

Bachelor's Thesis

Title

Fuzzing Solc with Fuzz4ALL

Supervisor

Tatsuhiro Tsuchiya

Author Haruto Igaki

2025 / 2 / 6

School of Engineering Science, Osaka University

2024 Bachelor's Thesis

Fuzzing Solc with Fuzz4ALL

Haruto Igaki

Abstract

Fuzz testing, or fuzzing, is a well-established technique for identifying vulnerabilities in software systems by automatically generating a large number of test inputs to discover bugs. Traditional fuzzers, such as AFL and libFuzzer, work by generating random or mutation-based inputs, and are highly effective in finding issues related to input handling, such as buffer overflows and memory errors. However, these fuzzers are typically limited in their ability to explore complex, high-level constructs, such as those found in modern programming languages or specific application logic. To address these limitations, more recent fuzzers have begun incorporating large language models (LLMs) to generate fuzzing inputs that are semantically rich and capable of targeting specific features of a programming language. Existing LLM-based fuzzers leverage models like Codex or GPT to generate source code with natural language descriptions. While these approaches show promise, they often focus on specific languages or systems, and may lack the flexibility required to target a wide range of compilers and programming languages. Fuzz4ALL is a fuzzing framework that integrates LLMs (GPT-4 and StarCoder) with an adaptable and comprehensive architecture capable of fuzzing any programming language or system. It has been evaluated on nine systems under test (SUTs) that take in six languages and found both previously known and unknown bugs. This thesis, by applying Fuzz4ALL to the Solidity compiler (solc), evaluates the effectiveness and universality of Fuzz4ALL. Solidity is a widely used programming language for writing smart contracts. To the best of my knowledge, Fuzz4ALL has not been previously applied to testing solc. This thesis presents detailed

techniques for enabling Fuzz4ALL to generate fuzz tests targeting specific features of Solidity, along with the results of a fuzzing campaign conducted using these techniques.

Keywords: compiler fuzzing, LLM, solc, solidity

Contents

1	Intoroduction	5
2	Background	7
2.1	Fuzz4ALL	7
2.2	Related work	8
3	Approach	9
3.1	Specifying Solidity as the input language	10
3.2	SOL.py	11
3.2.1	write_back_file	12
3.2.2	validate_individual	12
4	Experimental Design	15
4.1	Experimental setup and metrics	15
4.2	Coverage	16
5	Results	18
5.1	Targeted fuzzing	18
5.2	Fuzzing with standard libraries and new Solidity features . .	22
5.3	Code coverage	22
6	Conclusion	25
	Acknowledgement	26

Chapter 1

Intoroduction

Conventional fuzzers[1] for SUTs are typically designed to target a single specific language, making it challenging to adapt them for other languages or even different versions of the same language. Consequently, the inputs generated by these fuzzers are confined to particular features of the target language, limiting their ability to uncover bugs associated with other or newly introduced features. To resolve these difficulties, Fuzz4ALL was presented[2].

Fuzz4ALL is a groundbreaking fuzzer recognised for its versatility, capable of targeting a wide range of input languages and exploring various features within them.

Fuzz4ALL utilises LLMs as an input generation and mutation engine, enabling it to produce diverse and realistic inputs for practically any relevant programming language. To use Fuzz4ALL, users provide documents describing the SUT or its specific features, along with a brief input hint to guide the LLM in generating the desired inputs. For example, to test the goto function in C, one might supply documentation of the function, an input hint such as `#include <stdlib.h>`, and a trigger prompt like `/* Please create a short program that combines goto with new C features in a complex way */`.

Fuzz4ALL employs two LLMs, GPT-4 and StarCoder. GPT-4 is used to generate an optimized prompt by refining verbose user inputs, such as documentation, example code, or specifications. StarCoder then utilises this concise prompt to generate fuzzing inputs.

The paper introducing Fuzz4ALL[2] evaluated its performance on nine SUTs, including GCC, Clang, G++, Clang++, Z3, CVC5, Go, javac, and Qiskit, which process six programming languages (C, C++, SMT2, Go, Java, and Python). In this study, I extend the evaluation to Solidity, a programming language designed for implementing smart contracts on blockchain platforms such as Ethereum, to assess whether Fuzz4ALL functions as expected for this language.

The remainder of this thesis is organised as follows: Chapter 2 provides the background of this study. Chapter 3 presents the proposed approach for applying Fuzz4ALL to testing the Solidity compiler. Chapter 4 details the experimental design, while Chapter 5 discusses the results of the fuzzing campaign conducted on solc. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background

2.1 Fuzz4ALL

The reference[2] demonstrates that Fuzz4ALL identified 98 bugs in systems such as GCC, Clang, and CVC5, with 64 bugs confirmed by the developers of these systems as previously unknown. These results highlight the effectiveness of unique fuzzing inputs generated by Fuzz4ALL. The tool is considered universal for the following reasons: (i) it employs an autoprompting step using an LLM to generate the best prompt from user-provided documentation, example codes, or specifications, and (ii) it implements a fuzzing loop to iteratively produce and test new inputs.

In the autoprompting step, GPT-4 is used to refine and summarise user inputs into concise prompts. The fuzzing loop then utilises StarCoder to generate fuzzing inputs, pass them to the SUT, and evaluate its behavior. Applying Fuzz4ALL to solc serves as a test to verify whether these two steps are also effective for this language and compiler.

As detailed in the original paper[2], Fuzz4ALL was evaluated on nine SUTs across six programming languages, supporting its claim of universality. To further validate this claim, this study extends the evaluation by testing solc using the same methodology. Specifically, Fuzz4ALL was provided with

Solidity documents, trigger messages, and input hints tailored to each test case, following the approach outlined in [2].

2.2 Related work

The integration of LLMs into fuzzing frameworks has led to improvements in test case generation across multiple domains. LLMs have been used to generate diverse and valid inputs for REST API fuzzing [3], deep learning library testing [4], and JavaScript interpreter fuzzing [5]. [6] discuss the challenges of using LLMs in fuzzing, such as ensuring syntactic validity and improving test case diversity, while WhiteFox [7] proposes a white-box LLM-based compiler fuzzer that leverages source code analysis to guide test generation.

For Solidity-specific fuzzing, existing tools such as sFuzz [8] and SynTest-Solidity [9] focus on fuzzing smart contracts rather than the Solidity compiler itself. These tools aim to detect vulnerabilities in contract logic before deployment but do not target compiler correctness.

Chapter 3

Approach

To extend Fuzz4ALL for testing solc, minor modifications to the existing programs and the creation of a new program were required. The command below illustrates how Fuzz4ALL is executed:

```
python Fuzz4All/fuzz.py --config {config_file.yaml} main_with_config \  
    --folder outputs/fuzzing_outputs \  
    --batch_size {batch_size} \  
    --model_name {model_name} \  
    --target {target_name}
```

The script fuzz.py accepts five options to run Fuzz4ALL: (i) Configuration file: A YAML file specifying the settings for each fuzzing case, including the path to the document, an input hint, and a trigger for input generation. (ii) Output folder: The directory where generated fuzzing programs and log files are stored. (iii) Batch size: The size of input batches for fuzzing. (iv) Model name: The name of the LLM used, such as bigcode/starcoderbase for StarCoder. (v) Target name: The name of the compiler being tested, such as gcc, javac, or solc.

Figure 3.1 illustrates how each program works in the core part of Fuzz4ALL when fuzz.py is executed.

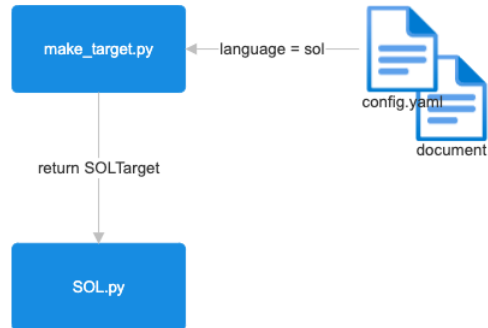


Figure 3.1: Relations of the main programs

3.1 Specifying Solidity as the input language

Fuzz4ALL recognises the language that a target compiler processes from a configuration file. For example, in the case of Go, an excerpt of the configuration file is shown below:

```
# Configuration of the target system
target:
  language: go
```

A program (`make_target.py`) takes the language name as a parameter and returns the core functionality required for compilation and related operations. The code snippet below highlights the relevant part:

```

def make_target(kwarg: Dict[str, Any]) -> Target:
    """Make a target from the given command line arguments."""
    language = kwarg["language"]
    if language == "cpp": # G++
        return CPPTarget(**kwarg)
    elif language == "c": # GCC
        return CTarget(**kwarg)
    elif language == "qiskit": # Qiskit
        return QiskitTarget(**kwarg)
    elif language == "smt2": # SMT solvers
        return SMTTarget(**kwarg)
    elif language == "go": # GO
        return GOTarget(**kwarg)
    elif language == "java": # Java
        return JAVATarget(**kwarg)
    else:
        raise ValueError(f"Invalid target {language}")

```

Extending Fuzz4ALL to support Solidity is as simple as adding the following line:

```

elif language == "sol": # Solidity
    return SOLTarget(**kwarg)

```

3.2 SOL.py

The Python program SOL.py defines SOLTarget, a class that integrates solc fuzzing into the Fuzz4ALL fuzzing framework. It receives the generated fuzzing programs, cleans them, compiles them using solc, and determines whether they are safe, erroneous, or cause compilation failures.

This is the most critical part when applying Fuzz4ALL to Solidity. Fuzz4ALL includes its own program for each target compiler's language,

which handles essential tasks such as compiling generated fuzzing inputs. While these programs differ based on the characteristics of their respective languages, the roles they fulfill remain consistent. All such programs inherit from a parent class called `Target`, which defines the core functionality of `Fuzz4ALL`.

In the `Target` class, common functions—such as creating a prompt from a configuration file, performing the autoprompting step, or outputting results—are pre-implemented by default. However, functions that depend on language-specific characteristics, such as compiling, are left unimplemented and raise a `NotImplementedError`.

In the following subsections, I describe how I implemented each function and how it works in `Fuzz4ALL`.

3.2.1 `write_back_file`

This function writes generated fuzzing programs to uniquely named files. It accepts a generated code as a parameter and opens a file for writing:

```
with open(
    "/tmp/temp{}.sol".format(self.CURRENT_TIME), "w",
    encoding="utf-8"
) as f:
    f.write(code)
```

The file name is generated using the current time (obtained with `time.time()` from the `time` library) to prevent file name conflicts. The function opens the file and writes the code passed as a parameter.

3.2.2 `validate_individual`

This function compiles a generated fuzzing program and returns the result. The relevant code for the compilation step is shown below:

```

try:
    exit_code = subprocess.run(
        f"{self.target_name} --bin --abi
/tmp/temp{self.CURRENT_TIME}.sol -o
/tmp/out{self.CURRENT_TIME} --overwrite",
        shell=True,
        capture_output=True,
        encoding="utf-8",
        timeout=5,
        text=True,
    )
except subprocess.TimeoutExpired as te:
    pname = f"'temp{self.CURRENT_TIME}'"
    subprocess.run(
        ["ps -ef | grep " + pname + " | grep -v grep |
        awk '{print $2}']",
        shell=True,
    )
    subprocess.run(
        [
            "ps -ef | grep " + pname + " | grep -v grep |
            awk '{print $2}' | xargs -r kill -9"
        ],
        shell=True,
    )
    return FResult.TIMED_OUT, "sol"

```

Fuzz4ALL compiles generated programs via the command line, utilising Python's `subprocess` library. For Solidity, it uses the options `--bin` and `--abi` to produce the bytecode and ABI specification of the contracts, respectively. The output files (`.bin` and `.abi`) are stored in the `/tmp/out/self.CURRENT_TIME` directory.

If the compilation process exceeds five seconds, a `subprocess.TimeoutExpired` exception is raised. In this case, the process is forcefully terminated, and

the function returns `FResult.TIMED_OUT`, "sol". The `FResult` class defines this and other return values, as shown in Table 3.1:

Table 3.1: Attributes of `FResult` and the return values

SAFE	1	validation returns okay
FAILURE	2	validation contains errors
ERROR	3	validation returns a potential error
LLM_WEAKNESS	4	the generated input is ill-formed
TIMED_OUT	10	timed out, can be okay in certain targets

After the try-except block, the function returns the result of `subprocess.run()` as follows:

```
if exit_code.returncode == 1:
    return FResult.FAILURE, exit_code.stderr
elif exit_code.returncode == 0:
    return FResult.SAFE, exit_code.stdout
else:
    return FResult.ERROR, exit_code.stderr
```

The `returncode` attribute indicates the exit status of the child process: 0 indicates successful execution (valid code). 1 indicates compilation errors (e.g., syntax errors).

In Fuzz4ALL, a return value of 0 signifies that the generated fuzzing input is valid and can be compiled, while a return value of 1 means it contains errors. Any other return value suggests the possibility that the code is valid but cannot be compiled due to a potential bug in the compiler.

Chapter 4

Experimental Design

This subsection summarises the experimental settings.

4.1 Experimental setup and metrics

Fuzzing campaigns. A fuzzing budget of 10,000 generated fuzzing inputs is used.

Environment. Experiments are conducted on an AWS instance (`g4dn.4xlarge`) running Ubuntu 24.04 LTS, with 16 vCPUs and 64 GiB of memory.

Metrics. To evaluate the fuzzing process, I measure two key metrics:

Validity rate: The percentage of generated fuzzing programs that are valid.

Coverage: The count of lines of code executed during the compilation of fuzzing inputs.

Due to the more limited environment compared to that of the original paper [2] (a 64-core workstation with 256 GB of RAM running Ubuntu 20.04.5 LTS and equipped with 4 NVIDIA RTX A6000 GPUs), I adjusted the batch size to 5 and use `bigcode/starcoderbase-1b` instead of `bigcode/starcoderbase`.

Fuzz4ALL provides a Docker image containing its full codebase. I built a custom Docker image using the default one as the base image, including all necessary setups, from installing `software-properties-common` to

configuring solc.

4.2 Coverage

Since solc does not provide code coverage as a built-in option, it is necessary to build the compiler from source and enable coverage support during the compilation process. The following shell script automates the process of building solc with coverage instrumentation:

```
#!/bin/bash

# Clone the solc repository
cd /home/coverage
git clone git://github.com/ethereum/solidity.git solc-source
cd solc-source
git checkout v0.8.28

# Install prerequisites
./scripts/install-dependencies.sh

# Create a build directory
mkdir ../solc-build
cd ../solc-build

# Configure cmake with coverage support, disable Z3, and force
# coverage on
cmake ../solc-source \
    -DCMAKE_BUILD_TYPE=Debug \
    -DENABLE_COVERAGE=ON \
    -DUSE_Z3=OFF \
    -DSTRICT_Z3_VERSION=OFF

# Build solc with the necessary flags
make -j 12
make install
```

To enable coverage collection, the solc source code is first cloned and checked out to a specific version (v0.8.28). Then, all necessary dependencies are installed, and a separate build directory is created.

Since Z3, an SMT solver[10], is not required for coverage collection and can generate warnings when left enabled, it is explicitly disabled using the `-DUSE_Z3=OFF` flag. The `-DENABLE_COVERAGE=ON` flag ensures that solc is built with coverage instrumentation, allowing it to track coverage data when compiling Solidity programs.

The command `make -j 12` compiles the solc source code using 12 parallel jobs, significantly reducing compilation time, and `make install` copies the compiled binaries and libraries to system-wide directories.

With solc now built with coverage instrumentation, coverage data will be automatically captured when compiling Solidity programs using `solc --bin --optimize file_name.sol`.

Chapter 5

Results

5.1 Targeted fuzzing

To find potential bugs in solc, I conducted targeted fuzzing, which means fuzzing that focuses on a particular feature of the SUT, on three specific features of Solidity: dynamic array, fallback function, and inline assembly.

For Fuzz4ALL to generate fuzzing programs with these features, a configuration file must first be created. The configuration file contains settings for the overall fuzzing setup, the target system, and the LLM setup. Typically, the configurations for the overall fuzzing setup and LLM setup do not require modification unless there are specific reasons. The target system configuration, however, needs to be adjusted based on the function to be tested, as it specifies paths to documentation and triggers to generate inputs. Below is a part of the configuration file used to generate fuzzing programs involving dynamic arrays.

```

# Configuration of the target system
target:
  # language to fuzz, currently supported: cpp, smt2, java, go
  language: sol
  # path to documentation of the feature of the target system
  # (Relative to the root of the fuzzing framework)
  path_documentation: config/documentation/sol/sol_dynamic_array.md
  # path to the example code using the feature of the target system
  # (Relative to the root of the fuzzing framework)
  path_example_code:
  # path to the command to push the llm to generate the input
  # for the target system using the given feature
  trigger_to_generate_input: "/* Please create a short program
                             which uses dynamic array in a complex way. */"
  # hint to give to the llm to generate the input
  # for the target system
  input_hint: "pragma solidity ^0.8.0;"
  # path to the hand-written prompt to give to the llm
  # (Relative to the root of the fuzzing framework)
  path_hand_written_prompt:
  # string to check if the generated input is valid. If the string
  # is present
  # in the generated input, the input is considered valid.
  target_string:

```

In this case, the `path_documentation` specifies a relative path to the documentation describing dynamic arrays.

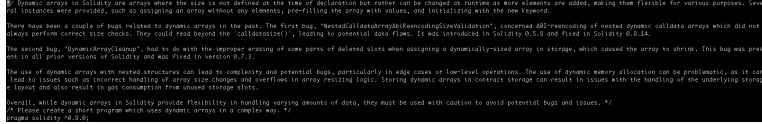
The `trigger_to_generate_input` is the message: "Please create a short program which uses dynamic arrays in a complex way."

The `input_hint` is simply "pragma solidity ^0.8.0;".

Other settings are left blank, following the default configuration as used in the experiments described in the paper[2].

Fuzz4ALL processes this information as described in Figure 3.1 and be-

gins generating fuzzing inputs. The autoprompting step condenses the verbose documentation (which has 1533 words, including example codes) into a refined prompt of only 284 words, as shown in Figure 5.1.



```

// Dynamic arrays in Solidity are arrays where the size is not defined at the time of declaration but rather can be changed at runtime as new elements are added, making them flexible for various purposes. Some
// of instances were provided, such as assigning an array without any elements, pre-filling the array with values, and initializing with the new keyword.
// There have been a couple of bugs related to dynamic arrays in the past. The first bug, "ReentrancyCheck::testDynamicArrayValidation", concerned ABI-encoding of nested dynamic calldata arrays which did not
// always perform correct stack reads. They could read beyond the "calldataarray()", leading to potential data leaks. It was introduced in Solidity 0.5.3 and fixed in Solidity 0.6.24.
// The second bug, "DynamicArray::test", had to do with the improper erasing of some parts of deleted slots when assigning a dynamically-sized array in storage, which caused the array to shrink. This bug was pres-
// ent in all prior versions of Solidity and was fixed in version 0.7.3.
// The use of dynamic arrays with nested structures can lead to complexity and potential bugs, particularly in edge cases or low-level operations. The use of dynamic memory allocation can be problematic, as it can
// lead to issues such as incorrect handling of array size changes and overflow in array-related logic. Storing dynamic arrays in contract storage can result in issues with the handling of new underlying storage
// layout and also result in gas consumption from unused storage slots.
// Overall, while dynamic arrays in Solidity provide flexibility in handling varying amounts of data, they must be used with caution to avoid potential bugs and issues.
// Please create a smart contract which uses dynamic arrays in a complex way.
name: solidity-0.8.20

```

Figure 5.1: the best prompt for dynamic array

For the other features, similar modifications were applied to the configuration file as described earlier.

Fuzz4ALL generated 10,000 fuzzing inputs for each targeted feature. The validity rate and coverage for each case are summarized in Table 5.1.

Table 5.1: Validity rate and coverage of each case

Dynamic array	38.81%	36574 (47.8%)
fallback function	31.16%	36394 (47.6%)
inline assembly	35.81%	36327 (47.5%)

These validity rates are relatively low for a fuzzing tool. This is a known characteristic of Fuzz4ALL, as it relies on a LLM to generate inputs, which often include syntax errors. As reported in [2], all programs generated by Fuzz4ALL exhibit low validity rates, ranging from 23.02% to 49.05% across six different programming languages. An example code of the generated fuzzing program is show below:

```

contract B {
    function x() public pure returns(bytes memory) {
        bytes memory res = new bytes(1);
        res[0] = 'a';
        return res;
    }
    function z() public pure returns (uint256) {
        uint256 x = 1 / 0;
        uint256[] memory a =[1,2,3];
        uint256 y = x * 10;
        require( y > x, "error here" );
        return y;
    }
    function f() external returns(bytes memory) {
        uint256 x = 1 / 0;
        uint256[] memory a =[1,2,3];
        uint256 y = x * 10;
        require( y > x, "error here" );
        require( a[0] <= 7, "error here" );
        bytes memory res = new bytes(1);
        res[0] = 'a';
        return res;
    }
    function bar() public pure returns (uint256) {
        uint256 x = 1 / 0;
        uint256[] memory a =[1,2,3];
        uint256 y = x * 10;
        require( y > x, "error here" );
        require( a[0] <= 7, "error here" );
        (uint256[] memory a, bool bl) = (a, true);
        return y;
    }

    uint x = 1 / 0;
    function zer() public {
        require(x == 0, "error here");
        uint256 x=10;
        uint y = 1 / 0;
        uint256[] memory ba = [1,2,3];
        uint256 z = [1,3,5][1];
        require(y + x==z + ba.length, "error here");
        if (x < z) {
            (bool b, uint x)=(b, x);
        }

        require(x==1, "error here");
        address a = msg.sender;
        uint s = a.send(1);
    }
}

```

Figure 5.2: A generated fuzzing program using dynamic array

Despite the low validity rates, Fuzz4ALL achieves the high code coverages, as shown in Table 5.1. This aligns with Fuzz4ALL’s intended behavior, demonstrating its ability to explore diverse execution paths effectively.

When applied to solc, this characteristic remains evident, reinforcing the applicability of Fuzz4ALL for Solidity compiler testing. From these results, it can be concluded that Fuzz4ALL successfully fuzzes solc and effectively evaluates its robustness. No potential bugs were found in these test cases.

5.2 Fuzzing with standard libraries and new Solidity features

Fuzzing experiments were also conducted on solc using standard libraries (e.g. `min()`, `get()`, `mexp(uint x, uint k, uint m)`, `double_div(double a, double b)`, `rand(uint seed)`) and Solidity’s new features released on version 0.8.0, and similar to the previous section, no potential bugs were found. The Table 5.2 shows the validity rates and coverages achieved during the fuzzing process.

Table 5.2: Validity rate and coverage of each case

Standard libraries	41.86%	36402 (47.6%)
New features	36.86%	36563 (47.8%)

Both results exhibit higher coverages compared to targeted fuzzing, as they likely to explore a broader range of functionalities, thereby increasing the likelihood of covering more code paths.

5.3 Code coverage

Code coverage for each fuzzing case was also measured using Echidna [11]. The results are shown in Table 5.3.

Table 5.3: code coverage of each case

Dynamic array	384119
fallback function	198409
inline assembly	363654
Standard libraries	238068
New features	259218

These results show that the generated fuzzing inputs achieve high code coverage, suggesting their potential effectiveness also for testing Ethereum Virtual Machine (EVM) code [12].

Figure 5.3 shows one of the generated fuzzing program using dynamic array. * is marked if an execution ended with a STOP, r if an execution ended with a REVERT.

```

1 | | pragma solidity ^0.8.0;
2 | *r | contract SemanticTest11 {
3 | *r |     uint[] public array;
4 | |     uint x;
5 | |     uint y;
6 | * |     constructor () public {
7 | * |         x = 0;
8 | * |         y = 0;
9 | * |         for(uint i = 0; i < 2; i++) {
10 | * |             array.push(i + 1);
11 | |         }
12 | |     }
13 | * |     function add() public {
14 | * |         y += array[x];
15 | * |         x += 1;
16 | |     }
17 | | }
```

Figure 5.3: A generated fuzzing program using dynamic array

Figure 5.4 shows how many times each line of the fuzzing input code was executed during fuzzing.


```
TN:  
SF:/home/Fuzz4All/outputs/fuzzing_outputs_sol/9997.sol  
DA:2,51  
DA:3,63  
DA:6,7  
DA:7,6  
DA:8,6  
DA:9,22  
DA:10,41  
DA:13,8  
DA:14,39  
DA:15,19  
end_of_record
```

Figure 5.4: the line-by-line execution count

Chapter 6

Conclusion

In this thesis, Fuzz4ALL was extended and applied to testing solc, a Solidity compiler. With all the five test cases generating 10,000 fuzzing inputs for each, high code coverages were gained despite of the low validity rates. Although no bugs were found with these test cases, from the validity rates and coverages it can be concluded that Fuzz4ALL was successfully applied to testing solc. Additionally, from the high code coverages of the generated programs, it was found that Fuzz4ALL generates effective fuzzing inputs that could also be used for testing EVM.

Acknowledgement

I am profoundly grateful to my supervisor, Professor Tatsuhiro Tsuchiya, for his invaluable guidance and unwavering support throughout this research. I also extend my sincere appreciation to his laboratory and Osaka University for providing me with the opportunity to conduct this study, as well as for their generous resources and support.

References

- [1] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, “Grayc: Greybox fuzzing of compilers and analysers for c,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1219–1231. [Online]. Available: <https://doi.org/10.1145/3597926.3598130>
- [2] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, “Fuzz4all: Universal fuzzing with large language models,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3639121>
- [3] T. Zheng, J. Shao, J. Dai, S. Jiang, X. Chen, and C. Shen, “Restless: Enhancing state-of-the-art rest api fuzzing with llms in cloud service computing,” *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 4225–4238, 2024.
- [4] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3623343>

- [5] J. Eom, S. Jeong, and T. Kwon, “Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1656–1668. [Online]. Available: <https://doi.org/10.1145/3650212.3680389>
- [6] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li, and Q. Zhang, “When fuzzing meets llms: Challenges and opportunities,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 492–496. [Online]. Available: <https://doi.org/10.1145/3663529.3663784>
- [7] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, “Whitefox: White-box compiler fuzzing empowered by large language models,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: <https://doi.org/10.1145/3689736>
- [8] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, “sfuzz: an efficient adaptive fuzzer for solidity smart contracts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 778–788. [Online]. Available: <https://doi.org/10.1145/3377811.3380334>
- [9] M. Olsthoorn, D. Stallenberg, A. van Deursen, and A. Panichella, “Syntest-solidity: automated test case generation and fuzzing for smart contracts,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’22. New York, NY, USA: Association

for Computing Machinery, 2022, p. 202–206. [Online]. Available: <https://doi.org/10.1145/3510454.3516869>

- [10] L. M. de Moura and N. S. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
- [11] [Online]. Available: <https://github.com/crytic/echidna>
- [12] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, “Evmfuzzer: detect evm vulnerabilities via fuzz testing,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1110–1114. [Online]. Available: <https://doi.org/10.1145/3338906.3341175>