

# Imperial College London

MENG INDIVIDUAL PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

---

## JEFRL: JavaScript Engine Fuzzing using Reinforcement Learning

---

*Author:*  
Pranav Maganti

*Supervisor:*  
Dr. Sergio Maffei

*Second Marker:*  
Prof. Alessio Lomuscio

June 23, 2023

## Abstract

Web browsers are an integral part of our everyday lives, being used for information, communication, and collaboration. As our reliance on these web technologies has grown, JavaScript engines have become a critical component embedded within web browsers, enabling the execution of dynamic and interactive content on web pages. However, the prominence of JavaScript engines has made them an attractive target for malicious actors aiming to exploit vulnerabilities to compromise the security of users' systems. Consequently, a significant amount of research has been dedicated to creating automated fuzz testing tools specifically designed for JavaScript engines.

We take a small amount of inspiration from FuzzBoost, a reinforcement learning mutation-based fuzzing tool for the C compiler, and introduce our ideas to implement a novel fuzzer for the JavaScript engine. We propose a mutation-based fuzzing tool for JavaScript engines, which uses reinforcement learning to guide to mutation of a program at the Abstract Syntax Tree (AST) level. To our knowledge, this fuzzer is the first of its kind, as no research has ever attempted to use AST level mutations with reinforcement learning.

From our testing, we found our fuzzer achieves 60% higher coverage than a random baseline showing that it has been successful in learning mutation sequences which increase coverage. We also found that our fuzzer did not reach the level of established state-of-the-art fuzzers, having increased the seed coverage by 17% less than DIE. However, we believe this technique shows incredible promise, and can achieve state-of-the-art performance, after some optimisation.

### **Acknowledgements**

I would like to begin by thanking my supervisor Dr. Sergio Maffeis and PhD student Myles Foley for their invaluable support and enthusiasm throughout the project.

I would like to thank my friends, without whom the last four years would have not been nearly as fun.

Finally, and most importantly, I would like to thank my family who have been a constant source of support and joy throughout my life.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Contributions . . . . .	5
1.2	Challenges . . . . .	6
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Web Browsers . . . . .	8
2.2	JavaScript Engines . . . . .	8
2.2.1	Design . . . . .	9
2.2.2	Past Vulnerabilities . . . . .	11
2.3	Fuzzing . . . . .	13
2.3.1	Generation-based Fuzzing . . . . .	14
2.3.2	Mutation-based Fuzzing . . . . .	15
2.3.3	Guided Fuzzing . . . . .	15
2.4	Reinforcement Learning . . . . .	16
2.4.1	Markov Decision Process . . . . .	16
2.4.2	Finding the Optimal Policy . . . . .	16
2.4.3	Q-Learning . . . . .	18
2.4.4	Deep Q-Learning (DQN) . . . . .	18
2.4.5	Double Q-Learning (Double DQN) . . . . .	19
2.5	Transformers . . . . .	20
2.5.1	Architecture . . . . .	20
2.5.2	BERT models . . . . .	22
<b>3</b>	<b>Related Research</b>	<b>24</b>
3.1	DIE . . . . .	24
3.2	FuzzBoost . . . . .	25
<b>4</b>	<b>Design</b>	<b>27</b>
4.1	Fuzzing as a Markov Decision Process . . . . .	28
4.1.1	Action Space . . . . .	28
4.1.2	State Space . . . . .	28
4.1.3	Episode Termination . . . . .	29
4.1.4	Rewards . . . . .	30
4.2	Environment . . . . .	30
4.2.1	Environment Initialisation . . . . .	30
4.2.2	Applying an action . . . . .	31
4.2.3	Execution . . . . .	33
4.2.4	Reward . . . . .	33

4.2.5	Corpus Expansion . . . . .	34
4.2.6	Return to agent . . . . .	34
4.3	Agent . . . . .	34
4.3.1	AST Vector Representation . . . . .	34
4.3.2	Double Deep Q-Network (Double DQN) . . . . .	36
<b>5</b>	<b>Implementation</b>	<b>37</b>
5.1	Language . . . . .	37
5.2	Pre-Processing . . . . .	37
5.3	JavaScript Engine Modification . . . . .	37
5.4	Training . . . . .	38
<b>6</b>	<b>Evaluation</b>	<b>40</b>
6.1	Experimental Setup . . . . .	40
6.2	Development Experiments . . . . .	40
6.2.1	AST Transformer . . . . .	40
6.2.2	Transformer AST fine-tuning . . . . .	41
6.2.3	Double DQN . . . . .	42
6.3	Research Questions . . . . .	45
6.3.1	Evaluating against naive baseline (RQ1) . . . . .	45
6.3.2	Evaluating against state-of-the-art (RQ2) . . . . .	46
6.3.3	Evaluating patterns found by DDQN agent (RQ3) . . . . .	48
6.3.4	Node type distribution . . . . .	48
6.3.5	Example test cases . . . . .	50
<b>7</b>	<b>Conclusion</b>	<b>52</b>
7.1	Future Work . . . . .	52
7.1.1	Optimising engine execution time . . . . .	52
7.1.2	Evaluating bug finding capabilities . . . . .	53
7.1.3	More fine-grained Replace action . . . . .	53
7.2	Ethical Considerations . . . . .	53
<b>A</b>	<b>Double DQN Component Training Times</b>	<b>54</b>

# Chapter 1

## Introduction

JavaScript has become an integral component of modern web development, enabling the creation of interactive and dynamic web pages that provide a seamless user experience [1, 2]. In order to execute JavaScript code, all browsers require a *JavaScript engine*, with some even electing to build their own from scratch [3, 4, 5]. JavaScript engines initially started as rudimentary interpreters [6]; however, they underwent significant enhancements and optimisations in response to the competitive landscape and drive to maximise their performance. [7, 8].

The increasing complexity and continuous code-base development of JavaScript engines have inevitably led to the introduction of bugs, with V8, the JavaScript engine used in Google Chrome, experiencing 76 high-severity vulnerabilities in the past three years alone [9]. These bugs can manifest in various forms, from typical bugs found in systems languages (C++), such as Use After Free (UAF), to ones unique to JavaScript engines. Bugs in the JavaScript engine pose challenges for developers, who expect consistent behaviour, and pose security risks as malicious entities can use these vulnerabilities to compromise billions of end-user systems that rely on JavaScript engines [10, 11].

Due to the severe consequences of bugs in JavaScript engines, a significant focus has been developing methods to test them automatically. Among these approaches, fuzzing has gained considerable attention due to its demonstrated success in uncovering bugs and vulnerabilities in complex software systems, including the C compiler [12].

Fuzzing is a dynamic testing methodology that involves generating random inputs and executing them on the System Under Test (SUT) to see if it causes unexpected behaviour, such as a crash or memory leak. Guided mutation-based fuzzing expands on this idea by starting with a set of seed files containing a wide range of inputs and mutating them based on feedback from execution. Several feedback mechanisms can be used, such as coverage or memory usage, with the fuzzer using these values to decide if this input should be kept for further mutation or discarded. Guided mutation-based fuzzing has found much success in JavaScript engine testing, with fuzzers such as Superior [13], DIE [14], and Fuzzilli [15] having found hundreds of bugs in several different engines.

Looking at guided mutation-based fuzzing from a broader perspective, we see that it

shares many similarities with sequential decision-making problems, where a decision-maker must select actions over time which maximise their final reward. In the context of guided mutation-based fuzzing, our mutations serve as actions, and the agent must strategically choose a sequence of mutations that elicit unexpected behaviour in the System Under Test (SUT). One well-known technique for solving sequential decision-making problems is Reinforcement Learning (RL), which has successfully trained agents that can find optimal strategies in several different environments [16, 17]. However, just because we **can** apply reinforcement learning to a problem does not always mean we **should**; We note that very little research has occurred in this area, and therefore, we must first consider what we aim to gain by applying this technique to the already well-researched and established area of JavaScript engine fuzzing.

Most guided mutation-based fuzzers rely on applying random mutations to the seed program to generate new inputs. However, research by Park et Al. [14] found that the seeds given to a fuzzer typically come from an engine’s test suite or previous vulnerabilities, which are designed to have a unique control flow and data dependencies that stress specific code paths. Applying random mutations leads to losing these semantic characteristics, which are usually unrecoverable. Park et Al. [14] attempted to solve this by creating their fuzzer called DIE, which avoids the mutations that drastically change a program’s control flow, such as removing if statements or for loops. This technique showed promising results compared to other random fuzzers such as Superion [13] and Nautilus [18].

By taking motivation from the ideas proposed by Park et Al. [14], we aim to improve existing guided fuzzing techniques by using Reinforcement Learning algorithms to decide the location and type of mutation. By doing so, we aim to automatically learn the aspects of code essential for finding vulnerabilities and hopefully improve upon hard-coded techniques that may have missed some deeper patterns.

## 1.1 Contributions

We believe to have made several unique contributions as follows:

- We propose a novel formulation of guided mutation-based fuzzing as a Markov Decision Process (MDP)
- We implement a unique Transformer-based Abstract Syntax Tree (AST) vector representation by adapting and improving existing techniques using Recurrent Neural Networks (RNN) for AST generation.
- We implement a novel (and, to our knowledge, the first) mutation-based JavaScript engine fuzzer which leverages reinforcement learning to choose a mutation action at each stage
- We quantitatively evaluate the effectiveness of our reinforcement learning directed fuzzer, both against a random benchmark and against other state-of-the-art fuzzers

## 1.2 Challenges

Our project involved running hundreds of experiments to fine-tune several variables, such as hyperparameters, architectures and reward functions. We faced several different challenges during the project:

- **Limited research into reinforcement learning directed fuzzing:** In our extensive background research, we only found two papers applicable to our area of interest. This lack of research meant that we had to try and take inspiration from existing fuzzers and devise novel ideas for how reinforcement learning may help improve upon them.
- **Unstable nature of reinforcement learning:** Reinforcement learning methods are hard to implement and tune effectively. Several challenges are faced when training an RL model, such as dealing with sparse rewards functions, balancing exploration vs exploitation and several other issues. Doing so takes much time due to the extensive training required for such models to learn from the environment.
- **Difficulties building an older version of engines:** To evaluate the bug-finding capabilities of our fuzzer, we wanted to use an older version of the engine with known bugs. However, due to the rapid development and ever-changing build tools, we could not build a version of the V8 JavaScript engine older than 2021.
- **Lack of documentation for other research fuzzers:** A lot of the fuzzers we wanted to use in our evaluation lacked the necessary documentation to build and run them. This significantly reduced the number of other state-of-the-art fuzzers we could use for comparisons.



# Chapter 2

## Background

In this chapter, we aim to give an overview of several relevant topics to this thesis. We start by presenting the basic architecture of the web browser and discussing how the JavaScript engine is a critical attack vector. We then overview the basic building blocks that most JavaScript engines employ, allowing us to better understand previous vulnerabilities in the engine.

We introduce the idea of Fuzzing or Fuzz testing, a testing methodology that uses randomly generated inputs to stress the system under test. We discuss several variations of fuzzing developed over the years and end the section with a review of a couple of existing JavaScript fuzzers.

We then look at the basics of Reinforcement Learning, a framework for describing and solving sequential decision-making problems. We introduce foundational ways these problems can be solved before moving on to more modern approaches using neural networks.

Lastly, we review the topic of Transformers, a family of state-of-the-art models that have become popular in Sequence Modelling. We introduce their architecture and a special class of Transformers called BERT models, which have become state-of-the-art in sequence classification.

## 2.1 Web Browsers

The most crucial role of a web browser is the rendering and execution of web content. Most browsers provide built-in JavaScript support, allowing for interactivity and dynamic elements within this web content. [19]. JavaScript engines are the part of a web browser responsible for the parsing and execution of JavaScript code. Before going more in-depth into the JavaScript engine itself, we will first introduce a web browser’s architecture to give more context about the environment in which these engines operate and why they are one of the most significant attack surfaces.

Many modern web browsers, like Google Chrome and Firefox, utilise a multi-process sandbox architecture [20, 21]. This design divides the browser into two processes: **parent** and **child**. The **parent** process handles the browser’s user interface, such as tabs and the address bar. It is a trusted process that manages the browser’s interaction with the operating system. In contrast, **child** processes are untrusted and intended to be used to run remote content. Due to this, **child** processes run in a sandbox environment that only allows access to essential system resources to prevent the system and user data from being compromised if an attacker exploits a vulnerability. One of the primary **child** processes is the **renderer** process which is responsible for parsing and executing/rendering several web content resources, including JavaScript.

The **renderer** process is one of the most vulnerable parts of the web browser because it regularly parses and executes code for third-party sources. Although this process is sandboxed, attackers may be able to exploit additional vulnerabilities in accessible unsandboxed processes to compromise the user’s system. In addition, a compromised **renderer** process could also perform Cross-site scripting attacks (XSS), in which the attacker gains access to authentication cookies for sensitive sites such as banks and emails. The JavaScript engine executes within this **renderer** process and has become a primary attack target to gain execution privileges inside the **renderer** process. Due to this non-trivial threat to the user’s system through the JavaScript engine, much research has focused on automatically detecting bugs at the engine level.

## 2.2 JavaScript Engines

In the pursuit of optimal performance, JavaScript engines have undergone significant advancements, employing various techniques to enhance code execution speed. As a result, major web browsers, including Google Chrome (V8), Mozilla Firefox (SpiderMonkey), and Apple Safari (JavaScriptCore), have developed their own JavaScript engines tailored to deliver superior performance and compatibility with web standards [3, 4, 5]. Although each engine differs slightly in implementation, they all follow the same basic design and components.

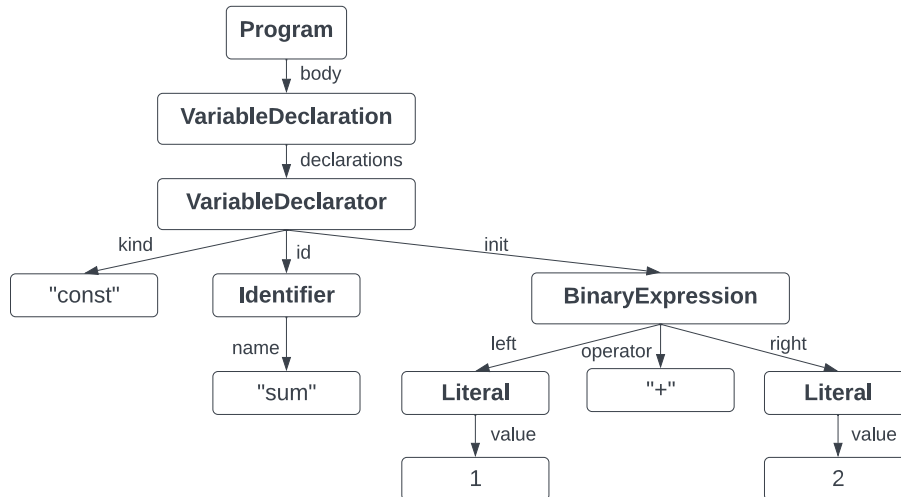


Figure 2.1: AST representation of a variable declaration statement

## 2.2.1 Design

### Parser

The parser plays a fundamental role in the JavaScript engine by tokenising the raw JavaScript code to construct an abstract syntax tree (AST). The abstract syntax tree (AST) is a hierarchical representation of the syntactic structure of JavaScript code and captures the relationships between various elements, such as statements, expressions, and declarations. The AST is an intermediate representation that facilitates efficient code analysis and transformations. For example, some carry out scope analysis on the AST and use this information for unused variable removal. To better understand the AST's structure, we can look at Fig. 2.1, which shows the AST representation of a variable declaration.

### Bytecode Generation

The engine then compiles the AST into bytecode, a low-level representation of the code the interpreter can consume and execute. The format of the bytecode differs between engine implementations, with some engines, such as SpiderMonkey, using a stack-based bytecode compared to V8, which uses a register-based bytecode. Listing 2.1 shows an example of the bytecode outputted by V8 for the variable declaration statement shown in Section 2.2.1.

```

1 LdaSmi [3]
2 StaCurrentContextSlot [2]

```

Listing 2.1: JavaScript bytecode compiled by V8 engine

Line 1 loads the value **3** into the accumulator register. Here we notice that there has been a slight optimisation in which the addition expression has been replaced by the results, as both the left and right operands were known integer values. Line 2 stores the value in the accumulator register to the context slot **2**. Context slots are used to store the values assigned to a variable.

Although, in this example, we see a small optimisation in the compiled bytecode, the JavaScript engine bytecode is generally largely unoptimised. For example, other

compilers may have generated no instructions for this code as we never reread the value in the variable. This lack of optimisation is a deliberate choice that stems from the primary objective of JavaScript engines: minimising load times. This is a crucial aspect that many users value in browser environments. Additionally, many code optimisation techniques heavily rely on type information, which poses a challenge given that JavaScript is a dynamically typed language. JavaScript variables can dynamically change their types during runtime, making it impossible to determine their types beforehand. Consequently, bytecode optimisations that require type information become impractical.

## Interpreter and Execution

The interpreter's main task is to take the compiled bytecode and execute it. The interpreter does this by taking the next instruction to be executed and passed to the appropriate **handler**. Handlers are small functions responsible for executing a particular bytecode instruction. In Listing 2.2, we see the V8 handler functions for the bytecode instructions we previously used. We see that the **LdaSmi** instruction gets the immediate value supplied to the instruction and then puts it in the accumulator. The **StaCurrentContextSlot** gets the value in the accumulator, parses the slot index and then stores the accumulator value to the slot index. We also see that both functions end with a call to **Dispatch**. This call looks up the next bytecode instruction in the dispatch table.

```
1 // LdaSmi <imm>
2 // Load an integer literal into the accumulator as a Smi.
3 IGNITION_HANDLER(LdaSmi, InterpreterAssembler) {
4     TNode<Smi> smi_int = BytecodeOperandImmSmi(0);
5     SetAccumulator(smi_int);
6     Dispatch();
7 }
8
9 IGNITION_HANDLER(StaCurrentContextSlot, InterpreterAssembler) {
10    TNode<Object> value = GetAccumulator();
11    TNode<IntPtrT> slot_index = Signed(BytecodeOperandIdx(0));
12    TNode<Context> slot_context = GetContext();
13    StoreContextElement(slot_context, slot_index, value);
14    Dispatch();
15 }
```

Listing 2.2: V8 bytecode handler functions

This interpretation step is considered relatively slow compared to executing pure machine code due to the unoptimised bytecode and the overhead of calling the appropriate handler for each instruction. When executing code which runs only a few times, the compilation overhead is not worth it for the execution speed up of the resulting machine code. However, when running a code block multiple times, the interpretation overhead can build up to a point at the trade-off become worthwhile. To allow such paths to be optimised, the interpreter also stores information about the input types to each bytecode operation.

## Just-In-Time (JIT) Compiler

The Just-In-Time compiler optimises frequently executed blocks of the code and compiles them to machine code [22]. It achieves this by using the bytecode from the parser and type information collected by the interpreter to optimise the bytecode and then compile it into machine code. The JIT compiler usually converts this bytecode to another internal representation which is more efficient for applying optimisations.

To better understand the kinds of optimisations that can occur, we will look at a technique called type speculation/specialisation, which many JavaScript engines employ. Type speculation involves making assumptions about the types of variables at specific points in the code and generating specialised machine code based on these assumptions. By assuming specific types, the compiler can avoid runtime checks and generate more efficient code tailored to those types.

To illustrate the concept of type speculation, let us consider the code provided in Listing 2.3. This code defines a **sum** function that takes two arguments and returns their sum. The code then calls this sum function in a loop for many iterations.

```
1 function sum(a, b) {  
2     return a + b;  
3 }  
4  
5 for (int i = 0; i < 1000000; i++) {  
6     sum(i, 1);  
7 }
```

Listing 2.3: Example code for type speculation in JIT compiler

Due to JavaScript being a typed language, several checks must be done on the variables involved in a binary expression before executing it. These checks are required as binary expressions are defined differently depending on the types of the left and right operands (Fig. 2.2) in the case of our example in Listing. 2.3, although we can see that the types of parameters **a** and **b** are integers, the program cannot know this until runtime. If the interpreter executes the code, it must execute all the checks shown in Fig. 2.2.

As the loop calls the add function many times, it will likely be labelled for optimisation by the JIT compiler. The JIT compiler uses type information from previous execution to speculate that the type of **a** and **b** will be integers. Predicting these types allows it to remove the type check previously mentioned from the compiled machine code. However, the types of variables are not always as simple as in this example. To safeguard against an instance where this type of speculation is wrong, it adds inexpensive type checks to assert that the type of a value is the one we expected. If not, we fallback to executing the code block with the interpreter.

### 2.2.2 Past Vulnerabilities

Many bugs in the infancy of JavaScript engines were common vulnerabilities, such as integer or buffer overflows. As JavaScript engines evolved and became more robust,

1. If `opText` is `+`, then
  - a. Let `lprim` be `? ToPrimitive(lval)`.
  - b. Let `rprim` be `? ToPrimitive(rval)`.
  - c. If `Type(lprim)` is String or `Type(rprim)` is String, then
    - i. Let `lstr` be `? ToString(lprim)`.
    - ii. Let `rstr` be `? ToString(rprim)`.
    - iii. Return the string-concatenation of `lstr` and `rstr`.
  - d. Set `lval` to `lprim`.
  - e. Set `rval` to `rprim`.
2. NOTE: At this point, it must be a numeric operation.
3. Let `lnum` be `? ToNumeric(lval)`.
4. Let `rnum` be `? ToNumeric(rval)`.
5. If `Type(lnum)` is different from `Type(rnum)`, throw a **`TypeError`** exception.
6. If `Type(lnum)` is BigInt, then
  - a. If `opText` is `**`, return `? BigInt::exponentiate(lnum, rnum)`.
  - b. If `opText` is `/`, return `? BigInt::divide(lnum, rnum)`.
  - c. If `opText` is `%`, return `? BigInt::remainder(lnum, rnum)`.
  - d. If `opText` is `>>>`, return `? BigInt::unsignedRightShift(lnum, rnum)`.
7. Let `operation` be the abstract operation associated with `opText` and `Type(lnum)` in the following table:
 

<code>opText</code>	<code>Type(lnum)</code>	<code>operation</code>
<code>**</code>	Number	Number::exponentiate
<code>*</code>	Number	Number::multiply
<code>*</code>	BigInt	BigInt::multiply
<code>/</code>	Number	Number::divide
<code>%</code>	Number	Number::remainder
<code>+</code>	Number	Number::add
<code>+</code>	BigInt	BigInt::add
<code>-</code>	Number	Number::subtract
<code>-</code>	BigInt	BigInt::subtract
<code>&lt;&lt;</code>	Number	Number::leftShift
<code>&lt;&lt;</code>	BigInt	BigInt::leftShift
<code>&gt;&gt;</code>	Number	Number::signedRightShift
<code>&gt;&gt;</code>	BigInt	BigInt::signedRightShift
<code>&gt;&gt;&gt;</code>	Number	Number::unsignedRightShift
<code>&amp;</code>	Number	Number::bitwiseAND
<code>&amp;</code>	BigInt	BigInt::bitwiseAND
<code>^</code>	Number	Number::bitwiseXOR
<code>^</code>	BigInt	BigInt::bitwiseXOR
<code> </code>	Number	Number::bitwiseOR
<code> </code>	BigInt	BigInt::bitwiseOR

Figure 2.2: ECMAScript definition of Binary Operations [23]

these bugs have been replaced with more domain-specific ones. This section looks at an example of a critical vulnerability in the V8 JavaScript engine.

**CVE-2019-5782**

CVE-2019-5782 exploits a bug which was present in the JIT compiler of V8.

As discussed previously, the JIT compiler optimises and compiles frequently executed bytecode into machine code. One of the optimisation components of this process is bounds check elimination.

When we index an array, the JavaScript engine (and most compilers) check this index to ensure that it is within the defined size of the array. This check is quite expensive. Therefore, if the engine can verify that this index is guaranteed to lie within the size of the array, then it can remove this bounds check.

The JIT compiler uses the type information collected along with a pre-defined set of rules imposed by the language to decide whether or not it can remove a bounds check. However, if there is a bug in implementing these checks, they can be exploited and lead to out-of-bounds accesses.

Given this, we can take a look at the Proof Of Concept (PoC) code for CVE-2019-5782 [24] shown in Listing 2.4. The `test` function is called many times with three parameters inside the for loop, which triggers the Just In Time compiler to optimise and compile the bytecode. Looking closer at the function, we see that it uses the value of `arguments.length` to decide on which index of the array to modify. The `arguments.length` is a function property that returns the number of arguments passed into the function. The JavaScript language sets a hard limit of **65534** on the number of parameters which can be passed into the regular call of a function. The JIT compiler will use this information to infer that the index calculated must always be 0 and hence, will remove the bound check for the array indexing on line 8.

```
1 oob index = 100000;
```

```

2 function test() {
3   let array = [1.1, 1.1];
4   let index = arguments.length;
5   index = index - 65534;
6   index = Math.max(index, 0);
7
8   return array[index] = 2.2;
9 }
10
11 for (let i = 0; i < 20000; i++) {
12   test(1,2,3);
13 }
14
15 print(trigger.apply(null, new Array(65534 + oob_index)));

```

Listing 2.4: PoC code for triggering CVE-2019-5782 [24]

However, the hard limit of 65534 does not apply when calling the function using the **apply** method. Therefore, after optimising the function by the JIT compiler, passing in more than 65534 parameters to the function using **apply** causes an out-of-bounds memory access. After achieving this out-of-bounds access, there are several methods which an attacker could use to execute remote code in the user's browser environment [24].

## 2.3 Fuzzing

Fuzzing is an automated software testing technique that involves (semi-)randomly generating many inputs to run through the System Under Test (SUT) [25]. The main goal of fuzzing is to generate inputs which stress the SUT such that it exhibits interesting behaviour. In this project, the SUT is the JavaScript engine; however, many fuzzing techniques used in compiler testing are still applicable. In compiler testing, interesting behaviour can take several forms, such as crashes, memory leaks or triggering internal assertions.

Compiler fuzzing has two main challenges: constructing valid test programs and constructing a diverse set of test programs. Constructing a valid program for a programming language is not trivial due to the wide range of syntactic and semantic structures between them. Constructing invalid programs is not helpful in compiler testing as the compiler tends to discard the program at early parsing stages. For example, a JavaScript program with a "return" statement outside a function body is classified as syntactically invalid, meaning the program will not reach the bytecode generation, interpreter or JIT compiler. Constructing diverse programs is also an essential part of fuzzing, as we want our inputs to stress different sections of the compiler to help increase code coverage and discover bugs in the system.

Given these challenges, most research has focused on two main types of fuzzing techniques: generation-based and mutation-based

### 2.3.1 Generation-based Fuzzing

Generation-based fuzzing involves the construction of inputs based on domain-specific knowledge or pre-defined grammar rules. This technique aims to generate inputs that conform to the syntax and structure expected by the SUT. We can break this type of fuzzing into two further sub-categories: grammar-directed and grammar-aided approaches.

**Grammar-directed fuzzers** take a language’s grammar rules as inputs and produce new inputs by starting at the root grammar rule. They do this by recursively selecting random productions of this rule until there are no more rules to expand or the program has reached a specific size of the program. In 6.1, we see an example of a basic grammar for the JavaScript language. In this case, our root rule would be the Program rule, and we would select a random production from the right-hand side of that rule. We then continue expanding all non-terminal rules until triggering the end conditions.

```
1 Program          ::= StatementList
2 StatementList    ::= Statement | Statement StatementList
3 Statement        ::= VariableDeclaration
4                   | IfStatement
5                   | WhileStatement
6                   | ...
7 VariableDeclaration ::= "var" Identifier "=" Expression ";"
8 IfStatement       ::= "if" "(" Expression ")" BlockStatement
9                   | "if" "(" Expression ")" BlockStatement
10                  "else" BlockStatement
11 WhileStatement    ::= "while" "(" Expression ")" BlockStatement
12 BlockStatement    ::= "{" StatementList "}"
13 Identifier        ::= <valid JavaScript identifier>
14 Expression        ::= ...
```

Listing 2.5: Rudimentary JavaScript Grammar

Although this method maintains the syntactic validity of the generated programs, it fails to guarantee semantic validity, which is extremely important for compiler testing.

**Grammar-aided** approaches also take a language’s grammar input and use heuristics to deal with these semantic issues. They do this by starting with template files with gaps in them to be filled using grammar. An excellent example of such a tool is CSmith [12], which has found plenty of success in finding bugs in the C compiler.

In addition to these well-established techniques, research has been undertaken into more unconventional generation-based fuzzers. One area of particular interest is Neural Network based program generation. Lee et Al. [26] proposed a JavaScript engine fuzzer called Montage, which uses a Neural Network based Language model to generate parts of a program. They achieved this by developing a novel formulation of an Abstract Syntax Tree as a sequence of sub-trees of depth one, which they call **fragments**. They then trained A Recurrent Neural Network language model to



predict the next fragment in the sequence given all the fragments before it. Finally, this language model was used to produce new inputs by taking existing JavaScript programs, removing a random subtree and then using the language model to generate a new subtree. This technique proved successful, finding new bugs in mainstream JavaScript engines such as JavaScriptCore and V8.

### 2.3.2 Mutation-based Fuzzing

Mutation-based fuzzing involves using a set of valid programs, often called seed inputs, and applying random modifications or mutations to generate new inputs. The idea behind mutation-based fuzzing is to explore the input space by making small changes to existing inputs and observing if this triggers any unexpected behaviour in the SUT. By only making minor changes to valid inputs, we are more likely to remain valid while potentially triggering unexpected behaviour. For general input fuzzing, mutations such as bit flips, value changes, data insertion or deletion, or other random transformations can be used. The advantage of using this technique is that it does not require prior knowledge of the input format and can apply to any input type. However, when used for the mutation of programs for compiler testing, using such mutations will likely lead to syntactically invalid inputs.

Research has focused on using more grammar-aware mutation techniques to combat this issue. These techniques usually parse the program into a different internal representation, such as an Abstract Syntax Tree (AST) [15, 13]. These internal representations abstract away a lot of the program's syntax, as we have seen in Fig. 2.1. By using this new mutation medium, we can maintain the code's syntactic validity.

### 2.3.3 Guided Fuzzing

Guided fuzzing is a technique that aims to intelligently guide the generation of inputs by leveraging feedback from the SUT's execution. This feedback helps identify inputs that exhibit interesting behaviour and guide the fuzzing process towards exploring different code paths. This technique is primarily used in mutation-based fuzzing as the feedback is more useful when making small mutations than after generating a whole program [27].

As with pure mutation-based fuzzing, the guided fuzzing process starts with an initial set of seed programs, which are then mutated and executed by the SUT. During the execution, feedback mechanisms, such as code coverage, memory usage or run times, are employed to determine how "interesting" an input is. Inputs that lead to previously unexplored code paths or trigger exceptional conditions are considered more interesting and prioritised for further mutation, while we discard ones which do not. The most common feedback used in guided fuzzing is code coverage due to its direct link with exploring new parts of the compiler. Coverage-guided fuzzing has been the main focus of research and development in the area of focus, with several fuzzers such as Superion [13], DIE[14] and Fuzzilli [15] all having success in the area.

## 2.4 Reinforcement Learning

Reinforcement learning is a framework for describing and solving sequential decision-making problems in which the goal is to maximise a cumulative reward by learning from previous interactions [28].

### 2.4.1 Markov Decision Process

The Markov Decision Process (MDP) provides a framework for describing a problem of learning from interaction to achieve a goal. We can define a Markov Decision Process as a tuple  $(S, A, P, R)$  where:

- $S$  is the set of all possible states
- $A$  is the set of all possible actions the agent can take
- $P(s'|s, a)$  is the joint probability of transitioning to state  $s'$  given we were in state  $s$  at the previous time step and chose action  $a$
- $R(s, s', a)$  is the immediate reward received after transitioning from state  $s$  to state  $s'$ , after choosing action  $a$

We can apply this definition to the problem of sequential decision-making as follows. We start by defining two parties: the agent and the environment. The agent is responsible for making decisions which maximise its rewards, while the environment is everything outside the agent with which the agent can interact. This interaction occurs at discrete time steps  $(t = 0, 1, \dots)$ . At each time step  $t$ , the agent is in a state  $S_t \in S$  that represents the environment, from which it chooses an action  $A_t \in A$ . As a consequence of this action, at the next time step, the agent is given a reward  $R_{t+1} \in R$  and is now in state  $S_{t+1} \in S$ .

We can then define the return  $G_t$  to be the reward accumulated by the agent after time  $t$ :

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where  $\gamma \in [0, 1]$  is the discount factor which balances the valuation of short-term rewards vs long-term rewards. A value close to 1 results in a far-sighted evaluation, and a value close to 0 results in a short-term evaluation.

The goal of our agent is to learn an optimal policy  $\pi : S \times A \rightarrow [0, 1]$  (i.e.  $\pi(a|s)$ ), which maximises the agent's expected rewards. In the next section, we can look at ways of finding/calculating an optimal policy  $\pi^*(a|s)$ , which maximises the agent's rewards.

### 2.4.2 Finding the Optimal Policy

We start by defining the value function  $v_\pi(s)$  and action value functions  $q_\pi(s, a)$  as follows:

$$v_\pi(s) = E[G_t | S_t = s, \pi] \tag{2.1}$$

$$q_\pi(s, a) = E[G_t | S_t = s, A_t = a, \pi] \tag{2.2}$$

The value function represents the expected discounted rewards given that we are in state  $s$  and follow a given policy  $\pi$ . Similarly, the action value function represents the expected discounted rewards given that we are in state  $s$ , have chosen action  $a$ , and follow a given policy  $\pi$  from that point onwards.

The value functions define a partial ordering on policies  $\pi$ . A policy  $\pi$  is said to be better than or equal a policy  $\pi'$  if its expected return is greater than or equal to that of  $\pi'$  for all states (i.e.  $\pi \geq \pi'$  if and only if  $v_\pi(s) \geq v_{\pi'}(s)$  for all  $s \in S$ ). There is always guaranteed to be at least one policy greater than or equal to all other policies. This policy is called the optimal policy  $\pi^*$  and can be defined as follows:

$$\forall s \in S : \pi^* = \arg \max_{\pi} [v_\pi(s)] \quad (2.3)$$

$$\forall s \in S, a \in A : \pi^* = \arg \max_{\pi} [q_\pi(s, a)] \quad (2.4)$$

Although the optimal policy is not unique, all optimal policies share the same optimal value function and action value functions, which are defined as follows:

$$v^*(s) = \max_{\pi} v_\pi(s) \quad \forall s \in S \quad (2.5)$$

$$q^*(s, a) = \max_{\pi} q_\pi(s, a) \quad \forall s \in S, a \in A \quad (2.6)$$

We can use these definitions to define the Bellman optimality equation, a foundational cornerstone for many Reinforcement Learning techniques. We can derive this by using the fact that the value of a state under an optimal policy must be equal to the expected return for the best action from that state:

$$\begin{aligned} v^*(s) &= \max_{a \in A} q_{\pi^*}(s, a) \\ &= \max_a E_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a E_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a E_{\pi^*}[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v^*(s')] \end{aligned} \quad (2.7)$$

We can carry out a similar derivation to find the Bellman optimality equation for  $q^*$ :

$$q^*(s, a) = \sum_{s', r} p(s', r | s, a) [r + \gamma q^*(s', a')] \quad (2.8)$$

If all transition probabilities  $p$  were known, we could solve the equation to find  $p^*$  as we have  $n$  equations (one for each state) and  $n$  unknowns. However, in many applied problems, these transition probabilities are unknown; hence, we require a different solution. Types of reinforcement learning algorithms that work in these unknown environments are known as Model-free learning algorithms.

### 2.4.3 Q-Learning

Q-learning is a prominent model-free learning algorithm which enables an agent to learn the optimal action-value function,  $q^*(s, a)$ . Q-learning employs a table, often called a Q-table, to store and update action-value estimates based on observed rewards and transitions in the environment.

Q-learning falls under the category of Temporal Difference (TD) algorithms, which learn from experience by updating value estimates based on the observed differences, or temporal differences, between predicted and observed rewards. The Q-table is initialised arbitrarily and refined over time through this iterative learning process. Additionally, Q-Learning is an off-policy algorithm, which uses a different behaviour policy  $\pi_B$  to explore the environment, rather than using the learned policy directly.

At each time step, the agent interacts with the environment by selecting an action, observing the resulting reward and new state, and updating the corresponding entry in the Q-table. The Q-learning algorithm updates the action-value estimates using the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right] \quad (2.9)$$

Here,  $\alpha$  denotes the learning rate, which controls the weight given to the newly observed information, and  $\gamma$  is the discount factor. This equation is an extension of the Bellman Optimally Equation shown in Equation 2.8. To ensure convergence to the optimal action-value  $q^*(s, a)$ , our behaviour policy, which we use to explore the environment, must guarantee that each state-action pair is visited infinitely often, i.e.  $\pi_B(s, a) > 0 \quad \forall s \in S, a \in A$ .

### 2.4.4 Deep Q-Learning (DQN)

While Q-learning works well for problems with a small number of discrete states and actions, it becomes impractical for complex and continuous state spaces. This impracticality occurs as Q-learning requires the storage of the Q-value for every state-action pair  $(s, a)$  in an array of dimensions  $|A| \times |S|$ , which is impossible for many real-world problems. Deep Q-learning addresses this limitation by utilising deep neural networks to approximate the Q-value function, known as a Deep Q-Network (DQN) [16]. The network takes the state as input and outputs Q-values for each action. The learning process involves updating the network parameters to minimise discrepancies between predicted Q-values and observed rewards.

As Deep Q-Networks are an extension of Q-learning, they are also an off-policy method that employs a different policy  $\pi_B$  to explore the environment and gather experience for learning the optimal policy. Ensuring sufficient exploration of the state space is crucial for adequate learning coverage. In order to strike a balance between exploration and exploitation, an epsilon-greedy approach is commonly used as the behaviour policy in DQNs. During training, the agent selects the action with the highest Q-value with probability  $(1 - \epsilon)$  (exploitation) and chooses a random action with probability  $\epsilon$  (exploration), where  $\epsilon \in [0, 1]$ . By defining our policy this way, we can change the amount of exploration and exploitation of our agent by changing our  $\epsilon$  value over time. By starting with a high epsilon, we can explore the

environment and collect diverse samples from which we can learn. We then start decaying  $\epsilon$  over time as the agent learns to allow it to slowly leverage its accumulated knowledge while occasionally forcing it to explore.

The training of a Deep Q-Network on its own is very unstable. One technique often employed to combat this is introducing a second Q-network called the target network. The target network is used to evaluate the target Q-values  $Q(S_{T+1}, s')$  in Equation 2.9 and is only updated with the weights of the primary Q-network at regular intervals. This approach helps to provide a more stable and reliable target for next-state value estimation during training.

The loss function for DQN is defined as the mean squared error between the predicted Q-values and the target Q-values. Given a transition tuple  $(S_t, A_t, R_{t+1}, S_{t+1})$ , the loss function is given by:

$$\mathcal{L}(\theta) = E \left[ (R_t + \gamma \max_{a'} Q_{\text{target}}(S_{t+1}, a'; \theta^-) - Q(S_t, A_t; \theta))^2 \right] \quad (2.10)$$

where  $\theta$  represents the parameters of the primary (policy) Q-network,  $Q(S_t, A_t; \theta)$  is the predicted Q-value for the current state-action pair,  $Q_{\text{target}}(S_{t+1}, a'; \theta^-)$  is the target Q-value calculated using the target network with parameter  $\theta^-$ ,  $R_t$  is the observed reward at time  $t$ ,  $\gamma$  is the discount factor, and  $E$  denotes the expectation over a batch of transitions.

Another technique used to stabilise the training process is Experience Replay. Due to the highly correlated nature of successive transitions, training the neural network on transitions as we receive them would result in a very high gradient in one direction. Using this training method would lead to the network constantly over-correcting to the space in the environment we are currently in instead of generalising to the whole environment. Experience replay is a technique that involves storing observed transitions in a buffer and drawing a random batch of transitions to train our network, making our training samples more diverse and less likely to be correlated.

## 2.4.5 Double Q-Learning (Double DQN)

While Deep Q-Learning has shown remarkable success in dealing with complex and continuous state spaces, it can still suffer from overestimating action values. Overestimation occurs when the Q-network assigns higher Q-values to specific actions than their true values, leading to sub-optimal policies. Double Q-Learning addresses this issue by decoupling the action selection and value estimation processes [29].

Like DQNs, Double DQN employs two separate neural networks: a policy network and a target network. The policy network is responsible for action selection; it takes the current state as input and computes the Q-values for each action. However, instead of directly using the Q-network's estimates to select the action, the target network is used to evaluate the value of the chosen action. Introducing independence in the estimation process prevents the Q-network from overestimating the action values, a known problem in DQNs [29].

To update the Q-network, we start by selecting the best action using the Q-network's estimates:

$$A_{t+1} = \arg \max_{a'} Q(S_{t+1}, a'; \theta) \quad (2.11)$$

We then use the target network to estimate the value of this state-action pair using  $Q_{\text{target}}(S_{t+1}, A_{t+1}; \theta^-)$ . Using this, we can modify our loss function from Equation 2.10 as follows:

$$\mathcal{L}(\theta) = E \left[ (R_{t+1} + \gamma Q_{\text{target}}(S_{t+1}, A_{t+1}; \theta^-) - Q(S_t, A_t; \theta))^2 \right] \quad (2.12)$$

The target network can be updated either through soft updates, where the target network slowly tracks the changes of the Q-network or through hard updates, where we update the weights of the target network with a copy of the main Q-network’s weights at fixed intervals.

## 2.5 Transformers

Since their introduction by Vaswani et al. [30], transformers have become the go-to architecture for sequence modelling. They vastly outperformed Recurrent Neural Networks, the previous state of the art, in several tasks such as sequence classification and machine translation. Although a lot of the recent interest in transformers has mainly been in the field of Natural Language, it is essential to note that we can apply transformers to any sequential data. We will first look at the general architecture of the transformer model and then take a closer look at a class of encoder-only models called BERT (Bidirectional Encoder Representations from Transformers) models.

### 2.5.1 Architecture

At the core of the Transformer architecture is the concept of attention, which was first introduced by Bahdanau et al. [31]. The general idea behind attention is to allow the model to focus on different parts of the input sequence while processing it, enabling it to capture the relevant information more effectively. We can see the whole model architecture of the transformer in Figure 2.3. The left side of the diagram is the encoder, which takes in the input sequence and outputs a hidden vector representation for each element. On the right side of the diagram is the decoder generates its output auto-regressively by taking all the outputs it has decoded so far, along with the hidden representation of the input sequence from the encoder to generate the following output element. This project only uses the encoder side of this model, and hence, in the rest of this section, we will focus on the layers involved in this task.

#### Input Embedding

In the Transformer model, input embedding is the initial step in processing sequential data. The input data to the transformer should be a sequence of integer ids, each corresponding to a specific item. For example, this may correspond to a word or token in natural language processing. The input embedding layer is responsible for converting this discrete input into a dense continuous vector representation which the transformer layers can use to form the output contextual representation for each element. These embeddings are typically learned during the training of the whole transformer model and are domain specific to the current task.

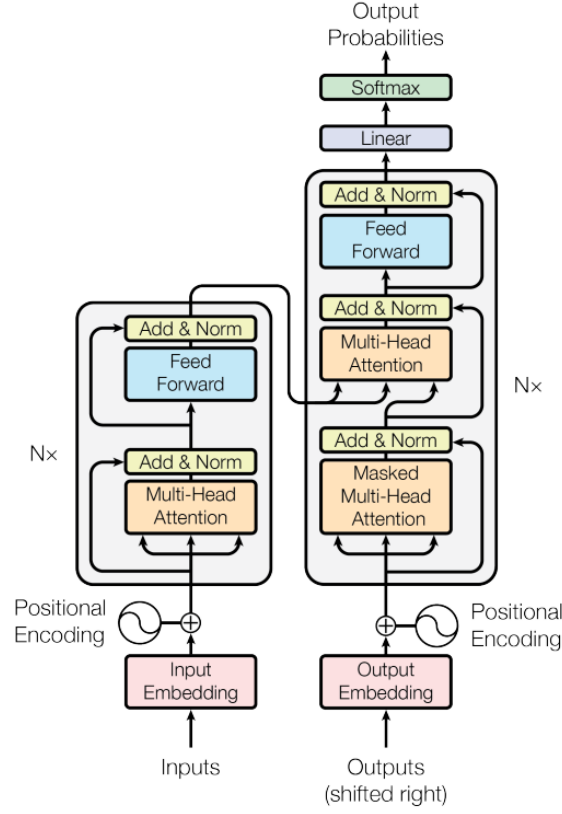


Figure 2.3: Encoder-Decoder Transformer Architecture [30]

### Positional Encoding

The transformer model is inherently position invariant, i.e. the order of the input sequence has no impact on the input. However, this is not desirable as the order of the sequence elements is essential for sequential modelling. For example, in English, the order of the words in a sentence matter: "the boy ran after the dog"  $\neq$  "the dog ran after the boy". Therefore, we must inject information about an element's absolute or relevant position in the sequence into our inputs. To achieve this, "positional encodings" are added to element embeddings at the start of the encoder and decoder stacks. The original paper introduced a sinusoidal positional encoding:

$$PE_{(\text{pos}, 2i)} = \sin(\text{pos}/10000^{2i/d_{\text{model}}})$$

$$PE_{(\text{pos}, 2i+1)} = \cos(\text{pos}/10000^{2i/d_{\text{model}}})$$

where "pos" is the element's position in the input sequence and  $i$  is the dimension of the embedding vector.

### Scaled Dot-Product Attention

Scaled Dot-Product attention calculates the relevance between elements in the input sequence. The input to this layer consists of queries and keys of dimension  $d_k$  and values of dimension  $d_v$ . We compute the dot products of each query with all keys, which acts as our similarity score. We then divide each of these values by  $\sqrt{d_k}$  and then apply the softmax function to obtain the weighting of each value to be used to

form the output of the query:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The division by  $\sqrt{d_k}$  produces a less peaky output from the softmax function, which avoids the risk of extremely small gradients during backpropagation.

### Multi-Head Attention

Instead of performing attention once of the input of dimension  $d_{model}$ , multi-head attention performs attention  $h$  times with different projection matrices for each head:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \quad (2.13)$$

$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \quad (2.14)$$

### Position-wise Feed-Forward Networks

Along with the multi-head attention sub-layer, each layer contains a fully connected feed-forward network applied separately and identically to each position. This sub-layer normally consists of two linear transformations with a ReLU activation in between:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2.15)$$

## 2.5.2 BERT models

The BERT (Bidirectional Encoder Representations from Transformers) family of models are an encoder-only transformer model [32]. The goal of this model is to map an input sequence  $(x_1, \dots, x_n)$  to a sequence of representations  $\mathbf{z} = (z_1, \dots, z_n)$ . It does this by removing the limitation that an element in the sequence can only attend to elements before, which was included in the initial transformer model. This technique enhances the vector representations for the sequence and elements, significantly improving these models' performance on the sequence and element classification tasks.

Along with the elements/tokens in the vocabulary, BERT also makes use of additional special tokens:

- '[CLS]' - Classification token, which appears at the start of every input sequence. The output vector of this token can be used as a contextual representation of the entire sequence once trained on downstream tasks.
- '[PAD]' - Padding token used when batching inputs of varying lengths together
- '[EOS]' - End of sequence, which appears as the last element of every input sequence
- '[MASK]' - Mask token used to represent tokens in the input sequence which have been replaced. Used in the Masked Language Modeling training objective



## Training Objective

The training objective of BERT models involves two key components: masked language modelling (MLM) and next sentence prediction (NSP).

**Masked Language Modeling (MLM)** In the MLM task, we randomly mask a certain percentage of the input tokens and train the model to predict the original masked tokens based on the context of the surrounding tokens. This objective encourages the model to learn bidirectional representations that capture contextual information from each token’s left and right contexts. During training, we mask approximately 15% of the input tokens. Out of these masked tokens:

- We replace 80% of them with a specific [MASK] token.
- We replace 10% of them with a randomly chosen token from the vocabulary.
- We keep 10% of them unchanged.

Next, we train the model to predict the original identity of the masked tokens. This training task enables BERT models to understand the relationships between words in the input sequence.

**Next Sentence Prediction (NSP)** The NSP task, initially introduced in the BERT model, involves predicting whether a given pair of sentences appear consecutively in the original text or paired randomly. This task helps the model capture relationships between sentences and understand the context of a given sequence.

However, subsequent research, such as RoBERTa, has shown that NSP training is unnecessary, which means we can exclude it without sacrificing the performance of the BERT model. RoBERTa removed the NSP task and focused solely on the MLM task during pretraining, achieving improved performance by optimising the hyperparameters and training setup.

**Fine-Tuning** Once BERT has been pre-trained using the above objectives, we must fine-tune downstream tasks. We introduce an additional neural network for element-level classification that takes that element’s output representation as an input and outputs the desired classes. As previously mentioned, we use the output representation of the '[CLS]' token for sequence-level classification by passing this representation to the classification network.

# Chapter 3

## Related Research

The fuzzing of JavaScript engines is heavily researched, with several papers on new techniques/methodologies published regularly. Comparatively, very little research has taken place into using reinforcement learning for fuzzing of any kind. In this chapter, we briefly overview DIE, one of the tools at the forefront of JavaScript engine fuzzing. We follow this by reviewing a paper by Li et Al. [33], that proposes a tool called FuzzBoost which uses reinforcement learning to carry out mutation-based fuzzing on the C compiler. To the best of our knowledge, this paper is the only research that has been undertaken into reinforcement learning-based compiler fuzzing.

### 3.1 DIE

DIE is a novel guided mutation-based fuzzer for JavaScript engines introduced in a paper by Park et Al. [14]. Their paper observes that most JavaScript engine mutation-based fuzzers start with a high-quality input corpus, such as an engine's test suite or PoCs of previous vulnerabilities. Such inputs usually have a very intentional structure that stresses specific code paths in the JavaScript engine. An example of such a structure can be shown in the Proof of Concept code we presented in Section 2.2.2. This code used a for loop to repeatedly call a function, to cause its compilation, which eventually led to the vulnerability. In addition to the structure of the code, the authors also found that it was important to preserve types of variables and objects within the program for similar reasons.

However, if a mutation-based fuzzer using random mutations were employed, it would likely remove or replace these crucial structures and types in the code, rendering the purpose of the seed file ineffective. Building on this observation, the authors developed their fuzzer DIE, which employs "aspect-preserving" mutations to preserve specific "aspects" in the fuzzing input. As we have discussed, these aspects can be either type-based or structure-based.

DIE uses an Abstract Syntax Tree (AST) based approach for its mutations. Using an AST representation allows it to maintain syntactic validity throughout mutation. DIE also uses the AST to store type information about variables and objects in the code. In order to carry out a mutation, DIE starts by selecting a random sub-

tree of the AST which does not have a structural purpose, such as an expression statement. This sub-tree is then replaced with one of the same type, built by their sub-tree generator. DIE uses other mutations to add statements to random code blocks within the program and a mutation to add variable declarations to a random location.

DIE then uses coverage feedback from the execution of the mutated sample to decide whether to add it to its corpus of seed files or discard it. After execution is complete, DIE reverts the mutation on the selected fuzzing input so that it can be reused for another round of mutation.

In addition to these techniques, DIE has also been implemented in a way, to allow for running it in a distributed environment. Doing so allows their fuzzer to leverage multiple processes or even machines to run several instances of their fuzzer in parallel, dramatically increasing their throughput.

Finally, they present their results, which include discovering several bugs in the latest versions of ChakraCode and JavaScriptCore at the time. In addition to this, they compared DIE to other JavaScript engine fuzzers (Superion and CodeAlchemist). They showed that DIE achieves a 3% increase in covered path discoveries over these tools.

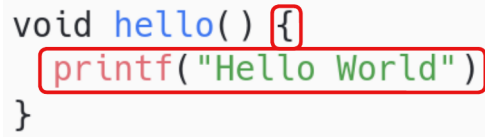
## 3.2 FuzzBoost

Li et Al. [33] propose a tool called FuzzBoost, which is a fuzzing tool for the C compiler which generates its outputs by performing multiple mutation steps on a seed program at the source code level. FuzzBoost uses reinforcement learning techniques to guide the mutation process and uses feedback from the compiler to learn the optimal policy for generating new test inputs. In particular, they use a Deep Q-Network for their action selection. We find that specific implementation details are sparse in their paper, and no code repository has been provided. However, we will try to give a brief overview of their ideas.

FuzzBoost starts by selecting a random program for mutation. It then selects a random sub-string, called the extraction window, of this program with a pre-defined token length. This extraction window is what they define as their state. In Figure 3. we give our interpretation of an extraction window of size five.

They use a word embedding model to convert this window into a vector representation which can then be passed to the DQN for action selection. They start by defining token-level mutation actions such as insertion, replacement, re-ordering, deletion and replication. These actions act on or after the last token in the state. Additionally, they define some movement actions that allow the window to either be shifted by one token to the left or right or be increased by one token to the left or right.

Although the authors displayed promising results in their evaluation, it lacks testing of these techniques in the real world. Firstly, the only comparison provided in the paper is between FuzzBoost and AFL, in which they investigate the amount of coverage achieved vs the number of tests generated. Although this emphasises that reinforcement learning-based approaches can learn patterns that increase coverage



```
void hello() {  
    printf("Hello World")  
}
```

Figure 3.1: An example of extraction window used in FuzzBoost

in fewer steps than random fuzzers, this metric does not capture the technique’s real-world applicability. Neural network-based reinforcement learning-based approaches have a considerable overhead compared to other, more conventional techniques, as they need to optimise network parameters. Therefore, evaluating the performance of reinforcement learning-based fuzzers in a fixed-time setting is essential to ensure a fair and useful comparison.

We also perceive several disadvantages to FuzzBoost’s fuzzing technique stemming from their base decision to apply mutations at a source code level. Due to their use of this representation, their mutations are less powerful as they only act on one token at a time. To show this, we can look at Figure 3.1 again. Assuming we started with the shown window, it would take at least eight actions to begin modifying the **void** return type of this function. Only allowing for movement and modification of one token at a time severely limits the fuzzer’s effectiveness.

# Chapter 4

## Design

In the limited research conducted in reinforcement learning enhanced compiler fuzzing, the only technique explored so far is a mutation-based technique which operates at the source code level.

This chapter aims to adapt this idea to make the mutations operate on the AST representation of the source code. We propose a novel formulation of mutation-based fuzzing as a Markov Decision Process (MDP), which uses an AST representation of the source code. Formulating the problem as an MDP allows us to apply a wide range of Reinforcement Learning techniques, which have been developed to find the optimal solution to these problems.

Additionally, we present our design for the environment and agent. We start with an overview of the initialisation state of the environment followed by a walk-through of how it processes actions it receives from the agent. Our environment uses ASTs as its state representation; however, most modern reinforcement learning techniques are neural network based, which require a numerical representation of the state space as input. To facilitate this, we also propose a novel approach to extracting a vector representation of an AST by building upon ideas of creating a sequential representation of an AST introduced in a paper by Lee et. Al [\[26\]](#).

## 4.1 Fuzzing as a Markov Decision Process

### 4.1.1 Action Space

Our agent aims to select actions  $A_1, A_2, \dots$ , which modifies the input program to produce interesting behaviour. We have split our actions into movement and mutation to allow the agent to select which part of the program to mutate. Movement actions allow the agent to move to a different AST node, while mutation actions act on the currently selected AST node. We also added an End action, which allows the agent to stop the current episode of fuzzing if it thinks it can no longer progress. Here is the complete list of actions which the agent can take:

- **Move Down** - Agent moves down to a random child of the current AST node. If the agent is at a leaf node, it stays at the same node
- **Move Up** - Agent moves up to the direct parent of the current AST node. If the agent is at the root, it stays at the same node.
- **Move Left** - Agent moves horizontally to the node to the left of the current AST node. If the current node is an only child, it stays at the same node.
- **Move Right** - Agent moves horizontally to the node to the right of the current AST node. If the current node is an only child, it stays at the same node.
- **Add** - Insets a new node as a child of the current node. The added node becomes the new selected node. No mutation occurs if the current node lacks a field supporting adding elements.
- **Replace** - Replaces the current node with a new node of the same supertype. The new node becomes the new selected node. For example, if the current node is a `BinaryExpression`, it can be replaced with any `Expression` node.
- **Remove** - Removes the current node from its parent. The parent of the removed node becomes the currently selected node. If removing the node would cause a syntactic error, no mutation occurs.
- **Modify** - Changes the operator of `UnaryExpression`, `BinaryExpression` and `AssignmentExpression` nodes. If the current node is not one of these, then no mutation occurs.
- **End** - Ends the current episode of fuzzing on the selected input test case

Initially, we did not include left and right actions in our action space, as at first glance, the agent can move up and down to achieve the same effect. However, the down action selects a random child of the current node to move to. Due to this randomness, the agent could get stuck in a loop of moving up and down to try and reach a specific node. Therefore, providing the left and right instructions gives the agent a deterministic way of moving between children.

### 4.1.2 State Space

To decide on its action, the agent requires information about the current node it can act on and the surrounding nodes it can move. We define the state to contain local and global components. The local component is the subtree of the program,

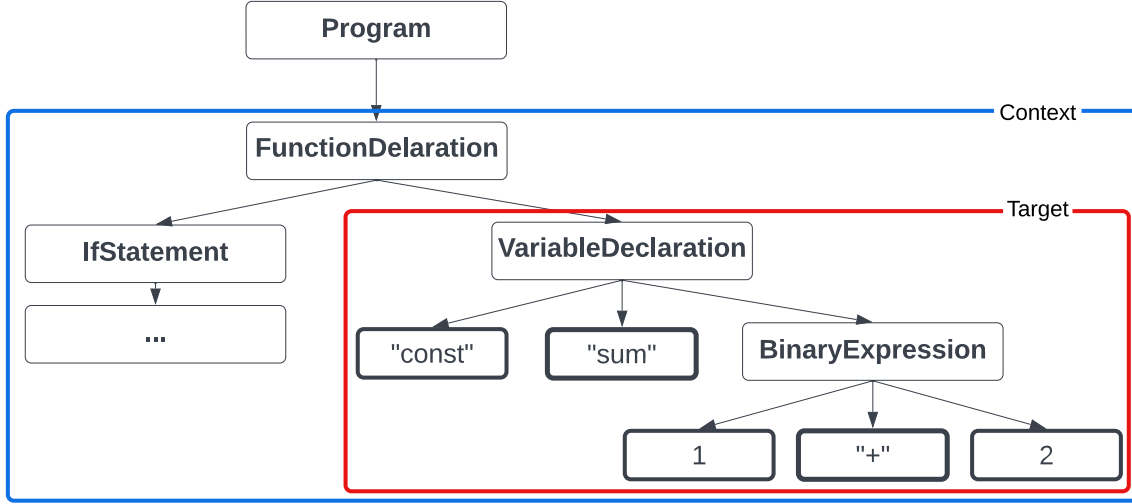


Figure 4.1: Visual representation of context and target states

with the root node being the currently selected node. The global component is the subtree, with the root node being the closest parent scope node to the currently selected node. We define a scope node as any node that creates a new scope for all nodes inside it. These nodes are Program, FunctionDeclaration, BlockStatement, and ClassDeclarations. We call the global component the **context** state and the local component the **target** state.

In order to give a better understanding of this representation, we can look at Fig. 4.1. We have used a simplified version of the AST previously shown for space reasons. The currently selected node is the VariableDeclaration node. Therefore, as shown in the figure, our **target** state is a subtree with the VariableDeclaration node as its root. Using the **target** state by itself would not give the agent any information about other nodes it can select instead.

As the agent has navigated to this node from the root of the program, it has already considered the other nodes. However, we note that the agent has no memory of how it gets to its current state due to the definition of the Markov Decision Process. It must decide on the next action solely based on its current state. This is known as the Markov Property, which forms the basis of the Markov Decision Process.

The FunctionDeclaration node is the closest parent scope to our selected node in this case. We use the subtree rooted at the FunctionDeclaration node as our **target** state shown in the figure. An alternative option for the **target** state may have been to use the whole program. However, we believe this would be too wide a scope to be useful to the agent.

### 4.1.3 Episode Termination

There are several conditions for which early termination of an episode occurs:

- The agent selects the **End** action
- The agent has triggered a crash

- The agent has carried out the maximum amount of mutations allowed in an episode

We limit the number of mutation actions in one episode to encourage the agent to develop the smallest sequence of mutations to trigger interesting behaviour and ensure that the agent does not get stuck mutating a test case forever. We investigate the optimal value for this parameter in Section 6.2.

#### 4.1.4 Rewards

The ideal overall goal of the agent is to trigger a crash in the JavaScript engine. However, this goal is likely to be very sparse as it may require a perfect sequence of 20+ mutations to create such a program. To combat this, we can supplement the sparse reward with more frequent rewards, which direct the agent towards this overall goal. We chose code coverage as our intermediate reward to encourage the fuzzer to learn sequences of mutations that stressed new parts of the JavaScript engine. Specifically, we reward the agent for an increase in coverage on the test input within an episode and reward it more if it achieves an overall coverage by every test case. During testing, we also found it essential to penalise fuzzer if its action does not cause any change in the state due to the conditions in 4.1.1.

Here we give an example of what such a rewards function may look like as this differs slightly from what we needed to do in practice:

$$R = \begin{cases} 3 & \text{if a crash is triggered} \\ 2 & \text{if test suite coverage is increased} \\ 1 & \text{if test case coverage is increased} \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

## 4.2 Environment

Given our MDP formulation of AST mutation-based fuzzing, we can now define the elements that make up our environment.

### 4.2.1 Environment Initialisation

We must first initialise the environment with the required data before interacting with the agent. The environment requires both an input corpus of seed files which it will use as its base for mutation, and a list of subtrees for each type of AST node. We use subtrees for mutation actions such as replace and add, which require a new node for their operation. The subtrees must be diverse as they are the primary source of new nodes during mutation. This requirement is in place due to lacking a generative element in our fuzzer. We decided to use this approach as we could generate a highly diverse set of subtrees with over nine million elements, sufficient to simulate any subtree we could have generated manually.

In addition to this data, the environment uses several parameters influencing fuzzing. In particular, we allow for the following parameters to change:

- Max number of mutations per episode



- Number of statements allowed in a program before applying a penalty to the reward
- Max number of statements allowed in a program before ending the episode
- Max number of times a program in the corpus can be selected for mutation without increasing coverage

We will discuss more about how these parameters affect different parts of the environment in the following sections.

The environment stores both episode-level and fuzzing-level data, as these are both required in different sections of the environment. Episode level data includes the current state the agent is in and two flags which indicate whether or not the agent has improved the current test cases coverage or overall coverage, respectively. Fuzzing level data is persistent over all episodes and contains cumulative data about the coverage achieved by all test cases generated so far. The environment uses this data to calculate whether an action increases total coverage.

At the start of each episode, the environment is reset, which involves selecting a new program from the corpus and resetting any episode-level data. As mentioned in Section 4.1, the overall state comprises both a context state and a target state which are subtrees of the program. We initialise both values to be the root of the program’s AST. The environment then takes an action from the agent and carries out the following steps:

1. Applies the action to the target state node in the AST
2. Executes the program if a structural change has been made to the AST (i.e. move or failed actions do not change the program, and hence, we do not need to re-execute the program)
3. Calculates the reward based on the success of the action, execution result (crash and coverage)
4. Returns the reward, new state and whether or not an end condition has been met

## 4.2.2 Applying an action

Although applying the actions in Section 4.1.1 may seem simple, much work is done behind the scenes to update the context state and maintain the sample’s semantic validity during mutation actions.

### Tracking Context State

We note that the context state can only change due to a vertical movement action (up and down). A naive approach to get the context state from the target state is trailing up the tree from our target state until reaching one of the scope nodes defined in Section 4.1. However, this operation would have to be done every time we select a move-up or move-down action, which is inefficient.

Instead, we use a stack-based approach which ensures that the relevant context state is always on the top of the stack. At the start of the episode, both our target and

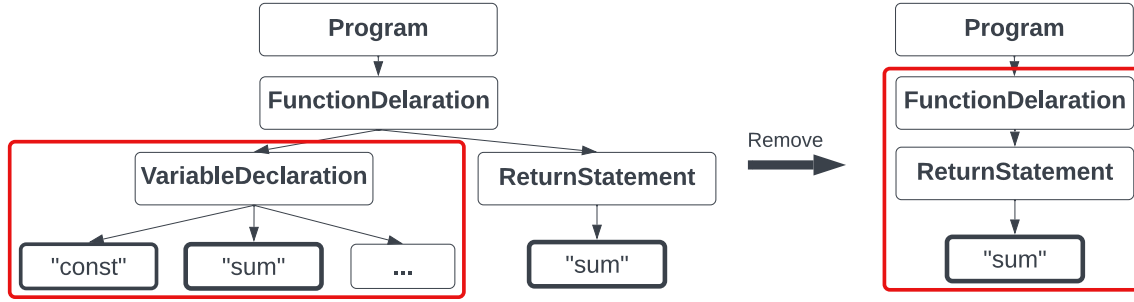


Figure 4.2: An example of a semantically invalid program as a result of Remove action

context states are the root node. Therefore, we start with the root node on the top of our context state stack. As we apply vertical movement instruction, we change the target state as usual, but we only modify the context state stack under the following conditions:

- Down action - If the target state before applying the action is a scope node, then we add the pre-action node to the top of the stack
- Up action - If the target state after applying the action is the same as the element on the top of the stack, we pop off the element. Ignore if the Program node is at the top of the stack.

By applying these conditions, we can ensure that the top of the stack is always the most recent scope node.

### Maintaining semantic validity

Carrying out the mutation actions alone will likely break the code's semantic validity. To illustrate this, we use Fig. 4.2, which shows the effect of a remove action on an example program. The VariableDeclaration statement is removed from the function due to the action. We can see that the resulting program is now semantically invalid as the **sum** variable in the ReturnStatement is no longer defined. The same can apply when adding or replacing code with random subtrees, as declarations in our current program may be changed, and declarations used in the inserted subtrees may not be present. To mitigate this, we added a scope analysis check after every mutation action to ensure we maximised the code's semantic validity.

This check carries out a complete scope analysis of a given AST, storing a list of available variables, functions (and the number of parameters) and classes at every AST node. After scope analysis, we apply a fixing algorithm which traverses the AST and fixes any references to functions, variables or classes that are not in scope. Carrying out scope fixing ensures that the code passes the semantic checks of the JavaScript engine, allowing it to stress deeper parts of the engine. If no available replacements exist in the scope, as in Fig. 4.2, and the value is a variable, we replace it with a random Literal node. If the value is a function or class, we replace them with a random built-in function or object name.

### 4.2.3 Execution

Execution of the program occurs if the action given by the agent is successfully applied and the action is a mutation action. If the action is unsuccessful, we skip this section as the AST has not changed. We must convert the AST into JavaScript code to execute the program in a JavaScript engine. A lifter iterates through the AST nodes and generates its associated code. From the execution, we receive the execution status (return code or timeout) along with branch coverage of the JavaScript engine achieved by the input. We will discuss how we instrument the engine to achieve this in Section 5.3.

### 4.2.4 Reward

The environment uses the information we have collected up until now to decide on the reward for the agent. In an ideal world, we would be able to use a reward function very close to the one presented in 4.1.4; however, during our testing, we found that this reward was too sparse for the agent to learn all the intricacies of the environment and action space.

To combat this, we added additional rewards to those found in Section 4.1.4 for the following conditions:

1. Action fails: -1
2. Test case coverage decreases: -0.1
3. Episode ends without increasing total coverage: -2
4. Program has more statements than a threshold:  $R + \min(0, 1 - \frac{\text{statements}}{\text{threshold}})$

Condition 1 aims to help the agent learn which actions apply to while nodes. We added this condition after observing that the agent tended to try to go up from the root node or down at a leaf node when this reward was not present. Adding this condition also helped with mutation actions such as Add, Remove and Modify, which only work on certain node types.

Condition 2 acts as a minor penalty for decreasing the coverage of the current test case. We added this penalty after several tests in which doing so helped the agent's performance. We tuned this penalty to be relatively small as we still want the agent to explore paths that may initially decrease coverage but then increase again.

Condition 3 aims to penalise the agent for not increasing total coverage within the span of the episode. Similarly to condition 2, we added and tuned this reward based on the agent's performance.

Condition 4 applies a penalty to all rewards, depending on the number of statement nodes in the current program. The penalty is 0 if the number of statements in the current program is within the threshold limit. However, when the program goes over this limit, it receives a negative penalty proportional to the number of statements it is over by. We added this penalty because the agent learned that it could just add an infinite amount of statements of the program and continually increase its coverage and hence reward. We used the number of statements as our metric as we wanted the agent to continue creating large nested expressions.

### 4.2.5 Corpus Expansion

If we generate a sample that increases the total coverage, we add it to the corpus, as it must hit a unique branch of the JavaScript compiler. By doing this, we allow future mutations to stress these unique branches, increasing the likelihood of finding a crash or increasing our coverage further. We also add a condition to our environment to remove any samples selected for mutation multiple times and not increase the total coverage in any of these episodes.

### 4.2.6 Return to agent

The environment returns the following information to the agent:

- New State - the state reached as a result of the action given by the agent
- Reward - the reward achieved as a result of the action given by the agent
- Done - Boolean flag to represent whether or not the end condition has been met

## 4.3 Agent

The agent’s goal is to take the current state of the environment as input and choose the optimal action that maximises its reward. Our agent uses a Double DQN model as it has had high success on several sequential decision-making problems [29]. Although better/alternative techniques such as Rainbow DQN [17] or Proximal Policy Optimisation [34] exist, they take more time to train/run [35]. As our goal is to maximise throughput while still aiming for convergence of our agent to some optimal policy, we chose the lightweight Double DQN architecture. In order to make use of a Double DQN model to choose our actions, we first require a vector representation of both our **target** and **context** states which are in the form of an AST.

### 4.3.1 AST Vector Representation

Several techniques exist for modelling the structure of AST, such as tree-based and graph-based neural networks. However, for this project, we took inspiration from Lee et al. [26], which introduces a technique called **fragmentation**, which reformulates the AST as a sequence of depth one subtree. We chose this option because Montage has proven this technique successful in modelling and generating JavaScript code from an input fragment sequence. Therefore, by adapting this idea for sequence classification and improving their architecture, we aim to produce a good representation of our JavaScript AST.

#### Fragmentation

We start by defining AST  $T$  as a tuple  $(N, E, n_0)$ , where  $N$  is the set of nodes in  $T$ ,  $E$  is the set of directed edges (from parent to child) in  $T$ , and  $n_0$  is the root node of  $T$ . We define the  $C(n_i)$  to be the immediate children of node  $n_i$ , i.e. all outgoing edges from  $n_i$ . We denote  $T_i$  as a  $T$  subtree with root  $n_i$ . Finally, we define a **fragment** to be a subtree with a depth of one. Formally this can be written as follows:

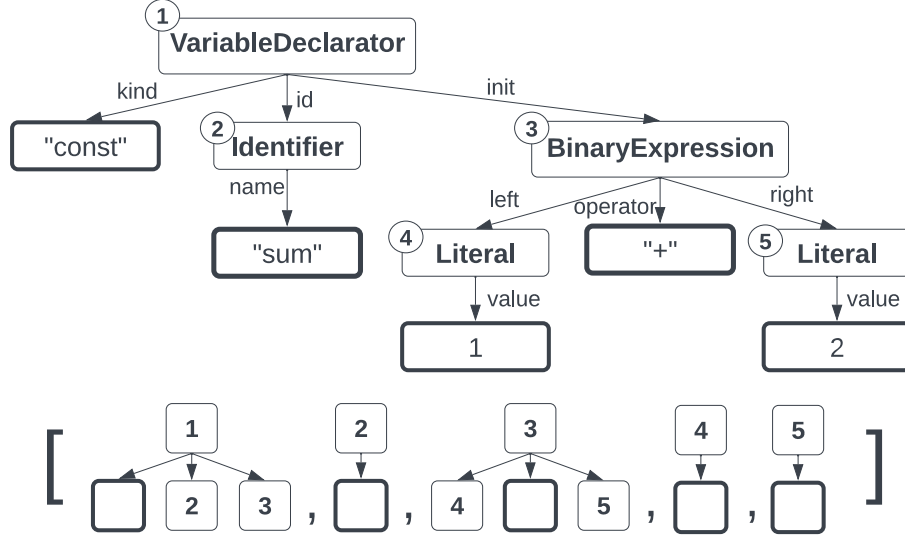


Figure 4.3: Converting AST from Fig. 2.1 to fragment sequence

**Definition 4.3.1** (Fragment). A fragment of an AST  $T = (N, E, n_0)$  is a subtree  $F_i = (N_i, E_i, n_i)$ , which satisfies the following properties:

- $n_i \in N$  s.t.  $C(n_i) \neq \emptyset$
- $N_i = n_i \cup C(n_i)$
- $E_i = \{(n_i, n') | n' = C(n_i)\}$

We carry out a pre-order tree traversal to generate a sequence of fragments from an AST. Each time we visit a node  $n_i$ , we extract a fragment with root  $n_i$ . We can see a visual representation of this process in Fig. 4.3 in which we have extracted the fragments from the AST shown at the top to produce the sequence below it. In addition, we denote the type of the fragment to be the type of its root node. For example, the first element of the fragment sequence in Fig. 4.3 would be of type "VariableDeclarator".

## Sequence Modelling

Montage uses a Long-Short Term Network (LSTM) architecture for sequence modelling. However, we have chosen to improve upon this by using a Transformer based model. As mentioned in Section 2.5, these models significantly improve upon LSTMs and have quickly become state-of-the-art.

We begin by forming a vocabulary of unique fragments. We convert all our input seed files into their corresponding AST fragment sequences. We then extract fragments with a frequency greater than 4 to form our vocabulary. We label any fragment with a frequency less than four as Out-of-Vocabulary, and we replace all occurrences with a placeholder fragment containing only the fragment type. For example, a **Literal** node with a random string "abcdefghijkl" may only appear in our seed files once and is not included in our vocabulary. However, we still want to identify this node as a **Literal** as it gives the model contextual information, which may impact the representation of nodes around it. We add all of these anonymous fragments to our vocabulary and then assign a unique integer id to each element in the vocabulary,

which is standard practice for preparing data for Transformer based models. As discussed previously, the BERT family of models makes use of extra tokens, such as '[MASK]', '[CLS]', '[PAD]' and '[EOS]'. We add these tokens to our vocabulary and assign them all ids. We augment our sequences with a '[CLS]' token at the start and an '[EOS]' token at the end.

BERT-style models have a limit on the input sequence length due to quadratically increasing computation and memory costs. Due to this, we had to trim any sequences we input to our model to a length of 512. We found that the fragment sequence length of 90% of our seed files was less than or equal to this limit. In addition to this, our agent will rarely be at the root node but instead at deeper nodes which further nullifies this limitation.

As described in Section 2.5, we pre-train the model using a slightly modified version of the Masked Language Modelling objective. Instead of using the training data as is, we propose that we augment each sample usage by taking a random sub-sequence from it and using this as our sample. After pre-training, we plan to use this model to take both **target** and **context** states from the environment and process them into their vector representations. From previous sections, we know that the **target** and **context** states could be any subtree of a program. Therefore, by taking a random sub-sequence from our input, we ensure that our training data is more representative of the data it will see later. We note that, by design, the '[EOS]' token is not always present at the end of the input sequence due to this random sub-sequencing. Our motivation behind this choice was that the model could learn to correlate a missing '[EOS]' token with a truncated sequence.

### 4.3.2 Double Deep Q-Network (Double DQN)

In order to train our Double DQN agent, we require Experience Replay, a fixed-size buffer of transition tuples, i.e.  $(s, a, s', r)$ . After every interaction with the environment, the agent stores these values in the Experience Replay buffer. Once we have collected several samples, we can start optimising our agent. We carry out optimisation after every interaction with the environment.

We start by sampling a random batch of transitions from the Experience Replay buffer. We then compute the vector representation of the **context** and **target** states and concatenate them to form a single vector representation for the whole state. We carry this process out for both  $s$  and  $s'$  and for every batch element. We can then use this single vector state representation as the input to our Double DQN. We use our processed transitions to calculate the Double DQN loss shown in Equation 2.12. Finally, we can use standard machine learning techniques to backpropagate our loss to optimise the policy network parameters and fine-tune the AST network parameters.

# Chapter 5

## Implementation

### 5.1 Language

We implemented our fuzzer using Python as our primary language. We specifically chose Python due to its rich pool of resources for both machine learning and reinforcement learning [36]. These were essential due to the use of Transformers and Deep Neural Networks, which are challenging to implement from scratch. In addition, Python has third-party libraries that allow for the parsing and generating JavaScript code to and from an AST, reducing the development cost.

### 5.2 Pre-Processing

Our fuzzer requires a normalised seed corpus and diverse subtrees to be effective. We start by taking input seed files of JavaScript programs from various sources, such as JavaScript engine test suites and vulnerability databases. For our AST representation, we make use of the ESTree standard [37], which supports several parsers and code generators such as Esprima [38] and Escodegen [39], which both have corresponding Python ports. These programs are parsed into their AST representation and executed to get their coverage data. For our engine, we decided to primarily focus on Google Chrome’s V8 engine due to its vast popularity and active development.

We normalise variable, function and class identifiers in the AST, as this hugely improves our AST vector representation since we are more likely to get a hit for these identifiers in our vocabulary. We then traverse the normalised ASTs to extract all subtrees and store these by root node type. Now that we have extracted a wide variety of subtrees from the full set of input seeds, we minimise these into a set that produces unique coverage.

### 5.3 JavaScript Engine Modification

For our fuzzer to work, it requires a coverage metric from the JavaScript engine. Producing this metric is not trivial, as most engines still need to build this feature. Most engines are built in C++ as they require a high level of performance.



Recording the coverage of a fuzzer requires inserting instrumentation before or during compilation. For example, AFL [40] uses its own augmented version of Clang [41], which instruments the code as it compiles it.

We used a technique and implementation developed by Fuzzilli [15], which uses Clang’s CoverageSanitizer [42]. This technique requires modifying the target JavaScript engine’s source code to add calls to specific functions defined by the sanitizer. We chose this approach as it had the lowest development cost while providing the same statistics as other methods.

The coverage sanitizer works by inserting a call to a user-defined function ‘\_\_sanitizer\_cov\_trace\_pc\_guard’ on every edge of the control flow graph with the specific index of the hit edge. By representing each edge as a bit in a bytearray, we can efficiently store and set edges which have been hit using the formula:

$$\text{edges}[\text{index}/8] \mid = 1 \ll (\text{index} \% 8)$$

In addition to this, the sanitizer also inserts a call to a function called ‘\_\_sanitizer\_cov\_trace\_pc\_guard\_init’ at the start of the program along with arguments which can be used to calculate the total number of edges in the program. This function is useful for initialising any data structures which may be needed to record coverage.

We use shared memory areas to store the coverage information so that our fuzzer can access it, which is written in Python. We set up the shared memory area in our Python code before calling the JavaScript engine. We store the id of the shared area in an environment variable which can be read at runtime by the engine. In the sanitizer initialisation function in the engine’s source code, we read the environment variable and access the shared memory area using this id. Finally, the initialisation function sets up the bytearray of appropriate size in the shared memory area and stores a variable indicating the total number of edges. After execution of the engine, we can access the shared memory area to read the desired coverage information.

## 5.4 Training

We designed our fuzzer to run in two stages; first, we pre-train our AST Transformer, then we run our Double DQN agent, which trains itself and fine-tunes the AST transformer simultaneously. However, once implemented, we found the second step of this to be impractical due to the optimisation time of a Transformer. Although the optimisation time was only 0.4 seconds, this is a considerable time for a fuzzer to waste not executing any inputs. Additionally, DQNs require hundreds of thousands of steps to show any sign of improvement in the policy. Due to this, we found it impossible to run our hyperparameter tuning and evaluation experiments with this model, as each run would take over 24 hours to complete.

Instead, we split training into three steps: pre-training the AST Transformer, fine-tuning the AST Transformer and training the Double DQN. The fine-tuning step involves training both our Double DQN agent and the Transformer simultaneously,



as discussed previously. However, instead of using this as our fuzzer, we save the AST Transformer weights resulting from the training.

Now for our final fuzzer, we initialise a new DQN agent with the fine-tuned weights for the AST Transformer and freeze them. Therefore, we no longer have to train the AST transformer while our main fuzzer is learning. This change reduces the optimisation time to 0.01 seconds, making the fuzzer more applicable to the real world.

# Chapter 6

## Evaluation

This chapter briefly overviews some of our experiments to influence our final architecture and hyperparameters. Following this, we aim to use the best model found as a result of this tuning to answer the following research questions:

**RQ1** Does the use of reinforcement learning show any improvement over a naive baseline strategy

**RQ2** How does the performance of a reinforcement learning-based fuzzer compare to other state-of-the-art fuzzers

**RQ3** Did the reinforcement learning fuzzer find any patterns about which parts were necessary for triggering interesting behaviour

### 6.1 Experimental Setup

We used two different machines for our evaluation experiments. For GPU-heavy task (Transformer training), we used a Ubuntu 22.04.2 LTS virtual machine equipped with 70 GB of RAM, a 16-core CPU (AMD EPYC Naples) and an NVIDIA A30. For our second machine, we used an Arch Linux machine equipped with 32 GB of RAM, a 6-core CPU (AMD Ryzen 5 3600 Processor) and an NVIDIA GeForce RTX 3070 Ti GPU for tasks not reliant on a high-specification GPU.

All experiments are carried out on version 8.5 of the V8 engine. We chose an older version of the engine to give all fuzzers the best chance to show their bug-finding capabilities. We also note that for all experiments, we use the same set of seed inputs with a base coverage of 15.35234%.

### 6.2 Development Experiments

#### 6.2.1 AST Transformer

We had multiple limitations on the architecture of the AST Transformer network. The first limitation was hardware-based, as we could only fit a specific network architecture and batch size on our GPU. The second limitation was time-based, as

Hyperparameter	
No. of layers	3
No. of attention heads	8
Hidden dimension	512
Intermediate Dimension	2048
Batch size	64

Table 6.1: Hyperparameters used for AST Transformer pre-training

		Dropout	
		0.0	0.1
<b>LR</b>	0.001	2.82777	2.83647
	0.0005	2.74609	2.79290
	0.0001	3.41621	3.40637

Table 6.2: Hyperparameter grid search for AST Transformer fine-tuning

we had to consider the initial pre-training time for the Transformer and the fine-tuning time together. Working under these limitations, in Table 6.1, we present a list of the final values chosen for each parameter.

With these limitations, we tuned two main parameters: the learning rate and dropout. We carried out a hyperparameter grid search for the model, which achieved the best l1-loss on a held-back test set at the end of the 50 epochs. We can see the results of this tuning in Table 6.2. The table shows that the best-performing configuration was a learning rate of 0.0005 and a dropout of 0. We re-ran our training for 100 epochs with the full dataset using these parameters to produce our final pre-trained model.

### 6.2.2 Transformer AST fine-tuning

As explained in Section 5.4, we use the final AST transformer model from pre-training and fine-tune it in the context of our problem. The same GPU limitations apply as we still train the AST transformer’s parameters. We were limited to using a batch size of 32 for our fine-tuning.

As done at the pre-training stage, we conducted a hyperparameter grid search to find the optimal values for this setting. Due to time constraints, we did not choose to modify DQN-specific hyperparameters such as gamma, replay memory size and steps between target network updates. We fixed these to reasonable values, which allowed for stable training. These values are shown in table 6.3.

Hyperparameter	
Replay memory size	10000
Gamma	0.99
Target update frequency	1000
Epsilon decay	0.99995
Batch size	32

Table 6.3: Hyperparameters used for AST Transformer fine-tuning

		Learning Rate	
		0.001	0.0001
<b>DDQN Architecture</b>	[512, 256]	15.45267	15.49162
	[2056, 1024, 512]	15.50024	15.45341
	[2056, 1024, 512, 256]	15.45957	15.48570

Table 6.4: Final coverage achieved for each hyperparameter pair during AST Transformer fine-tuning

		Learning Rate	
		0.001	0.0001
<b>DDQN Architecture</b>	[512, 256]	-2.38	-5.34
	[2056, 1024, 512]	-0.73	-1.13
	[2056, 1024, 512, 256]	-2.87	-3.03

Table 6.5: Average reward achieved for each hyperparameter pair during AST Transformer fine-tuning

We chose the following two hyperparameters in the search, which we believed to affect fine-tuning the most: the learning rate and the Double DQN architecture. At this stage, we tuned the Double DQN architecture, as it significantly impacts what fine-tuned AST Transformer might learn. This impact is due to the loss used to optimise both networks relying on the action values predicted by the Double DQN network. Therefore, if our Double DQN were bad at predicting these values, the performance of our AST Transformer would also suffer. To carry out our hyperparameter grid search, we ran our agent with each configuration for 50,000 steps and then recorded their final coverage and average reward. We present the results of this hyperparameter tuning in Tables 6.4 and 6.5. This table shows that the DDQN architecture of [2056, 1024, 512] and a learning rate of 0.001 produce both the highest coverage and reward. Therefore, we choose these parameters as our optimal ones for this stage of training.

In addition to hyperparameter tuning, we carried out an extra experiment to see how pre-training our AST transformer affects our agent’s performance. We did this by carrying out two runs; one with the final pre-trained AST Transformer and one with a newly initialised AST Transformer. From the results in Fig. 6.1, we see that in the same number of actions, the agent using the pre-trained AST Transformer performs considerably better than the newly initialised one. More quantitatively, using a pre-trained transformer resulted in 30% more coverage in the same number of steps. This improved performance backs up our choice to use a pre-training step before fine-tuning our downstream task.

### 6.2.3 Double DQN

We took the AST Transformer from the best-performing agent in fine-tuning to train a new agent. We freeze the parameter of the AST Transformer so they no longer change. We use the same Double DQN network architecture as found to be optimal for the previous step. As we no longer train our transformer network, we can now include batch size in our tuning. We once again tune the learning rate, but we also

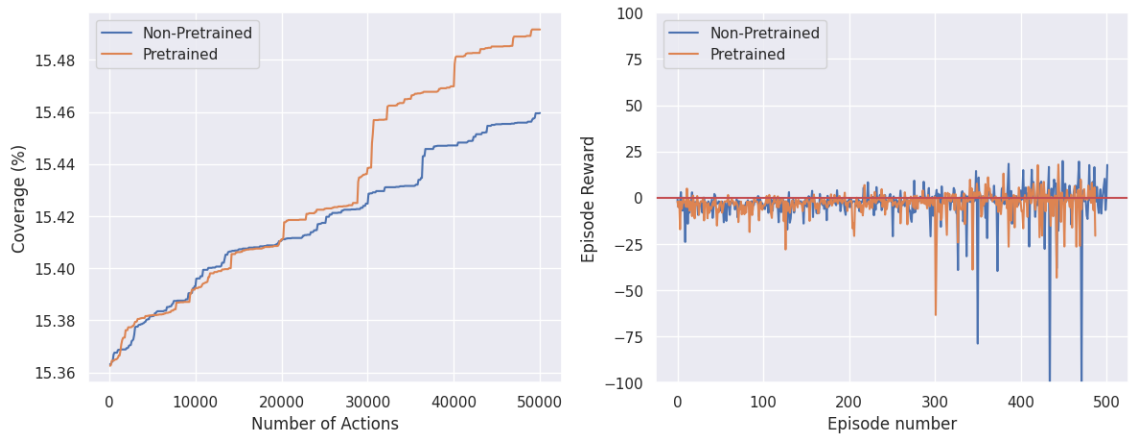


Figure 6.1: Pre-trained AST Transformer vs Un-trained AST Transformer: Coverage vs Number of Actions (left) and Rewards for every 5th episode (right)

tune more granular hyperparameters directly relevant to the Double DQN, such as epsilon decay, gamma value, steps between updates and replay memory size. We also have a parameter for clipping the gradient of the network updates. However, we found that any value greater than one leads to unstable training, so we kept this constant.

Due to a large number of hyperparameters, we could not do an exhaustive search in the space. Instead, we manually optimised values, improving the fuzzer’s coverage in a fixed number of steps. We mainly did this by changing one parameter at a time and looking at its effect on the coverage. For the interest of space, we give an example of the tests we carried out below and include the full set of optimal hyperparameters in Table 6.6.

We start by investigating the effect of replay memory size on the final coverage and stability of training. We chose two values for the replay memory of 75,000 and 100,000 as we found that values below this lead to catastrophic forgetting [43], a common issue in all neural network techniques. The results of this experiment can be found in Fig. 6.2. These results show that using a bigger replay memory (100,000) improves the final coverage achieved and produces more stable positive rewards. We also calculate the reward for the last 500 episodes, at which point the epsilon has decayed to a low value, which gives us a better idea of the raw performance of the agent. This calculation gives us an average reward of **6.24** for the big replay memory (100,000) and an average reward of **0.10** for the lower replay memory (75,000). This further justifies the selection of 100,000 as our replay memory size. We carried out tests that replay memory sizes greater than 100,000 but did not find it significantly increased the agent’s performance.

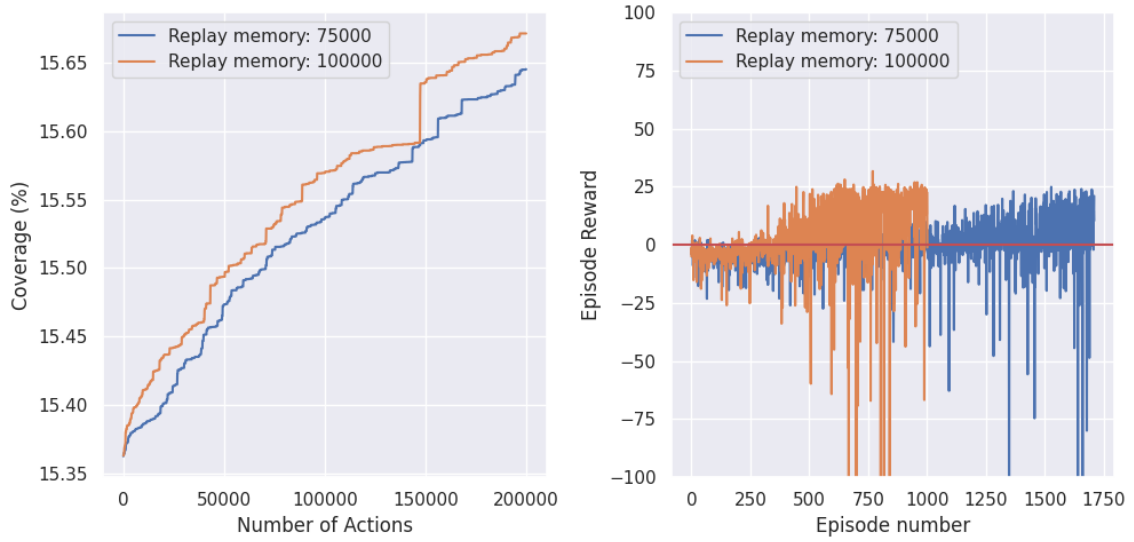


Figure 6.2: Big replay memory (100,000) vs Small replay memory (75,000): Coverage vs Number of Actions (left) and Rewards for every 5th episode (right)

Hyperparameters	
Learning Rate	0.0005
Replay memory size	100000
Epsilon Decay	0.99997
Target Update Rate	1000
Gamma	0.99

Table 6.6: Hyperparameters used for DDQN training

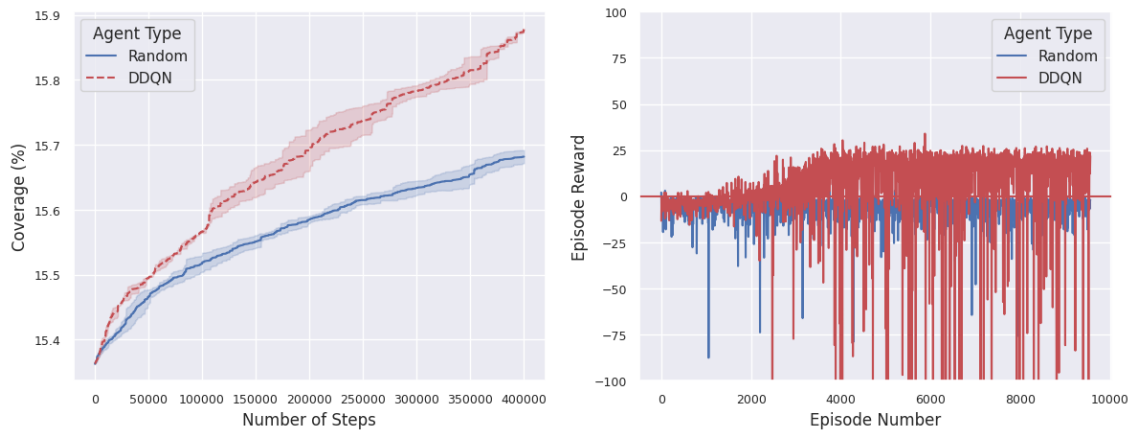


Figure 6.3: DDQN agent vs Random agent: Coverage vs Number of Steps (left) and Episode rewards for every 5th episode (right)

	Mean Final Coverage (%)	Std Deviation (%)	Avg. Execs
Random	15.68172	0.01238	73809
DDQN	15.87769	0.0015	289135

Table 6.7: Final coverage results of DDQN vs Random agent

## 6.3 Research Questions

### 6.3.1 Evaluating against naive baseline (RQ1)

We start by evaluating our fuzzer against a naive baseline. We use an agent that randomly selects an action at each time step for our naive baseline. This comparison allows us to see the effect, if any, the use of reinforcement learning is having on the fuzzer. For this comparison, we ran three runs of 400,000 steps for both our Double DQN agent and Random agent. We use the same seed corpus during all runs. We can see the results of this experiment in Fig. 6.3 and Table 6.3. We see that the DDQN agent both achieved better final coverage and a higher reward on average than the random baseline. This shows that our DDQN is learning some policy which helps mutate the test inputs more effectively.

However, as mentioned before, we want to ensure that we evaluate the real-world performance of our fuzzer, which involves comparing the amount of coverage gained per unit of time. We see a plot in Fig. 6.4 between our DDQN agent and the random agent. We see that the random agent increases its coverage at a much higher rate at the beginning of fuzzing. As the time running the random fuzzer increases further, we start to see a drop-off in coverage increase.

In order to explain this behaviour, we first have to look at the number of executions carried out by each agent shown in Table 6.8. Our random agent carries out much fewer executions than our DDQN agent. As we discuss later in our evaluation, the engine execution is costly, taking up almost 0.1 seconds per execution. Over the course of the fuzzing run, the DDQN agent spends five and a half hours more solely on executing the engine.

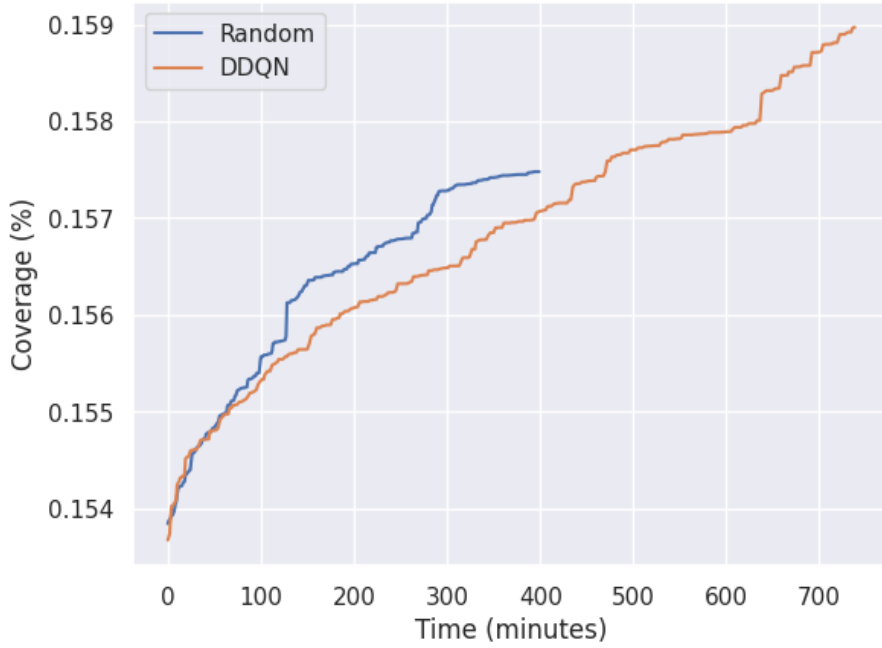


Figure 6.4: Coverage (%) vs time (min) for DDQN agent and Random agent

The random agent does not learn from its interaction, so it has a high action failure rate. Due to this, it executes the engine less than our DDQN agent and the random agent to carry out more actions in the same space of time. When the random agent gets lucky and a mutation action succeeds, if it is still near the run's beginning, it gets the "easy" coverage. However, after a certain point, it can no longer improve its coverage using this technique as all the "easy" to find paths have been explored. Due to this, if we continue executing the baseline agent, we expect it to finish lower than the DDQN at the end of the 12 hours shown on the graph.

Although we did not implement our fuzzer this way, we may in the future want to use a purely random strategy to gain the "easy" coverage quickly and then follow up with our DDQN fuzzer once it hits a roadblock.

### 6.3.2 Evaluating against state-of-the-art (RQ2)

We continue evaluating the real-world performance of the DDQN by comparing it to other state-of-the-art fuzzers. In this section, we aim to carry out a time-based performance analysis of our DDQN fuzzer against Fuzzilli and DIE, which, as we have discussed, are two of the most cutting-edge fuzzers for JavaScript engines. We gave every fuzzer the same input corpus of seed programs for this test and carried out two runs of each fuzzers for 12 hours on the same machine. We note that we used the default setup of DIE, which initialises eight process to run distributed fuzzing.

Our DDQN agent and DIE require a separate pre-processing step of the input corpus, which is included in this 12-hour window. Due to the differences in how each fuzzer measures and stores statistics, to ensure consistency, we used the final set of test cases produced by each fuzzer and re-ran them through our instrumented JavaScript engine to produce the final coverage results.



	Avg. Coverage (%)	Std. Deviation (%)	Avg. Execs	Crashes
<b>Fuzzilli</b>	16.081185	0.118625	530383	0
<b>DIE</b>	15.97604	0.02396	841810	0
<b>DDQN</b>	15.88373	0.00622	296769	0

Table 6.8: Final coverage results of DDQN vs State-of-the-Art fuzzers

	Coverage Increase (%)	Percentage difference
<b>Fuzzilli</b>	0.728845	+ 37.1%
<b>DIE</b>	0.6237	+ 17.3%
<b>DDQN</b>	0.53139	-

Table 6.9: Percentage difference between coverage gained by each fuzzer

Firstly we see that none of the fuzzers managed to generate an input which caused a crash in the JavaScript engine. We expected this result as most fuzzers need several days of running to find a crash [44]. This time is needed because although bugs are present in the version of the JavaScript engine we chose, they are usually quite subtle and require a convoluted, carefully crafted input to be triggered.

Secondly, we notice that our DDQN fuzzer achieved less coverage than both Fuzzilli [15] and DIE [14]. We note that Fuzzilli and DIE have been developed over multiple years, allowing them time to optimise their fuzzers to maximise throughput and effectiveness. In particular, Fuzzilli was first released in 2020 and has since become a production-level fuzzer, having been integrated as part of V8’s fuzz testing suite. Comparatively, we have undertaken this project in approximately eight months, which is important to consider, especially since evaluating fuzzers and making improvements can take weeks due to the long runtimes needed for evaluation.

To better understand the difference in the coverage values, we calculate the percentage difference of increase for both DIE and Fuzzilli with respect to our fuzzer. Table 6.9 shows the results of these calculations. Looking at these values, we see a more promising picture of the performance of the DDQN, with it only being 17% off DIE.

We also consider the number of engine executions carried out by each fuzzer. Our DDQN fuzzer carries out less than half the executions compared to DIE and Fuzzilli. As mentioned at the beginning of this section, DIE uses distributed fuzzing to increase its throughput significantly. Additionally, Fuzzilli implements a technique called Read Eval Print Loop (REPL) in which the JavaScript engine sub-process is started once and re-used multiple times. Comparatively, we create a new subprocess on every iteration, with significant overhead on startup and cleanup. Unfortunately, we did not have time to implement either of these techniques. However, considering that we did not use any such optimisation and still achieved a high coverage increase shows that our fuzzer can be useful in the real world.

To confirm that we were losing time due to our executions, we recorded the time taken for an action to be applied to the AST, the time for generating the code from the AST, the time to execute the engine (when needed) and the time to optimise the DDQN network and the time. We display the average time for each section in

	Average time (s)
Action	0.00322
Code Generation	0.00061
Execution	0.17058
Optimisation	0.00322

Table 6.10: Runtime breakdown of each component of our DDQN fuzzer

Table 6.10. We see that our execution is part of the fuzzer which takes the most time. Therefore, by improving our throughput we could be on-par if not be better, the state-of-the-art fuzzers.

### 6.3.3 Evaluating patterns found by DDQN agent (RQ3)

In this section, we explore the general behaviour of our fuzzer by looking into the actions it chooses and the node types it gravitates towards. Additionally, we look at some of the output test cases of our fuzzer to further understand what it has learned over the training period.

#### Action distribution

We begin by looking at the action distribution shown in Fig. 6.5. Here we see that our fuzzer heavily favours the replace action, having chosen it almost 300,000 times in the 400,000 steps we ran it for. Comparatively, we see that the second most action was move down, which was selected only 36,000 times. Although this seems low at first, after looking at the seed corpus, we see that the average depth of an AST in the corpus is only 9.45. On average, it takes a maximum of 9 actions for the agent to get to its desired node, not considering randomness. Therefore, even though the number of times the agent selects these movement actions is low, they are still effective in helping the agent move around.

Looking further down, we see that the end, modify and remove actions are invoked less than 8000 times. To check if these actions only occurred when our  $\epsilon$  was high, we can look at the action distribution for the last 100,000 steps, shown in Fig. 6.6. In this new distribution, all of our mutations except replace occur less than 1500 times. This indicates that the fuzzer has not found them useful for increasing coverage. We find this a surprising result as we expected at least some of these to help create test cases.

### 6.3.4 Node type distribution

We now look at the distribution of node types on which the fuzzer acts, which can be found in Fig. 6.7. For readability, we have removed any node with a frequency of less than 50. From this graph, we see that the most acted on is the ExpressionStatement node. This is a very interesting behaviour as this lines up with the technique used in DIE to preserve structural aspects. In their implementation, the primary mutation action is to select a random expression statement in the code and then mutate it. It seems that our fuzzer may have learned a similar policy to this. By checking the type of action occurring on this node, we see that over 27,000 of the 32,000 actions

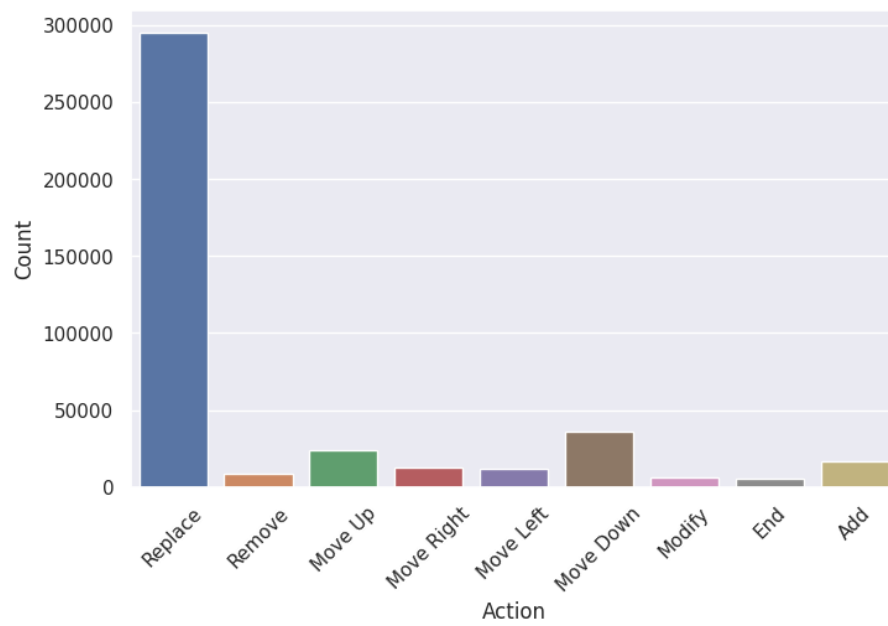


Figure 6.5: Distribution of actions over after 400,000 steps

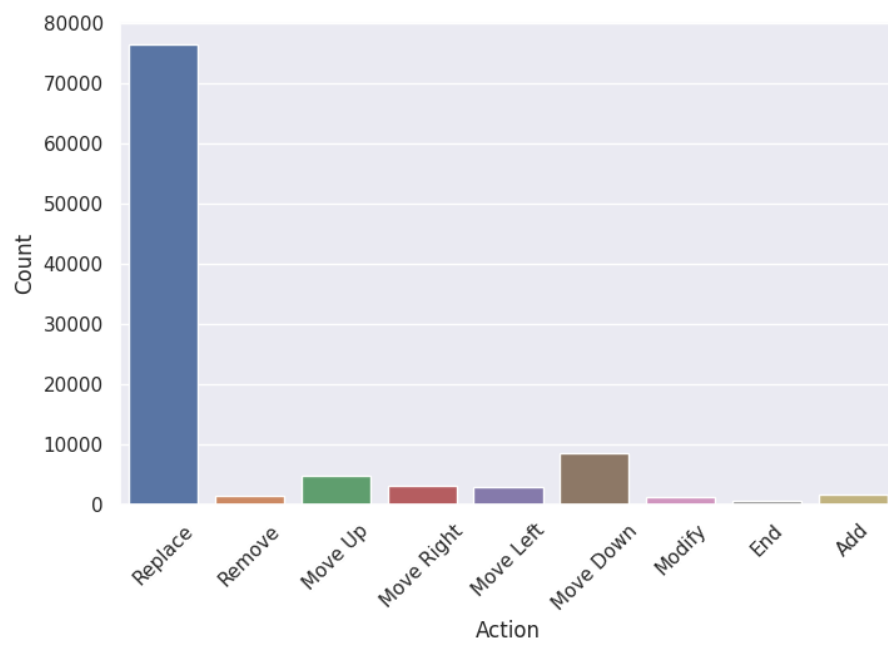


Figure 6.6: Distribution of actions for the last 100,000 steps of fuzzing

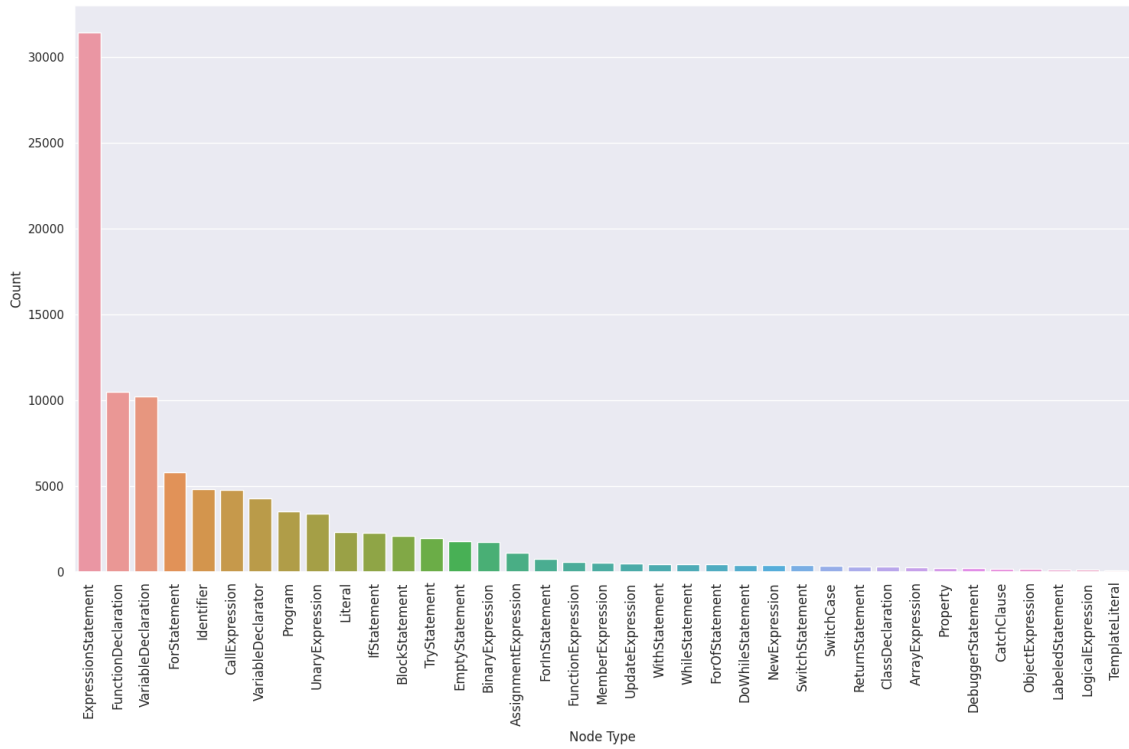


Figure 6.7: Distribution of node types visited in the last 100,000 steps of fuzzing

that acted on the node were replace actions. This indicates that our hypothesis is, in fact, correct, and our fuzzer has learned one of the main mutation strategies employed by DIE.

The next few most popular node types are more structural elements of the program, such as FunctionDeclarations and VariableDeclaration. Once again, looking at the actions carried out on each one of these node types, we find the same outcome as before; the primary action acting on the node is Replace. We do not consider this to be surprising as we know that Replace is by far the most popular action overall. However, the fact that our fuzzer is choosing to mutate these structural elements shows that it is still beneficial to modify them sometimes.

### 6.3.5 Example test cases

We look at how the above observations translate to an actual test cases. Here we look at a test case generate by our fuzzer which resulted in increased coverage:

```

1 +function f22(v5) {
2 +   v5 = v5 | 0;
3 +   return (v5 | 0) / -2147483648 | 0;
4 }
5 for (var v3 = 0; v3 < 10000; ++v3) {
6 -   var v4 = f0('The spirit is willing but the flesh ...') | 0;
7 +   var v4 = f22(-9223372036854775808) * f22(1000000000.0) | 0;
8   if (v4 != -1136304128) {
9   }

```

```
10 }
```

Listing 6.1: Example of test case with structural preservation

In this example, the fuzzer has added a function at the program's top. Additionally we see that the changes preserve the structure of the of the for loop and instead mutate the Expression assigned to the `v4` variable. This once again, reaffirms that our fuzzer has learned to preserve structural aspects of the code.

# Chapter 7

## Conclusion

Overall in this project, we implemented a novel reinforcement learning guided mutation-based fuzzer for the JavaScript engine. Our technique introduces the idea of using Abstract Syntax Tree (AST) level mutations with a reinforcement learning agent instead of source code level mutations which others have tried. In order to achieve this goal we had to come up with several novel idea such as our formulation of AST mutation-based fuzzing as a Markov Decision Process.

We also developed a novel way to compute a vector representation of an AST, by converting it to a sequence and using Transformer-based model to get our final representation.

In our evaluation, we investigated the performance of our reinforcement learning-based approach to both a random mutation strategy and to other cutting-edge JavaScript engine fuzzers. Our results found that our fuzzer produces 70% more coverage than a random baseline approach. We also found that our fuzzer is only gains 17% coverage that DIE, which is a state-of-the-art JavaScript engine fuzzer.

Overall, our implementation, results and evaluation show promising signs for using reinforcement learning in fuzzing the JavaScript engines and expanding it to be used in compilers.

### 7.1 Future Work

We have several ideas for both improvements and also optimisations which could be made to push our fuzzer to state-of-the-art levels:

#### 7.1.1 Optimising engine execution time

As shown in our evaluation, our fuzzer needs to catch up regarding the number of executions over time. As previously mentioned, this can be largely improved by using the Read Eval Print Loop (REPL) employed by Fuzzilli. Using this technique, we could start the engine subprocess once and use it multiple times, removing the overhead or continually starting and ending processes. We have seen that this is the main bottleneck in our implementation. Therefore, by alleviating it, we believe that we would be able to reach the coverage levels of Fuzzilli and DIE.

### 7.1.2 Evaluating bug finding capabilities

Unfortunately, we were not able to evaluate the bug-finding capabilities of our fuzzer. From previous research [14, 44], we can see that it usually takes 3 days to find a bug in the JavaScript engine, however, due to the tight timeline of the project, which did not allow for us to carry out multi-day runs of our fuzzer.

### 7.1.3 More fine-grained Replace action

As seen in our evaluation the Replace action was the dominant choice of our fuzzer. As a result of this we think that we can make the Replace action even more fine-grained. Instead, of replacing our target state with a random node with the same sub-type we can instead use actions which specify the type of node we want to replace it with. This greatly reduces the randomness in the fuzzing process and may allow the agent to pick up even better mutation strategies.

## 7.2 Ethical Considerations

Although this project aims to find bugs in the JavaScript engine to help improve its robustness and reliability, malicious entities are also looking to exploit such bugs for their gain. Vulnerabilities are critical in the case of JavaScript engines, which run in browsers that users may use to access sensitive information such as bank accounts. Therefore we emphasise that any bugs found by any software produced during this project should not be exploited for malicious purposes. In addition, we strongly advise that if any un-patched bugs are found in the current versions of any JavaScript engines, bug reports should be raised through the appropriate channel.

In our experiments, we use various neural network models, some of which require high-performance GPUs. We understand that carrying out such tasks can emit a large level of carbon dioxide, which is harmful to the environment. To this extent, we only used the GPUs for as long as necessary and did not spend copious amounts of time running large hyperparameter optimisations.

# Appendix A

## Double DQN Component Training Times

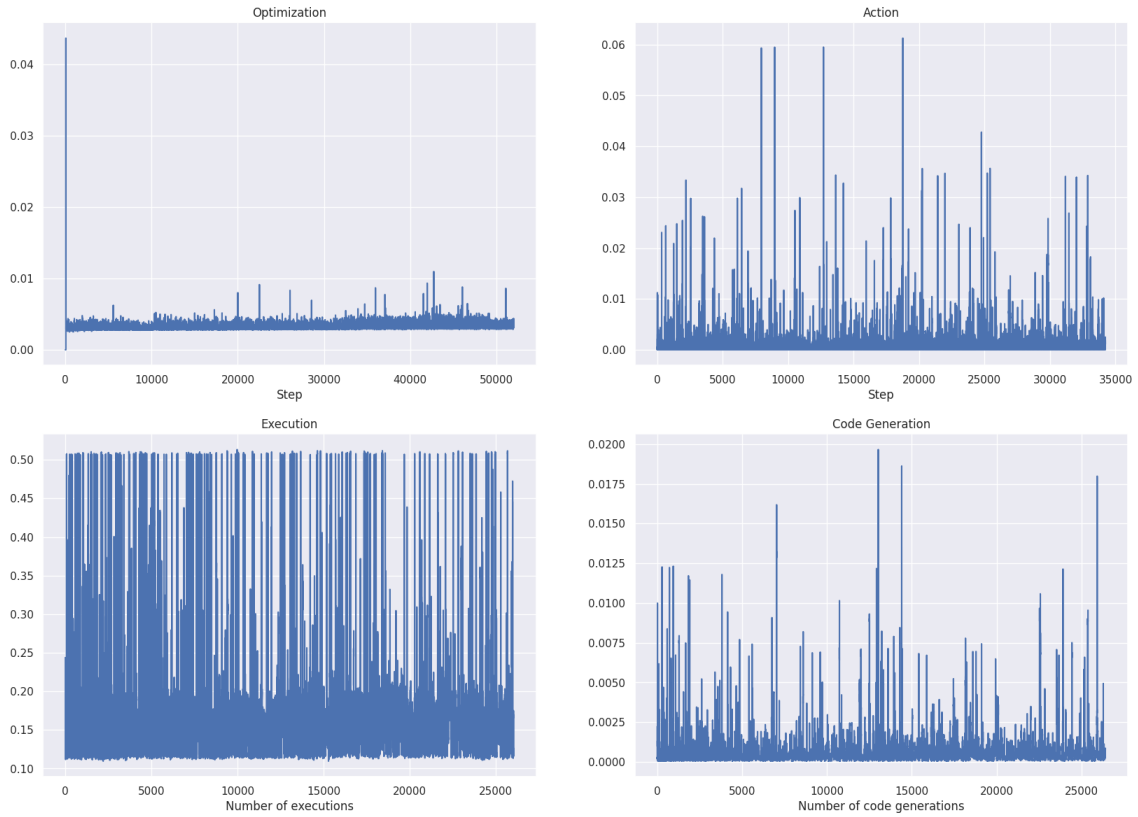


Figure A.1: Runtime of each component of our Double DQN fuzzer



# Bibliography

- [1] Corporation M. JavaScript basics; 2019. Available from: [https://developer.mozilla.org/en-US/docs/Learn/Getting\\_started\\_with\\_the\\_web/JavaScript\\_basics](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics).
- [2] Usage statistics of JavaScript as client-side programming language on web-sites;. Available from: <https://w3techs.com/technologies/details/cp-javascript>.
- [3] V8; 2023. Available from: <https://v8.dev/docs>.
- [4] SpiderMonkey; 2023. Available from: <https://spidermonkey.dev/>.
- [5] WebKit; 2015. Available from: <https://webkit.org/project/>.
- [6] Wikipedia. Netscape Navigator — Wikipedia, The Free Encyclopedia; 2023. [Online; accessed 18-June-2023]. <http://en.wikipedia.org/w/index.php?title=Netscape%20Navigator&oldid=1156285961>.
- [7] McIlroy R. Firing up the Ignition interpreter; 2016. Available from: <https://v8.dev/blog/ignition-interpreter>.
- [8] Meurer B, Bynens M. JavaScript engine fundamentals: Shapes and Inline Caches; 2018. Available from: <https://mathiasbynens.be/>.
- [9] National Vulnerability Database; 2023. National Vulnerability Database. Available from: <https://nvd.nist.gov/vuln/search>.
- [10] Kang Z. A Review on JavaScript Engine Vulnerability Mining. Journal of Physics: Conference Series. 2021 feb;1744(4):042197. Available from: <https://dx.doi.org/10.1088/1742-6596/1744/4/042197>.
- [11] ITU. Web usage statistics; 2022. Available from: <https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx>.
- [12] Yang X, Chen Y, Eide E, Regehr J. Finding and Understanding Bugs in C Compilers. SIGPLAN Not. 2011 jun;46(6):283294. Available from: <https://doi.org/10.1145/1993316.1993532>.
- [13] Wang J, Chen B, Wei L, Liu Y. Superion: Grammar-aware greybox fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE; 2019. p. 724-35.

- [14] Park S, Xu W, Yun I, Jang D, Kim T. Fuzzing javascript engines with aspect-preserving mutation. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE; 2020. p. 1629-42.
- [15] Groß S. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines; 2018. .
- [16] Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, et al. Playing atari with deep reinforcement learning. arXiv preprint arXiv:13125602. 2013.
- [17] Hessel M, Modayil J, Van Hasselt H, Schaul T, Ostrovski G, Dabney W, et al. Rainbow: Combining improvements in deep reinforcement learning. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32; 2018. .
- [18] Aschermann C, Frassetto T, Holz T, Jauernig P, Sadeghi AR, Teuchert D. NAUTILUS: Fishing for Deep Bugs with Grammars. In: NDSS; 2019. .
- [19] Garsiel T, Irish P. How browsers work; 2011. Available from: <https://web.dev/howbrowserswork/>.
- [20] Multi-process architecture. Chromium; 2023. Available from: <https://www.chromium.org/developers/design-documents/multi-process-architecture/>.
- [21] Security/Sandbox/Process model - MozillaWiki. Mozilla; 2019. Available from: [https://wiki.mozilla.org/Security/Sandbox/Process\\_model](https://wiki.mozilla.org/Security/Sandbox/Process_model).
- [22] Clark L. A crash course in just-in-time (JIT) compilers mozilla hacks - the web developer blog; 2017. Available from: <https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/>.
- [23] ; 2022. Available from: <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>.
- [24] Glazunov S. In-the-Wild Series: Chrome exploits; 2021. Available from: <https://googleprojectzero.blogspot.com/2021/01/in-wild-series-chrome-exploits.html>.
- [25] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery. Pearson Education; 2007.
- [26] Lee S, Han H, Cha SK, Son S. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer; 2020.
- [27] Chen J, Patra J, Pradel M, Xiong Y, Zhang H, Hao D, et al. A survey of compiler testing. ACM Computing Surveys (CSUR). 2020;53(1):1-36.
- [28] Sutton RS. Reinforcement Learning, Second Edition : An Introduction : An Introduction. The Mit Press; 2018.
- [29] Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning. In: Proceedings of the AAAI conference on artificial intelligence. vol. 30; 2016. .

- [30] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is all you need. *Advances in neural information processing systems*. 2017;30.
- [31] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:14090473*. 2014.
- [32] Devlin J, Chang MW, Lee K, Toutanova K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:181004805*. 2018.
- [33] Li X, Liu X, Chen L, Prajapati R, Wu D. FuzzBoost: Reinforcement Compiler Fuzzing. In: *Information and Communications Security: 24th International Conference, ICICS 2022, Canterbury, UK, September 58, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag; 2022. p. 359375. Available from: [https://doi.org/10.1007/978-3-031-15777-6\\_20](https://doi.org/10.1007/978-3-031-15777-6_20).
- [34] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. *arXiv preprint arXiv:170706347*. 2017.
- [35] Kozlica R, Wegenkittl S, Hirländer S. Deep Q-Learning versus Proximal Policy Optimization: Performance Comparison in a Material Sorting Task. *arXiv preprint arXiv:230601451*. 2023.
- [36] Nabeel S. Research on Machine Learning in Python: Main Developments and Technology Trends in DS, ML, and AL.
- [37] ESTree. ESTree. GitHub; 2023. <https://github.com/estree/estree>.
- [38] jQuery. Esprima. GitHub; 2021. <https://github.com/jquery/esprima>.
- [39] Tooling E. Esgodegen. GitHub; 2020. <https://github.com/estools/escodegen>.
- [40] Google. AFL. GitHub; 2021. <https://github.com/google/AFL>.
- [41] Clang C Language Family Frontend for LLVM;. Available from: <https://clang.llvm.org/>.
- [42] SanitizerCoverage - Clang 17.0.0git documentation;. Available from: <https://clang.llvm.org/docs/SanitizerCoverage.html>.
- [43] Wikipedia. Catastrophic interference — Wikipedia, The Free Encyclopedia; 2023. [Online; accessed 21-June-2023]. <http://en.wikipedia.org/w/index.php?title=Catastrophic%20interference&oldid=1150297000>.
- [44] Wang J, Zhang Z, Liu S, Du X, Chen J. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler.