverage is a widely accepted and fundamental metric in the context of fuzzing. Sydr-Fuzz provides the `sydr-fuzz pycov` command, which utilizes *coverage.py* [3] to collect coverage information. The *pycov* command offers a range of coverage visualization formats, including report (in the specialized coverage.py format), html, xml, json, and lcov. While lcov is a popular choice for coverage visualization, it cannot be used with Python egg-files due to compatibility issues with the *genhtml* [7] tool. Therefore, the html format is commonly used for Python coverage visualization.

3.3. Crash Triaging. Crash triaging is an essential step in the dynamic analysis pipeline. When it comes to fuzzing, the number of crashes can be significant. Manually analyzing each crash and identifying those that represent the same error is a laborious and time-consuming task. To overcome this challenge, Sydr-Fuzz introduces a *casr* command that leverages the Casr [2] toolset to automate crash reports collection, triaging, and severity estimation. The `sydr-fuzz casr` command has the following main stages.

(1) The *casr-python* tool executes a fuzz target on all crashes and generates crash reports. In case of discovering a crash in native code, the *casr-san* helps generate the report. The *casr-python* and *casr-san* utilize the Python crash report and sanitizer report, respectively. The resulting report includes valuable information such as:
   - PythonReport – original Python crash report,
   - CrashSeverity – severity estimation (exploitable, probably exploitable, and not exploitable),
   - Stacktrace (Traceback),
   - Crashline – source code line of crash,
   - Source – source part containing the crash line,
   - and other details.

(2) The *casr-cluster* tool performs a deduplication algorithm on the *casr-python* reports, primarily based on the stack traces. The algorithm filters out standard library functions, fuzzer and sanitizer functions, exception handlers, and other utility functions. The crashes are considered identical if their stack traces are identical after filtering.

(3) The *casr-cluster* tool clusters the deduplicated reports using two pseudo-metrics: TopDist and RelDist [29]. TopDist measures the minimal position offset of the current frame relative to the topmost one, while RelDist calculates the distance between matched frames in two stack traces. These pseudo-metrics are used to determine the difference between two stack traces (*similarity*). Clustering is carried out according to formula

$$CLdist(CL_i, CL_j) = \underset{a \in CL_i, b \in CL_j}{max}(dist(a, b)), \tag{1}$$

where $dist(a, b) = 1 - similarity(a, b)$, CL_i and CL_j – different classes.

The crash triaging process generates a summary of clusters, with each cluster summary detailing the number of crashes it contains and a brief description of each crash, including error information (Python exception or sanitizer error) and crash line. Furthermore, the reports for each of the triaged crashes are also provided.

This cluster summary provides an overview of the identified crashes, allowing developers to focus on the most critical and frequent ones. The description of each crash with error information and crash line helps in understanding the root cause of the issue and the code segment responsible for the crash. Moreover, the individual reports for each triaged crash can provide further insights into the specific problem encountered and aid in resolving the issue.

Overall, the summary of clusters and individual reports generated by Casr can be a valuable resource in the debugging process and help developers save time by prioritizing the most severe and frequent crashes.

3.4. Continuous Integration. Continuous fuzzing integration utilizing the Sydr-Fuzz [33] framework can be automated using GitLab CI, as demonstrated by the OSS-Sydr-Fuzz [15] project. It requires the following prepared files:

- a Dockerfile to build the project and fuzz targets,

- fuzzing configuration TOML-files,
- target seed corpus and dictionaries.

To prepare fuzz targets, developers should build the project with the address sanitizer to detect crashes in native extensions and import the Atheris library for Python fuzzing. The CI artifacts include an archive of triaged crashes and their corresponding Casr [2] reports, the corpus, collected coverage, and log files for each step of the pipeline.

The fuzzing process is launched when an external CI trigger event occurs. The Docker container is built (utilizing all cores) and used for all project fuzz targets, and each fuzz target is launched through all dynamic pipeline steps. All fuzz targets are analyzed in parallel. Each dynamic analysis step utilizes at most 4 cores for a single fuzz target. The fuzzing job output includes resulting corpus, coverage information, Casr reports, and error triggering seeds.

By automating the fuzzing process, developers can more efficiently identify and address potential bugs and vulnerabilities, improving the overall quality and security of the software. The use of GitLab CI and the proposed toolset simplifies the process and reduces the time and resources required for effective fuzzing.

§4. Attack surface analysis

To effectively fuzz the code of AI frameworks, it is important to identify the potential attack surface and narrow down our focus to the parts we are interested in.

Firstly, we can divide the implementation of the frameworks into two main parts: the kernel that contains the core functionality, and the ecosystem which includes both the framework's own extensions and third-party libraries that are based on the framework.

Secondly, while the major functionality of AI frameworks is implemented using low-level languages such as C/C++, they also provide programming interfaces for Python. These interfaces are generated using code generation, extensions, and other binder libraries.

Thirdly, AI frameworks rely on several third-party software modules for implementing their kernel functionality. These dependencies may include libraries for data serialization and deserialization, data processing, and more. Errors in such dependencies can be critical and therefore, we need to consider them as well.

To conduct our analysis, we focus on identifying errors in the Python API using Sydr-Fuzz [33] with Atheris [1] engine. We create Python fuzz targets that cover the kernel interfaces, which internally interact with the kernel program interfaces written in C/C++, through Python bindings. This approach enables us to test both the C/C++ and Python code simultaneously. Additionally, we also prioritize fuzzing the Python API of external libraries from the AI ecosystem.

§5. Evaluation

We applied our Python dynamic analysis pipeline to two of the most widely used machine learning frameworks, TensorFlow [23] and PyTorch [19]. Additionally, we explored the TensorFlow ecosystem and found that the h5py [9] tool was well-suited for fuzzing with Sydr-Fuzz [33]. By utilizing our dynamic analysis pipeline, we were able to comprehensively analyze these frameworks and identify potential vulnerabilities, improving the security and reliability of these widely used tools.

5.1. Experimental Setup. Our experiments were conducted on a machine with two AMD EPYC 7542 32-Core processors and 512GB RAM, running Ubuntu 20.04 and Python 3.8. To ensure consistent and reproducible results, all analysis stages were launched within Ubuntu-based Docker containers with the target projects pre-built and installed [15]. The Docker build stage was optimized to utilize all available cores, while each fuzz target analysis was restricted to using at most 4 cores. In order to balance the need for comprehensive testing with practical constraints, we stopped fuzzing when there was no new coverage for one hour or when the total fuzzing time exceeded 24 hours.

5.2. PyTorch. We conducted an analysis of PyTorch 1.13 [19], an open-source machine learning framework, through its Python API. After examining the kernel with core functionality, we determined that the PyTorch JIT module was the most suitable for writing fuzz targets, specifically focusing on the `torch.jit.load` function, which loads a `ScriptModule` or `ScriptFunction` that has been previously saved with `torch.jit.save`.

To perform our fuzzing process, we built the PyTorch project from source and linked it to the libFuzzer library, adding necessary instrumentation for ASAN support (`-fsanitize=fuzzer-no-link,address`). We then installed the produced `.whl` package via `pip`.

Our fuzz target for the Python API function [16] involved calling `torch.jit.load` with a file containing input bytes as a parameter. Utilizing four cores, we discovered two crashes [12,21] during the fuzzing process, both of which were related to null pointer dereference and SEGV due to read memory access in a third-party PyTorch dependency called *flatbuffers*. Subsequently, we were able to find these same two crashes by fuzzing the C++ PyTorch API with Sydr-Fuzz [33] via AFL++ [27] fuzzer. We submitted a pull request to PyTorch in order to address these issues [20].

5.3. h5py. The HDF5 [10] binary data format is a crucial part of the AI ecosystem, and h5py [9], a Pythonic interface to it, is widely used in TensorFlow [23] module Keras [24] for saving and loading models in HDF5 format.

To prepare for the dynamic analysis pipeline, we built and installed the h5py 3.8.0 project in a specific way. First, we built the HDF5 project, the C core of h5py, from source and linked it to libFuzzer library. We also added necessary instrumentation for ASAN support (`-fsanitize=fuzzer-no-link,address`). Second, we built h5py itself in the same way using the preliminary built HDF5 core module. Finally, we installed the produced `.whl` package using `pip`.

Since the library's purpose is to manage HDF5 files, we chose the `File` object constructor as the fuzz target [14]. The fuzz target contained `h5py.File` object initialization that took a file with input data in HDF5 format as a parameter. During the fuzzing process on 4 cores, we discovered one crash that led to an out-of-bounds access on read [18]. It is noteworthy that this crash was also found by fuzzing the C-project HDF5 itself through `H5Fopen/H5Dopen2` functions. We submitted a pull request to HDF5 in order to address this issue [5].

5.4. TensorFlow. We applied the Sydr-Fuzz Python dynamic analysis pipeline to TensorFlow 2.11 [23], one of the most popular open-source machine learning platforms. After building the project from source and linking it to the libFuzzer library with added ASAN support, we extensively researched the TensorFlow Python API and wrote fuzz targets for modules such as `io`, `audio`, `keras`, `image`, and `strings` [17]. Despite our efforts, no crashes were found during the fuzzing process.

5.5. Crash Triaging with Casr. Casr [2] was used to simplify crash analysis. Initially, Python reports were generated for the crashes that were

Table 1. Crash Triaging with Casr (4 cores).

Fuzz Target	PyTorch (load)	h5py (file)
Number of Crashes	345	190
Deduplicated Reports	2	1
Number of Clusters	2	-
Reports Collection (sec)	3767	601
Deduplication (sec)	2	1
Clustering (sec)	1	-

found. The reports were then deduplicated and clustered, and severity estimation was performed. Casr was run on 4 cores.

Table 1 shows that Casr was effective in reducing the number of crashes. For PyTorch, the tool reduced the number of crashes from 345 to 2 that were significantly different, while for h5py, it reduced the number of crashes from 190 to just 1. Generating the reports was the most time-consuming part of the process, as each crashing seed required a fuzz target run, which had a long Atheris initialization stage. However, report deduplication and clustering were quick, taking only a couple of seconds.

§6. Conclusion

In conclusion, our analysis of different approaches to testing AI frameworks led us to propose our own approach utilizing Sydr-Fuzz [33], which includes a sequential pipeline of coverage-guided fuzzing with Atheris [1], corpus minimization, coverage collection, and crash triaging with Casr [2] that includes crash deduplication, clustering, and preparing human-friendly convenient reports. This pipeline was automated with GitLab CI.

We chose to focus on fuzzing Python APIs of AI frameworks and tools from their ecosystem, building C/C++ parts of libraries with libFuzzer [31] and sanitizers instrumentation to provide full-fledged fuzzing. We settled on fuzzing `load` function from PyTorch JIT module, file opening module from h5py, and some functions for parsing different formats from TensorFlow. Through this approach, we were able to find crashes in C/C++ parts of PyTorch and h5py through their Python APIs, ultimately deduplicating and clustering them into a manageable number of crashes: 2 crashes

in PyTorch [12, 21] and 1 crash in h5py [18]. All the found crashes were patched by corresponding pull requests [5, 20].

References

1. *Atheris: A coverage-guided, native Python fuzzer*, https://github.com/google/atheris.
2. *Casr: Collect crash reports, triage, and estimate severity*, https://github.com/ispras/casr.
3. *Coverage.py: A tool for measuring code coverage of Python programs*, https://coverage.readthedocs.io.
4. *Fix of endless loop error in TensorFlow*, https://github.com/tensorflow/tensorflow/pull/56455/files.
5. *Fix out of bounds in hdf5/src/h5fint.c:2859*, https://github.com/HDFGroup/hdf5/pull/2691.
6. *FuzzedDataProvider*, https://github.com/google/fuzzing/blob/master/docs/split-inputs.md#fuzzed-data-provider.
7. *genhtml: Generate html view from lcov coverage data files*, https://linux.die.net/man/1/genhtml.
8. *Google sanitizers*, https://github.com/google/sanitizers.
9. *h5py: HDF5 for Python*, https://github.com/h5py/h5py.
10. *HDF5 project*, https://github.com/HDFGroup/hdf5.
11. *Hypothesis library*, https://hypothesis.works/.
12. *Null pointer dereference in third_party/flatbuffers/include/flatbuffers/vector.h:158:48*, https://github.com/pytorch/pytorch/issues/95061.
13. *OSS-Fuzz: Continuous fuzzing for open source software*, https://github.com/google/oss-fuzz.
14. *OSS-Sydr-Fuzz h5py project*, https://github.com/ispras/oss-sydr-fuzz/tree/master/projects/h5py.
15. *OSS-Sydr-Fuzz: Hybrid fuzzing for open source software*, https://github.com/ispras/oss-sydr-fuzz.
16. *OSS-Sydr-Fuzz PyTorch project*, https://github.com/ispras/oss-sydr-fuzz/tree/master/projects/pytorch-py.
17. *OSS-Sydr-Fuzz TensorFlow project*, https://github.com/ispras/oss-sydr-fuzz/tree/master/projects/tensorflow-py.
18. *Out of bounds access on read in hdf5/src/h5fint.c:2859:13*, https://github.com/HDFGroup/hdf5/issues/2432.
19. *PyTorch project*, https://github.com/pytorch/pytorch.
20. *Segmentation fault in flatbuffers when parsing malformed modules*, https://github.com/pytorch/pytorch/pull/95221.
21. *SEGV in flatbuffers/base.h:406:23*, https://github.com/pytorch/pytorch/issues/95062.
22. *Sydr-Fuzz trophies*, https://github.com/ispras/oss-sydr-fuzz/blob/master/TROPHIES.md.
23. *TensorFlow: An open source machine learning framework for everyone*, https://github.com/tensorflow/tensorflow.

24. *TensorFlow Keras module*, `https://www.tensorflow.org/api\_docs/python/tf/keras?version=nightly`.
25. *TorchVision project*, `https://github.com/pytorch/vision`.
26. *Using instrumentation with Atheris and native extensions*, `https://github.com/google/atheris/blob/master/native\_extension\_fuzzing.md`.
27. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, *AFL++: Combining incremental steps of fuzzing research*, 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020.
28. H. V. Pham, T. Lutellier, W. Qi, and L. Tan, *CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries*, 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1027–1038.
29. G. Savidov and A. Fedotov, *Casr-Cluster: Crash clustering for linux applications*, 2021 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE, 2021, pp. 47–51.
30. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, *AddressSanitizer: A fast address sanity checker*, 2012 USENIX Annual Technical Conference (USENIX ATC 12), 2012, pp. 309–318.
31. K. Serebryany, *Continuous fuzzing with libFuzzer and AddressSanitizer*, 2016 IEEE Cybersecurity Development (SecDev), IEEE, 2016, p. 157.
32. K. Serebryany, *OSS-Fuzz - Google's continuous fuzzing service for open source software*, USENIX Association, 2017.
33. A. Vishnyakov, D. Kuts, V. Logunova, D. Parygina, E. Kobrin, G. Savidov, and A. Fedotov, *Sydr-Fuzz: Continuous hybrid fuzzing and dynamic analysis for security development lifecycle*, 2022 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE, 2022.
34. Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, *Deep Learning Library Testing via Effective Model Generation*, ESEC/FSE 2020, ACM, 2020, pp. 788–799.
35. A. Wei, Y. Deng, C. Yang, and L. Zhang, *Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source*, ICSE '22, ACM, 2022, pp. 995–1007.

Ivannikov Institute
for System Programming of the RAS,
Lomonosov Moscow State University

E-mail: `Yegorov_Ilya@ispras.ru`

E-mail: `kobrineli@ispras.ru`

E-mail: `pa_darochek@ispras.ru`

E-mail: `vishnya@ispras.ru`

E-mail: `fedotoff@ispras.ru`

Поступило 6 сентября 2023 г.