Institute of Software Engineering

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Applying Machine Learning Techniques for Improving Test Input Generation in Embedded Fuzzers

Jannick Stuby

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Halimeh Agh Halit Eris
Commenced:	June 19, 2023
Completed:	January 30, 2024

Abstract

In the rapidly evolving landscape of the Internet of Things (IoT), the number of connected devices is growing exponentially. This growth is accompanied by an increasing demand to ensure the security and continuity of such devices, as they are often used in critical applications. Similar to other areas of software engineering, automated testing of embedded devices, namely fuzzing, is a promising approach to ensure the quality of such devices, while reducing the time and effort required for testing. However, the unique characteristics of embedded devices, such as limited resources and the lack of a standardized operating system, make it difficult to apply existing testing approaches to them. This thesis aims to understand the current state of the art of the fuzzing of embedded devices and researches possibilities to augment this process with the help of machine learning, to improve the input generation stage during fuzz testing.

To this end, we first conduct a literature review to identify the current state of the art of fuzzing embedded devices, highlighting differences to traditional desktop fuzzing. We then conduct an in depth review to evaluate existing machine learning approaches to traditional, as well as embedded fuzzing. Finally, using the information gathered, we propose a machine learning-based approach to augment the input generation stage during the fuzzing process of embedded devices. Our results show that machine learning can be introduced to fuzzing of embedded devices, but it requires significant manual effort to do so. Furthermore, we show that the current state of the art of publicly available methods is not yet mature enough to be used in general practice.

Contents

1. Introduction

In the rapidly evolving landscape of software engineering, the robustness and security of embedded systems have become a growing concern. As these systems increasingly underpin critical infrastructure and everyday technologies, their vulnerability to security breaches poses significant risks for critical infrastructure, industry processes as well as the security and privacy of consumers. This bachelor thesis, titled "Applying Machine Learning Techniques for Improving Test Input Generation in Embedded Fuzzers", aims to explore and improve the capabilities of fuzz testing in embedded systems through the innovative application of ML techniques. This introduction is segmented into Background and Motivation, Research Objectives and Research Questions, as well as Scope and Methodology, offering a comprehensive overview of the thesis's focus and direction.

1.1. Background and Motivation

In an era where technology is deeply integrated into the fabric of everyday life, ensuring the robustness and security of software and systems has emerged as an increasingly important requirement in the field of software engineering.

"Smart" devices are more and more adopted in a plethora of use cases, spanning critical infrastructure, such as industrial control systems in power plants, water treatment facilities, and transportation systems, including automotive applications, as well as in consumer electronics, such as smart home devices, wearables, surveillance systems, and medical devices. The growing reliance on these devices and systems has made them a prime target for malicious actors, who seek to exploit vulnerabilities in software and systems to gain unauthorized access to sensitive data, disrupt critical infrastructure, or cause physical harm. Especially in the case of consumer electronics, the security and privacy of users are at risk, as these devices often collect and process sensitive data, such as health data, location data, or biometric data, while device security is not or just weakly enforced [1].

Those devices are called embedded systems, integral to a myriad of applications, thus demanding rigorous testing to ensure their security and reliability. To achieve this rigorous testing, often fuzzing can be used. Fuzz testing, or fuzzing, is a well-established method in traditional software testing used to identify software bugs and vulnerabilities. It involves providing malformed or random data as inputs to a system to trigger failures and uncover security issues. However, in the realm of embedded systems, traditional fuzzing approaches face unique challenges. These

include the complexity of embedded system architectures, limited resources, and the intricacy of interactions between hardware and software components, making it necessary to guide and adapt the testing process manually, thus reducing its efficiency and effectiveness.

The integration of machine learning (ML) techniques in this context presents a promising possibility for improvement. By leveraging ML algorithms, it is possible to intelligently generate test inputs that are more likely to uncover hidden vulnerabilities in embedded systems, using it's ability to learn from data, adapt to new patterns, and predict outcomes.

In the recent years, a range of exploited vulnerabilities targeting embedded systems have been disclosed, showing the advancing threat model and highlighting the need for more robust testing processes.

For example, in 2020, a set of 19 security issues, classified as zero-day vulnerabilities [1] in a popular TCP/IP library by *Treck* were discovered and disclosed under the name *Ripple20*. Because this library is used in a wide range of embedded systems, including medical devices, consumer electronics, and industrial control systems, a multitude of devices became vulnerable [2]. Another example are the *Urgent/11* vulnerabilities, a set of 11 security issues regarding the *VxWorks* Real-Time Operating System, which is used in a multitude of devices, including medical devices, industrial control systems, and consumer electronics [3]. Those issues were also classified as zero-day. Even the widely known vulnerabilities Meltdown [4] and Spectre [5] were applicable to embedded systems, as they target used microprocessors.

These vulnerabilities were found to be exploitable by attackers, allowing them to take control of the device and execute arbitrary code. These examples highlight the need for more advanced testing processes for embedded systems, which can be achieved by integrating ML techniques into the fuzzing process.

To improve and sustain device security and reliability, in the past decade a multitude of standards and guidelines have been proposed to enhance the process of software engineering and hardware development. These standards and guidelines are often used as a basis for certification processes, which are required for the deployment of embedded systems in critical infrastructure, such as the automotive industry. Such standards include IEC 61508 [6], and IEC 62443 [7], which are security standards for industrial control systems and their networks, as well as many ISO standards.

The incorporation of ML in fuzz testing aligns with the principles laid out in multiple ISO standards on software security and testing. Those standards advocate for systematic, efficient, and replicable testing processes, emphasizing the need for innovative approaches to enhance software testing methodologies[2], which can be achieved by integrating ML techniques into the fuzzing process. *ISO 26262 - Road vehicles - functional safety* [12] and *ISO 21434 - Road vehicles - cybersecurity engineering* [13], in particular, are safety standards for automotive systems, which highlight the need

[1] zero-day meaning defining a severe vulnerability that must be fixed immediately

[2] cf. ISO 27001:2013-A12.6.1 [8], ISO 12207 [9], ISO 22301 [10], ISO 29119-1:2022 [11], IEC 61508 [6], IEC 62443 [7]

for a comprehensive testing process that explicitly includes the use of fuzz testing to identify vulnerabilities in embedded systems.

1.2. Research Objectives and Research Questions

The primary objective of this research is to explore the application of machine learning techniques to improve the test input generation phase of embedded fuzzers. This exploration seeks to contribute to the field of software testing by focusing on:

- Understanding the current state of fuzz testing methodologies for embedded systems
- Exploring various machine learning models suitable for generating test inputs
- Proposing a framework that integrates a chosen ML model into the fuzz testing process

Therefore the research will be guided by the following research questions:

RQ 1 What process is usually used in embedded fuzzers for input generation?

This question aims to understand the current state of fuzz testing in embedded systems, focusing on the input generation phase. It will be answered by conducting a comprehensive literature review of existing embedded fuzzers and their input generation processes, as well as traditional fuzzers to understand the context of fuzzing in general.

RQ 2 What ML techniques have already been used for improving input generation in traditional/desktop fuzzers?

By answering this question, it will be possible to identify and analyze existing ML techniques that have been applied to traditional fuzzing in general, to highlight and understand their potential for application in embedded fuzzers. This will be achieved by conducting a thorough literature review of existing research on ML in fuzzing.

RQ 3 How will the ML approach understand that it is on the right path to finding vulnerabilities in embedded systems?

By analyzing the results of the previous questions, it will be possible to identify and select a suitable fuzzer as well as a fitting ML model for the input generation phase of the fuzzing process. This question aims to understand how the ML model can be trained to generate test inputs that increase the likelihood to uncover vulnerabilities in embedded systems or increase code coverage by including gainable runtime information during the fuzzing loop. It will be answered by combining the findings during the answering of the previous ques-

tions, as well as analyzing the implementations of proposed approaches, if they are available.

RQ 4 How can the input generation stage be improved in Embedded Fuzzers (EFs)?

This question will be answered by proposing an approach, dependent on the findings of the previous questions, to improve the input generation stage of an existing fuzzer by integrating a suitable machine learning model, by taking different ideas and applied techniques found in the literature into account. Additionally challenges will be highlighted, that have to be solved during a possible implementation, as well as some guidelines for the implementation.

These questions are designed to guide the research towards an extensive understanding of the current state of fuzz testing in embedded systems and the potential for ML to enhance these processes.

1.3. Scope and Methodology

The scope of this research is confined to applying machine learning techniques specifically for improving the input generation phase during fuzz testing of embedded systems. It does not highlight other aspects of embedded system testing or the broader application of machine learning in software testing.

The main goal will be the proposal of a fuzzing prototype, which integrates machine learning techniques into the input generation phase of the fuzzing process by augmenting an existing fuzzer. The prototype will be evaluated against the non-augmented fuzzer, which uses traditional fuzzing methods, to assess its effectiveness in identifying vulnerabilities in embedded systems. The prototype will be based on an emulation based fuzzer for embedded systems, as it allows for a more controlled and reproducible testing environment, which is essential for the evaluation. Furthermore, emulation based fuzzing gives the opportunity to select a wider range of targets to fuzz on a bigger scale, as the overhead of configuring accesses to hardware devices and the necessity of obtainable hardware are both eliminated.

The methodology involves a combination of theoretical and practical approaches. Initially, a comprehensive literature review will be conducted to understand current practices and the potential for machine learning in this domain, comparing and evaluating the findings in the context of traditional fuzzing. As some research work in the domain of traditional and embedded fuzzing has only been published on ArXive [3], therefore has not been peer-reviewed, this literature review will not only include white literature, but also grey literature. Following this, various machine learning models will be identified, analyzed, and one of those will be selected for further testing, based on the applicability and potential effectiveness. The practical phase involves developing a prototype implementation that integrates this model

[3] https://arxiv.org/

into the fuzzing process choosing an existing fuzzing tool, followed by an empirical evaluation to assess its effectiveness in real-world scenarios, especially comparing the augmented fuzzer to the baseline.

In conclusion, this research aims to contribute to the field of software security by combining traditional fuzzing methods used for embedded devices with modern machine learning techniques, enhancing the security and reliability of embedded systems.

1.4. Structure of the Thesis

This document is structured as follows. After the current Chapter 1, which introduced the problem domain, highlighted the motivation and explained the research approach containing the Research Questions and methodologies, the following Chapter 2 provides an overview of the background and related work in the field of software testing, distinguishing between traditional fuzzing, embedded fuzzing, and machine learning. Chapter 3 presents our proposed approach to improve the fuzzing of embedded devices, highlighting possible challenges and implementation guidelines. Finally, Chapter 4 concludes the thesis and provides an outlook on future work.

2. Related work and Fundamentals

This chapter, will discuss the fundamentals and related work required to understand the topic of this document. we will start by creating a general overview of fuzzing with regards to all necessary research and projects relevant for state-of-the-art traditional fuzzers. Subsequently we will introduce machine learning (ML) concepts, that have been used in promising research regarding enhancement of the input generation phase during fuzzing, followed by an overview of embedded fuzzing and its fundamentals. This aims to guide the reader to the following section, which will examine already used approaches and applications of machine learning to the input generation phase of both desktop and embedded fuzzers. This chapter will also answer RQ 1, RQ 2 and RQ 3.

2.1. Traditional Fuzzing

This section will introduce the concept of fuzzing, its taxonomy, and its application in the field of software testing, initially focusing on general taxonomy and history beginning later highlighting some milestone fuzzing projects.

2.1.1. Fuzzing Definition

Fuzzing in general is an approach to automated software testing, aiming to find bugs and vulnerabilities by providing it with random or semi-random input, which is then processed by the software. The goal is to find bugs and vulnerabilities, which are triggered by malformed or unexpected input, which is often not handled correctly by the software, leading to crashes or other anomalies. The term “fuzzing” was first introduced by Miller et al. [14], who used the term to describe a technique, that was used to test Unix programs by providing them with random input. The term “fuzzing” is derived from the term “fuzz”, which is used to describe random data, that is utilized to fill buffers or other data structures, which are then used by the program. “Fuzzing” is often used synonymously with the term “fuzz testing”, which is a more general term, describing the process of testing software by providing it with random input, which is then processed by the software.

Following the definition of [15], Eisele et al. [16] describe the process more formally by denoting it as a tuple $(\mathcal{D}, \mathcal{F}, \mathcal{P})$, where $\mathcal{D}$ is the space of all inputs a target system $\mathcal{S}$ can take, $\mathcal{F}$ represents all saved inputs, that are selected during the process and $\mathcal{P}$

defines a probability function, that is used to select an input t_i to be added to $\mathcal{F}$ with probability p_i. All selected inputs can than be denoted as

$$\mathcal{F} = \{t_i \mid t_i \in D\}_{i=1}^{N} \tag{2.1}$$

Additionally, some kind of general observation mechanism is introduced by [16], to save how the system behaves under the input t_i, formally described as

$$\mathcal{O}_{t_i} \overset{\text{observe}}{\longleftarrow} \mathcal{S}_{\mathcal{C}}(t_i), 1 \leq i \leq N \tag{2.2}$$

Using this information, a very basic algorithm for fuzzing can be described as seen in Algorithm 1, which is directly derived from [15]. This algorithm does not take any additional mechanics to improve the fuzzing process into account, namely any kind of alteration on inputs starting from the seed corpus $\mathcal{C}$ and continuing with the generated inputs $\mathcal{F}$, it rather generalizes the adaption mechanism with new feedback in the $adjust()$ function in line 8, where the probability distribution is updated according to new findings, leaving any specifics out of the context. Such mechanics will be highlighted in the following sections.

Algorithm 1: Basic general fuzzing process, directly derived from [15]

Input: Target system $\mathcal{S}$ with configuration $\mathcal{C}$, initial seed corpus $\mathbb{C}$, probability function $\mathcal{P}$
Output: Inputs leading to unexpected behaviour $\mathcal{T}_x$

```
1 T_x = ∅
2 while not terminated AND not aborted do
3 |   t_i ← t_i ∈ D picked with probability p_i
4 |   O_{t_i} ⟵observe S_C(t_i)
5 |   if ¬expected(O_{t_i}) then
6 |   |   T_x ← T_x ∪ {t_i}
7 |   end
8 |   P ← adjust(P, O_{t_i})
9 end
```

Fuzzing is a very active field of research, with many different approaches and projects being published every year, which will be discussed in the following sections.

2.1.2. Taxonomy of Fuzzing

Historically, fuzzers were rather simple applications, aiming to provide programs under test (PUT) with randomized input, consisting of some kind of test case generator, which created random input strings or randomly mutated existing valid input, followed by a delivery module, which was needed to feed created strings to a specific program, taking all necessary steps to provide the target with acceptable input, and a monitoring module, which was responsible for detecting crashes and other anomalies. This approach is called *black-box fuzzing*, as the fuzzer has no knowledge

of the PUT's internal structure, and is only able to observe its behavior from the outside. It is still used today, but has been extended by a more sophisticated approach, called *white-box fuzzing*, which is able to use information about the PUT's internal structure, such as the source code, as well as static or dynamic analysis techniques, with the goal to create "less random", meaning more sophisticated, input strings, which are more likely to trigger bugs or unwanted behavior. In between those concepts lies the so called *grey-box-fuzzing*, which is able to use information to guide the fuzzer's input generation process. The separation between those approaches is not always a straight line. Many fuzzers may use some runtime information, while still being classified as black-box, while some white-box fuzzers may only use approximations or only analyze some parts of the PUT. Following, we will introduce the different approaches to fuzzing, as well as the more often used classification approach depending on the input adaption technique.

Black-Box Fuzzing

Black-Box Fuzzing is the most basic approach to fuzzing, directly following Miller et al. [14], who first introduced the term "fuzzing", and is still used today. Similar to black-box-testing in general software testing, it does not include knowledge of the programs internals to create test cases. It is based on the idea of creating random input strings, or randomly mutating existing valid seed inputs, and feeding the randomly generated or mutated strings to the PUT, without any knowledge of the PUT's internal structure. The only information, that is used by the fuzzer, is the PUT's observable behavior, such as crashes or other anomalies, which are used to determine the quality of the generated input strings and to select the next string to be mutated.

White-Box Fuzzing

White-Box Fuzzing [17][18] is a more sophisticated approach, which uses information about the PUT's internal structure, such as the source code, as well as static or dynamic analysis techniques, with the goal to create "less random", meaning more sophisticated input strings, that are more likely to trigger bugs or unwanted behavior. Such analysis could be Dynamic Symbolic Execution [19], where path constraints are created before the fuzzing process and then solved during fuzzing, which speeds up the fuzzing process and allows for better code coverage, but at the same time increases the complexity of the fuzzer, as well as the time needed to create the path constraints. Another method is the usage of taint analysis [20], which is used to track the dataflow through the PUT, and to identify data which is directly used by the PUT, which is then used to guide the fuzzing process.

Grey-Box Fuzzing

Grey-Box Fuzzing is a hybrid approach to fuzzing, that only uses partial information about the PUT's internal structure, trying to find a compromise between efficiency and effectivity, for example by using code coverage or taint-flow information to guide the fuzzer towards unexplored code paths [20][21][22][23], with the goal to either improve general code coverage, to explore as much execution paths as possible, or even to guide the fuzzer to paths which were deemed possibly vulnerable by performing some kind of analysis beforehand [24]. Grey-Box-Fuzzing tries to balance between minimal fuzzing overhead and obtaining as much information during runtime as necessary to create a powerful approach to fuzzing. To achieve such an approach on embedded devices, it is either necessary to use some kind of emulation, which allows for dynamic instrumentation and analysis of the PUT's runtime behavior, which will be discussed later, or introduce sophisticated monitoring and incorporation of new runtime information to the fuzzing process.

Mutation-Based Fuzzing

As randomly created input might present problems, because the PUT will most likely only accept correctly formatted input, which leads to the string being rejected during format checks, thus never reaching deeper code parts and leading to a very inefficient fuzzing process, mutation-based fuzzing often starts with a seed corpus of valid inputs. Those inputs might have been gathered during monitoring of the PUT's normal behavior, or might have been created by hand according to a known input specification, and often contain different valid input variations. During the fuzzing process, this seed corpus is then used to start the fuzzing loop, after which the strings are then randomly mutated to increase code coverage. During these mutations, the fuzzer might apply different mutation strategies, such as bit flips, byte flips, or mutations according to more sophisticated heuristics like dictionary-based mutations, which uses predefined values to mutate. Many of those standard mutation strategies are implemented in the fuzzing framework libFuzzer [25], which is part of the LLVM compiler infrastructure[1].

Generation-Based Fuzzing

As mutation-based fuzzers might not be able to create valid inputs, which are accepted by the PUT, generation-based or model-based fuzzers try to overcome that problem, by creating inputs adhering to a precise grammar describing valid inputs aiming to increase code coverage or only modifying parts of input strings, to comply with a specific input model, that is created beforehand. Therefore all created input string are valid in the sense of syntax, thus being accepted by the PUT. Typically, those grammars have to be crafted by hand, which is a very labour-intensive manual task. This approach is especially useful for PUTs, which are based on a

[1] https://llvm.org/docs/index.html

well-defined input specification, such as network protocols, file formats like HTML or XML or similar formats. More refined approaches might even use functions or methods of the PUT to encapsule random input data, to pass the programs input validation checks [26]. Such an approach might not present optimal, as the "carrier" function might discard or sanitize a lot of the input data, which can be provided through accessible functions. The Fuzzer Diane [27] tries to overcome this by identifying so called "fuzzing triggers" in the PUT's source code, representing functions that are called after the input validation checks, but before the input is prepared for further processing, for example network serialization or similar operations, thus limiting the amount of discarded input data.

Hybrid Fuzzing

As explained above, often it is not directly possible to classify fuzzers as either black-box, white-box or grey-box, as many fuzzers use a combination of different approaches, as well as mix of mutation-based and generation-based fuzzing, even intertwined with some type of machine-learning augmentation, as seen in 2.5. Especially more advanced fuzzing approaches, proposed in the recent years, often try to find a good compromise between very efficient fuzzing and a generalizable, easy to use and adaptable approach, thus combining different methods and previous proposals to create a hybrid project, combining the advantages of all described methods.

2.1.3. Overview of Traditional Fuzzing

In the past 10 years, there have been countless proposals on how to improve different aspects of traditional fuzzing approaches, focusing on input generation, fault observation and monitoring, input delivery or even crash analysis, just to name a few. Figure 2.1 shows the number of publications, which were published in the past 10 years solely in the IEEE Xplore Digital Library[2], and contained the term "fuzzing" in their title. Those publications were then counted and grouped by year, to create the figure below. As can be seen, the number of publications has been steadily increasing over the past 10 years, with a slight decrease in 2020, which can be attributed to the ongoing COVID-19 pandemic, which has led to a decrease in publications in general. The figure only shows works which were published on the IEEE Xplore platform, which is only one of many different platforms to publish research in this area, and only displays publications that include the term "fuzzing" in their title, disregarding all other works, that were either published on other platforms, or do not include the term "fuzzing" in their title, although they might still be contributing to the topic.

This shows, that fuzzing is a very active field of research, with many new approaches and ideas being published every year. Disregarding the fact, that there are countless

[2] cf. https://ieeexplore.ieee.org/Xplore/home.jsp

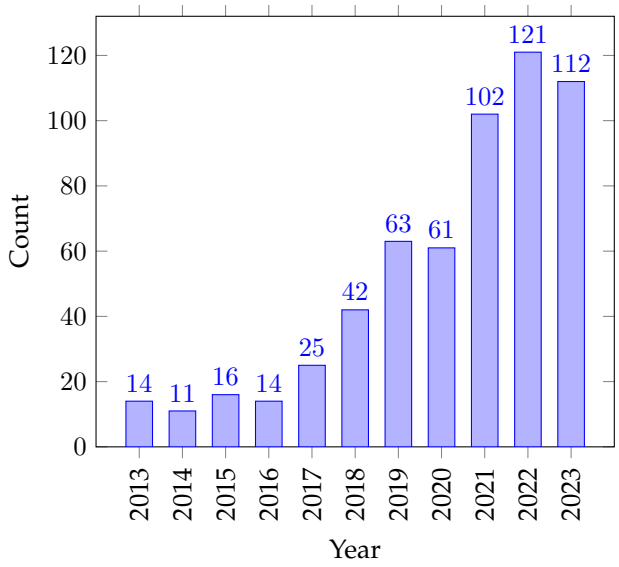

Figure 2.1.: Number of publications with the term "fuzzing" in the title on IEEE Xplore Digital Library in the past 10 years

promising approaches and projects, we want to highlight a few "milestone" ideas, that have shaped the world of fuzzing and built the basis for a lot of following research and popular projects, and can be seen as the state-of-the-art open source standards in traditional fuzzing. Proprietary fuzzers of course still exist, and might present at least comparable performance, but due to their closed source nature, it is difficult to compare them to open source projects, as they are not easily accessible and do not present a used basis for further research.

Popular Fuzzers

Many promising but often highly specialized fuzzing projects and approaches have been published, as we cannot highlight all of them, we will focus on those, which we deemed most relevant to the topic of this thesis. This includes projects, which built the basis for following research but also keystone approaches, that introduced new and promising ideas, which are still used today. Nevertheless, this will just be a brief overview to understand the domain of fuzzing, as this thesis will focus on a very specific problem of embedded fuzzing, which will be discussed in the following section.

American Fuzzy Lop (AFL) [3] [28] stands as a de-facto baseline standard in the realm of fuzzing tools, renowned for its speed and effectiveness as well as it's ease of use. AFL's distinctive feature is its innovative genetic algorithms, that refine test

[3] https://lcamtuf.coredump.cx/afl/

cases based on code coverage, based on a set of different random mutations and deterministic steps by incorporating different fuzzing strategies, proven to be effective in the past, an approach that has revolutionized fuzzing methodologies. The tool's ability to rapidly generate test cases while dynamically adapting to the code it tests, makes it a preferred choice for many security researchers, additionally different instrumentation tools can be used to monitor state transitions, which is used by AFL to quickly determine and incrementally refine interesting inputs further down the process. AFL's impact is evident in its extensive use and the numerous vulnerabilities it has helped uncover [29].

Its successor AFL++[4] [30][31] which is an enhanced fork of AFL, brings additional features and improvements to the original AFL design. It includes several optimizations and new functionalities aimed at increasing fuzzing efficiency and effectiveness. AFL++ integrates community-driven improvements, focusing on aspects like speed optimization, crash exploration, and advanced instrumentation. Additionally, AFL++ incorporates performance improvements and supports advanced fuzzing techniques like persistent mode and shuffling schedules, further optimizing the fuzzing process. This tool represents the collaborative efforts of the open source community to optimize what fuzzing tools can achieve and adapt tools for specific use cases, as it layed the baseline for many further adaptions and ideas, often extending it's capabilities to new or refined approaches.

AFLGo [5] [21], a variant of AFL, takes a new approach by focusing on targeted fuzzing. Unlike traditional fuzzers that explore code paths indiscriminately, AFLGo focuses on specific locations within the codebase, that are deemed to be vulnerable beforehand, enabling a more directed and efficient approach in scenarios where certain code regions are of higher interest. This is done by introducing a simulated annealing power schedule, that is able to incrementally prefer inputs triggering code closer to specified regions. Therefore interesting or relevant code segments can be analyzed comprehensively, which is particularly useful for testing patches and verifying specific bug fixes, making AFLGo a valuable tool for developers and security professionals focusing on specific code segments Böhme et al. were able to present 39 new bugs in commonly used libraries [21], that were already tested by other fuzzers before, thus showcasing the effectiveness of their tool.

VUzzer [32] is a tool that exemplifies the application of evolutionary algorithms in fuzzing. It employs an application-aware approach, utilizing data flow analysis to enhance the generation of test inputs, by assigning fitness scores to each input and refining those scores with control flow and data flow information. VUzzer's intelligence lies in its ability to accommodate to the application's structure, guiding the fuzzing process more effectively towards code areas deep inside the programs logic. This approach not only increases the chances of finding complex vulnerabilities but also reduces the overhead commonly associated with traditional fuzzing methods.

[4] https://aflplus.plus/
[5] https://github.com/aflgo/aflgo

During evaluation, VUzzer was able to find new bugs in evaluation datasets[6], as well as in real-world applications.

Skyfire [34] stands out for its specialized approach in fuzzing programs that process complex input formats like XML or HTML. It employs a grammar-based, data-driven approach to generate semantically valid test inputs according to a input model, which is then used to generate test cases that are more likely to hit and stress the nuanced parts of the code responsible for processing complex inputs. A so-called probabilistic context-sensitive grammar is employed to generate syntactically correct input samples, that at the same time is able to calculate the probability for each grammar rule to be active in a given context, thus creating seed inputs that are likely to increase code coverage. Those initial seed inputs are then fed to AFL for further fuzzing, decreasing the amount of redundant inputs generated by the same. Skyfire's contribution to the fuzzing domain is significant, particularly in the context of web security and the testing of applications that handle structured data extensively, like parsers or processors for XML, HTML, JSON or other highly structured text. Skyfire was able to uncover many different bugs in real-world applications, including Internet Explorer 11 [34].

2.2. Introduction to Machine Learning Techniques

This section will give a general overview on machine learning and its subclasses, as well as the most important concepts and techniques, which will be used in the following chapters. This section will not be a comprehensive introduction to machine learning, but rather a brief overview, highlighting the most important concepts, which will be used in the following chapters. For a more comprehensive introduction, we recommend the book "Artificial Intelligence, A Modern Approach" [35].

2.2.1. Artificial Intelligence

Google Cloud[7] defines Artificial Intelligence (AI) as "*a set of technologies implemented in a system to enable it to reason, learn, and act to solve a complex problem*" [36]. Machine learning (ML) is subsets of AI [37], Deep Learning (DL) again, is a subset of ML while reinforcement learning (Rl) can be both, a subset of the broader classification ML [38] as well as specific DL [39][40][41][42]. All of those subsets are used to describe a set of tools and approaches for different problems, hence differing in technical specifics, which leads to this classification. In this subsection, we will highlight important information on each of these topics, focusing especially on areas of research, that are important for the following chapters, in other words, approaches

[6] cf. https://www.darpa.mil/program/cyber-grand-challenge and [33]
[7] https://cloud.google.com/

that have already been used in research regarding enhancement of the input generation phase during fuzzing. This aims to guide the reader to the following sections, which will examine already used approaches and applications of machine learning to the input generation phase of both desktop and embedded fuzzers.

Machine Learning

Machine learning is a more specific field of Artificial Intelligence, that was defined by Arthur Samuel as "the field of study that gives computers the ability to learn without explicitly being programmed"[8]. In todays world, this means, that a machine learning algorithm is able to learn from data, which is provided to it, and make decisions or predictions over it, without being explicitly programmed to do so. Nowadays, machine learning is employed in countless tasks, such as image recognition, speech processing, chat bots, natural language processing, medical diagnosis, problems in the area of biochemistry, stock trading and many more, the goal always being to create algorithms that simulate human intelligence, which is why machine learning is often used synonymously with Artificial Intelligence. The basis of machine learning build mathematical optimization problems, so the field is tightly coupled with computational statistics.

Commonly, machine learning is divided into three subcategories, depending on the type of data the model is trained with, which are *supervised learning*, *unsupervised learning* and *reinforcement learning*. we will discuss reinforcement learning in depth below. Using supervised learning, models are trained on so called labeled data sets, where each data-point has some kind of description to it, which is called a label. This label can be a category, a class, a value or similar, depending on the problem the model is trained to solve. The model then to learns a function, that maps the input data to the correct label, which is then used to predict the label of new data points. If such labeled data does not exist, unsupervised learning can be used, where the model is trained on unlabeled data, finding patterns and structures in the data, which can then be used to cluster the data into different groups, what in turn can further be used to predict future developments of the respective pattern based on the learned information.

Deep Learning

During deep learning, neural networks, a specific type of machine learning model, are used. Neural networks aim to work in a similar way as the human brain, consisting of multiple connected processing nodes, called neurons, each of which doing its own processing on inputs and sharing outputs with other nodes. During deep learning, neural networks with many layers are used, with a varying degree of interconnectivity and data flow direction, depending on the specific model used. A common type of such neural networks is the so-called *feedforward neural network*,

[8] cf. [37][43], although he doesn't seem to have written that in his prominent papers [44][45]

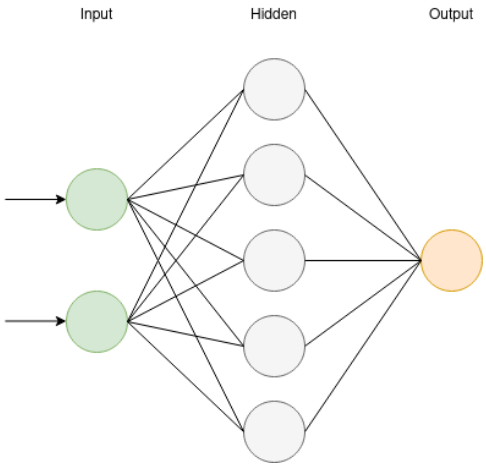

Figure 2.2.: Visualization of a Feedforward Neural Network, inspired by [46]

which consists of multiple layers of neurons, where each neuron is connected to all neurons in the next layer, but not to those in the same or previous layers. The initial or entry layer is called the input layer, the final layer is called the output layer, while all intermediate layers are defined as hidden layers. The input layer is used as an entry point to feed the model with data, which is then processed by some amount of hidden layers, those in turn providing the output layer with the result of the processing. The output layer then provides the result of the model, which can be used to make predictions or decisions.

The number of hidden layers is called the depth of the model, hence the name deep learning, while the number of neurons in each layer is called the width of the model. Each layer can correspond to a specific feature of the input data, processing the same to evaluate said feature. See 2.2 for a reference visualization of a feedforward neural network. Again, supervised and unsupervised algorithms exist, solving similar problems as the respective non deep learning algorithms. Applications of deep learning especially include problems, where vast amounts of data are available, often unstructured, as deep learning models are able to learn and extract complex patterns in the data, which in turn can be used to make predictions or decisions.

Recurrent Neural Networks

To mitigate possible limitations of feedforward neural networks, recurrent neural networks (RNN) were established [47][48]. This type of neural network allows for bidirectional information flow by enabling connections between neurons in the same layer, as well as connections to previous layers, thus allowing for a more complex processing of the input data, especially for connected sequences of input [49][50][51]. For connected sequences of data, where past data plays a role, RNNs offer a great performance increase, as they are able to remember past data and use it to process future data. One might see the possibilities, RNNs enable for fuzzing, as this problem

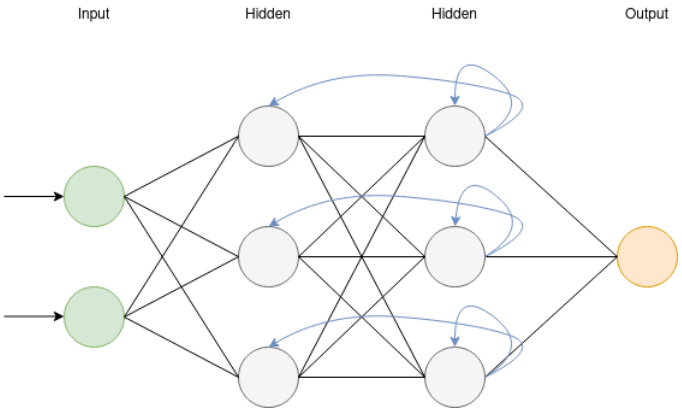

Figure 2.3.: Visualization of a Recurrent Neural Network, inspired by [46]

is based on the processing of sequences of data, where past data greatly influences the processing and selection of future data. See 2.3 for a reference visualization of a RNN, the blue looping arrows indicating possible connections. Classic RNNs are able to retain information on data processed in the very recent past, comparable to a very small short-term memory, making slightly larger connected sequences of input data already challenging to correctly process. Because of this, a special type of RNN, the long short-term memory model was introduced, as discussed in the next paragraph.

Nowadays, many natural language processing problems, where context plays a role, are better solved by using the Transformer architecture [52], that does not use recurrent connections, but rather uses attention mechanisms to process sequences of data by processing whole sections in contrast to single tokens. This architecture is not discussed further in this thesis, as in the context of fuzzing, transformers did not really gain popularity yet, but might be an interesting approach for future research.

Long Short-Term Memory Models

To improve the short-term memory capacities of RNNs, long short-term memory (LSTM) models were introduced [53]. Additionally to the short-term memory of RNNs, a new cell state is added, that retains important information for a longer time. This cell state is controlled by three gates, which are called the forget gate, the input gate and the output gate. The forget gate selects information, that should removed in comparison to the last state, the input gate chooses information to be added to the cell state and the output gate selects information, that will be outputted. This allows for a more complex processing of the input data, as the model is able to decide, which information is important and should be retained, and which information is not important and should be forgotten. See 2.4 for a reference visualization of the computing component of a LSTM model. The LSTM component receives an input $I(t)$ at time t, as well as the previous cell state $C(t-1)$ and the previous hidden

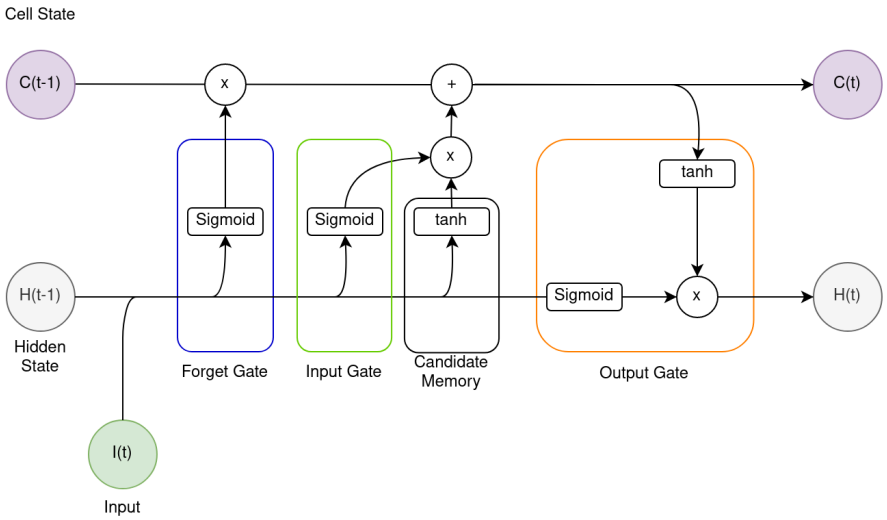

Figure 2.4.: Visualization of the computing component of a LSTM model, inspired by [54]

state $H(t-1)$. Those states are vectors, that store the necessary information. The forget gate then uses $H(t-1)$ and the current input $I(t)$ to calculate a selector vector, which is then multiplied with $C(t-1)$ to remove the selected information. Using the sigmoid activation function, the selector vector basically has the value 0 at positions in the vector, that should be removed, all other positions contain the value 1.

The new entry added to the cell state is calculated by the input gate and the candidate memory. They both receive a single vector, consisting of the concatenated $I(t)$ and $H(t-1)$, from which the input gate again calculates a selector vector to select the information to add from the normalized candidate vector, that is produced by the candidate memory using the $\tanh$ function. The result of this multiplication is then added as a new entry to the cell state. Finally the output gate calculates the output of this cell, which is again done using a selector vector and the normalized cell state, which is then passed to the next cell as a new hidden state. The output of the cell is then used to make predictions or decisions, depending on the problem the model is trained to solve. Each of those components are basically new independent layers in the complete architecture of the LSTM model, which are then connected to each other in a fashion, that is determined suitable for the problem the model is trained to solve.

Seq2Seq

Another machine learning approach, that has been applied to traditional fuzzing, is the Seq2Seq model [55], which is another type of neural network, that is able to predict sequences of tokens, thus making it useful for input generation tasks. In it's core, seq2seq is a encoder-decoder based model, both of which are LSTM models, that are

connected to each other. The encoder is used to encode the input sequence into a vector, capturing the context and essential information of the input in question. This vector is then used by the decoder to produce the final output sequence, predicting each element of the output sequence by taking all available information into account, even the previously predicted values. As an addition to the initial seq2seq approach, Bahdanau et al. introduced an improved model with an attention mechanism [56] to also be able to make predictions on long input sequences.

Reinforcement Learning

In his book "Artificial Intelligence, A Modern Approach" [35], which is the de facto standard book for introductory lectures to AI, Stuart Russell defines reinforcement learning (RL) as a type of machine learning, where "*an agent interacts with the world and periodically receives rewards (or, in the terminology of psychology, reinforcements) that reflect how well it is doing*" [35]. So this agent is not only trying to maximize the reward, but also has to interact with the environment, which is not the case in other ML approaches. Such an interaction is done by taking actions on the environment, whose result is then evaluated, which in turn provides the agent with a reward, which is in turn used to update the agent's policy, the function that maps the agent's state to an action. OpenAi[9] calls his process the "*agent-environment interaction loop*" [57], as shown in 2.5.

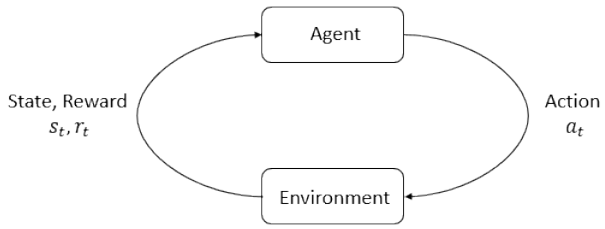

Figure 2.5.: Agent-Environment Interaction Loop, from [57]

In the context of this Thesis, we will mainly focus on a specific type of RL, called *Q-Learning* [58][59][40], which is a model-free RL algorithm, that learns a representation of action and corresponding utilities, utility in this case is the expected reward of an action taken in a specific state. This representation is called a *Q-Table*, which is a table of all possible states and actions, as well as the corresponding utilities, which are called *Q-Values*. The Q-Table is initialized randomly, and then updated during the training process, which is done by taking actions on the environment, which in turn provides the agent with a reward, which is then used to update the Q-Table.

Using the notation used in [60], ultimately introduced in [61], reinforcement learning and Q-Learning can be described as follows. The idea is based on a Markov Decision Process $\mathcal{M}$, which is a stochastic process that is defined as $\mathcal{M} = (X, A, P_0)$. X is the

[9] https://openai.com/ and https://spinningup.openai.com/

set of all possible states, A is the set of all possible actions and P_0 is the transition probability function, which returns the probability $P_0(U \mid x, a)$ for $(x, a) \in X \times A$ and $U \subset X \times \mathbb{R}$, which means if the actor performs action a while in state x, the system will shift to another state in X and the actor will earn a reward U, which is a real number. So for a transition from state x to state x', through performing action a, P_0 provides the transition probability $P(x, a, x') = P_0(\{x'\} \times \mathbb{R} \mid x, a)$, which is the probability of transitioning to state x' and earning a reward $r \in \mathbb{R}$, which is the expected reward of performing action a in state x.

To extend this to the idea of reinforcement learning, we introduce an agent, that over time learns to maximize the cumulative reward $\mathcal{R}$ earned over time, while causing state transitions by choosing an action and in turn observing the respective state transition in correlation with the earned reward. To prioritize rewards in the near future, we introduce a discount factor $\gamma \in (0, 1)$, which is used to calculate the discounted cumulative reward $\mathcal{R}$ as follows:

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{2.3}$$

Additionally, we introduce a policy π, which is a function mapping the state x to an action a, which is then used to determine the agent's behavior, who chooses an action a_t while observing x_t based on $a_t \sim \pi(\cdot \mid x_t)$, meaning a_t is distributed according to the policy of x_t. This leads to the expected cumulative reward

$$\mathcal{Q}^{\pi}(x, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid x_0 = x, a_0 = a\right] \tag{2.4}$$

Now to solve the problem, we just have to approximate the optimal $\mathbb{Q}$ function, which in the case of [60] is done by recalculating the Q-Table after each action, using the following formula:

$$\mathcal{Q}(x_t, a_t) \leftarrow \mathcal{Q}(x_t, a_t) + \alpha\left[r_t + \gamma \max_a \mathcal{Q}(x_{t+1}, a) - \mathcal{Q}(x_t, a_t)\right] \tag{2.5}$$

with α denoting the learning rate as a real valued number greater than 0 but at most 1. Now the agent can act according to the Q-function by performing an action for each state it observes, that maximizes the expected cumulative reward, which is done by choosing the action yielding the highest Q-Value for the current state $a_t = \arg\max_a \mathcal{Q}(x_t, a)$. This causes a state transition and is repeated until a time constraint is met or the problem is deemed solved, for example by winning a game.

Deep Reinforcement Learning

The approach to reinforcement learning described above works very well, when the state space is small, but for more complex problems, as the state space grows, the Q-

Table grows exponentially, which makes it infeasible to use. To overcome this problem, deep reinforcement learning (DRL) was introduced, which uses deep neural networks to approximate the Q-Function or in general the policy after which the RL agent acts on the environment. In case of Q-Learning, the resulting neural network is called a *Q-Network*, that is then trained using the Q-Learning algorithm, which is called *Deep Q-Learning*. If a model-based approach is chosen, usually a neural network is trained using a supervised approach, to finally predict the actions taken by an agent, those predictions are then used to actually act on the environment.

2.3. Embedded Fuzzing

After introducing the general concept of fuzzing and necessary fundamentals on machine learning techniques, we will now focus on embedded fuzzing, which is the main topic of this thesis. we will start by introducing the concept of embedded systems, followed by a classification of embedded systems, which will be used to differentiate between different types of embedded systems, as well as to highlight the differences between traditional fuzzing and embedded fuzzing. Subsequently, we will discuss the challenges of embedded fuzzing, followed by a general overview of the usual process of embedded system fuzzing, which will be used to highlight the differences between traditional fuzzing and embedded fuzzing. Finally, we will introduce the current state of the art embedded fuzzers, which will be used as a basis for the following chapters, as well as to answer RQ 3.

Similar to traditional fuzzing, fuzz testing of embedded devices is a very active field of research, especially in the recent years, as the count of used embedded systems in many different industries increases every year. Figure 2.6 again shows the number of publications per year on the platform IEEE Xplore in the past 10 years, containing the terms “embedded” and “fuzzing” in their metadata. The market for embedded systems is growing every year[10], even after the semiconductor crisis during COVID-19[11], highlighting the increasing need for sophisticated testing approaches to ensure the security, privacy and continuity of devices especially in critical infrastructure, automotive applications and medical devices, as well as consumer electronics.

2.3.1. Embedded Systems

Generally speaking, an embedded system is a device, which is designed to be embedded into a larger system of electrical components, mostly sensors or similar items, that interacts with the surroundings in some way, and which is developed and integrated to fulfill a specific purpose. But also devices like smart home devices, which are not directly embedded into a larger system, but rather interact with the surroundings in a more direct way, as well as mini computers like a Raspberry Pi Zero 2.7, or even Wifi routers, can be considered embedded systems.

[10] cf. [62]
[11] cf. [63]

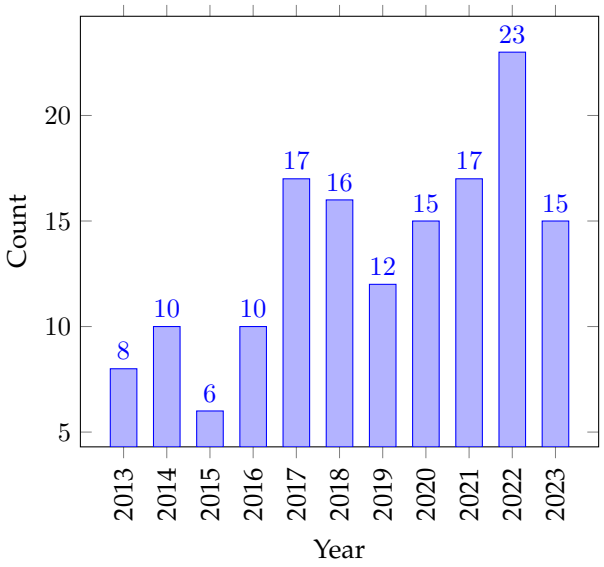

Figure 2.6.: Number of publications with the terms "fuzzing" and "embedded" in all metadata on IEEE Xplore Digital Library since 2013

Embedded systems are based on a microprocessor, often in combination with memory and some way to interact with peripherals, and are usually designed to be as small and energy efficient as possible, as well as being resource efficient, while still fulfilling their dedicated tasks. See 2.7 and 2.8 as examples of such a microprocessor based device including peripherals to interact with the surroundings. Nowadays, such devices can be found in many different use cases, spanning from critical infrastructure over automotive applications to consumer electronics like smart home devices. Especially in larger systems, usually many different embedded devices are used, all of which fulfilling their specific purpose while still interacting with the other devices to accomplish the overall purpose of the system as a whole. It is important to note, that in the context of this thesis, we will not distinguish between the terms embedded system and embedded device, as the purpose of this thesis is focused on singular embedded devices, not interconnected systems of multiple embedded devices.

Typically, the architecture of embedded devices can be abstracted as seen in 2.9, consisting of a hardware layer, a system software layer and an application software layer. The hardware layer consists of the actual hardware, such as the microprocessor, memory, peripherals, which are used to interact with the surroundings. The system software layer consists of firmware, that contains the bootloader,the operating system as well as device drivers to manage the hardware and, in general, provides interfaces to the application software layer. The latter eventually consists of the actual applications, if such are necessary to fulfill the purpose of the device. The application software layer is the only layer, which might be visible to the user, as it is the only layer, which is directly interacting with the surroundings in a matter a human user can actually interact with, apart from hardware interactions through

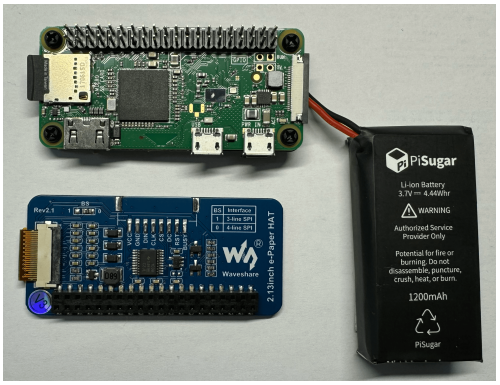

Figure 2.7.: Raspberry Pi Zero, a mini computer based on a microprocessor, including a standalone battery and a small display

Figure 2.8.: Asus RT-AC58U, a Wifi router based on a mipsel_74kc microprocessor, with antennas and further I/O ports

sensors and actors. But as the formulation of this explanation already implies, this abstraction is not always applicable, as there also exist many devices, which do not have a dedicated operating system, but rather use a monolithic firmware, that is directly interacting with the hardware, lacking the abstraction of further software layers. Furthermore, especially in the context of embedded fuzzing, it is important to note, that the application software layer is not always visible to the user, as it might be hidden behind some kind of interface, for example a network interface, which is only accessible through dedicated network connections, or a serial interface, only accessible through a serial connection.

While many different architectures for embedded devices exist, for example RISC-V [64], MIPS [65] or ARM based architectures [66], this work will mainly focus on ARM Cortex-M standard, which is a 32-bit architecture using a reduced instruction set (RISC) and is widely adopted in practice as well as predicted to still gain popularity in the future [67].

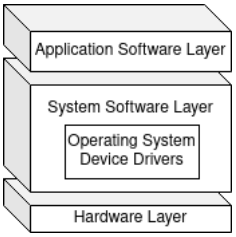

Figure 2.9.: Example architecture of embedded systems after Noergaard [68]

2.3.2. Classification of Embedded Systems

Classification of Embedded Systems can be done in many different ways, but especially for fuzzing, it is useful to distinguish them by differentiating between the used operating system (OS), because it is mostly responsible for ensuring the systems continuity, thus handling fault recovery as well as more advanced security mechanisms and providing interfaces to ensure the fulfillment of the dedicated purpose.

In this context, Muench et al. [69] distinguish between three different types of embedded systems.

The first class are devices, which are based on a *General-Purpose OS*, which is mostly an OS based on the Linux Kernel but with minimal user environment. Those devices are often not as specialized as their counterparts in the other classes, suffering from suboptimal runtime conditions as they are using an repurposed operating system not specifically made for embedded devices. The second class contains devices, which are based on *Custom OS*, that is created solely to fit embedded devices. Such OSs are designed to be used on devices with limited computational resources and limited architectural capabilities, such as missing Memory Management Units or reduced instruction sets, increasing the usability and performance while being energy and resource efficient, one example for such an *Custom OS* being the widely used *ZephyrOS* [12].

The last class consists of highly specialized devices without a direct Operating System abstraction, adopting *monolithic firmware* to serve a highly specific task. Such *monolithic firmware* is mostly just a simple control loop handling interrupts that are created by peripherals to handle events and control actions on the surrounding environment.

Other types of classification exist, such as classifying systems according to their performance, the performance of the used micro-controller, the used architecture, their purpose or even the industry field they are used in, but those classifications are not as useful in the context of software testing, as they don't engrain important runtime information like the used OS, which is essential for fuzzing.

[12] cf. `https://zephyrproject.org/` for further details

2.3.3. Challenges of Embedded Systems Fuzzing

Additionally to challenges that might occur during the general fuzzing process, embedded fuzzing presents multiple unique challenges, which have to be overcome to create a sophisticated fuzzing approach. Muench et al. highlighted those challenges in their paper [69], which we will discuss in the following.

One of the biggest issues during embedded fuzzing is the fault detection, on which the process heavily relies. In contrast to desktop computers, embedded devices are not designed to have a lot of fault protection measures, that lead to a crash during unexpected behavior, or are even designed to simply do a silent restart after encountering a fault. Furthermore, especially productive devices often lack any kind of I/O possibility for the user to interact with the device, thus making it impossible to detect a fault, as the device is not able to communicate it to the user. To enable efficient fuzzing of such devices, sophisticated instrumentation must take place, to detect faults and be able to use such information to guide the testing process. This instrumentation presents the next challenge, as it is often not trivially possible to instrument the program or system under test, as it is not possible to install any kind of software on the device, or the system under test is not able to communicate with the outside world, thus making it impossible to use any kind of instrumentation. To overcome this, it might be necessary to use some kind of emulation, which allows for dynamic instrumentation and analysis of the PUT's runtime behavior, which will be discussed in 2.3.4.

Especially during fuzzing based on the actual device, another challenge to overcome is the performance of the device itself, which is generally much less powerful than a desktop computer, impacting the efficiency and speed of automated testing, as well as scalability fuzzing instances, which normally is easily possible during traditional fuzzing but presents a big challenge for embedded devices, as multiple actual devices are needed to scale the fuzzing process.

To overcome the limitations of fuzzing on the actual device itself, a common approach is to emulate the hardware, which will be discussed below. While this solves the resource constraints and possibly instrumentation issues, an emulation approach presents its own challenges. The correct representation of hardware in a virtualized environment requires a deep understanding of the devices specifications, especially in more complex devices. If such information is not available, it is often necessary to reverse engineer the device, which is a very time consuming task, that requires a lot of manual work. Especially if no emulation configuration is available for the chosen emulator, meaning the device is not yet supported by it, it is necessary to manually configure the emulator to correctly represent the device, which is a laborious task, necessitating deep knowledge and a lot of time.

2.3.4. Classification of Embedded Systems Fuzzing

Because of the challenges discussed above, different types of fuzzing approaches exist for embedded systems, mostly classified by the representation of the actual firmware of embedded application in the process. The most common classification differentiates between *Hardware-Based Fuzzing* and *Emulation-Based Fuzzing*, which will be discussed in the following, while especially focusing on different types of emulation, as this is the most commonly used method to perform embedded fuzzing.

Hardware Based Fuzzing

The first thing coming to mind when thinking about fuzzing of embedded devices, probably include the device itself somewhere in the testing process, as it is the target of the whole procedure. In fact, different approaches utilizing the device itself exist, but they are not the only way to fuzz embedded software. Usually, the device to be tested is hooked into some kind of fuzzing loop, that often includes a proxy to communicate with the device, for example to forward the input generated by a fuzzer running on a stronger desktop machine to the device and receiving debugging information back to feed to the fuzzing engine. This can be done in many different ways, including using proxies like gdb-server[13] [70], debug interfaces available on the hardware [71][72][73], as well as execution directly on the hardware, which is especially possible in stronger devices with general purpose operating systems, like a Raspberry Pi[14] [74], but also includes very sophisticated approaches, that require deep understanding of the hardware being tested [75][76][77]. Other authors even propose fuzzing on network interfaces provided by the system under test [26][78], which in turn only enables the testing of network facing applications but at the same time eliminates many challenges regarding correct hardware representation and input formats. A hardware-in-the-loop situation is illustrated in Figure 2.10.

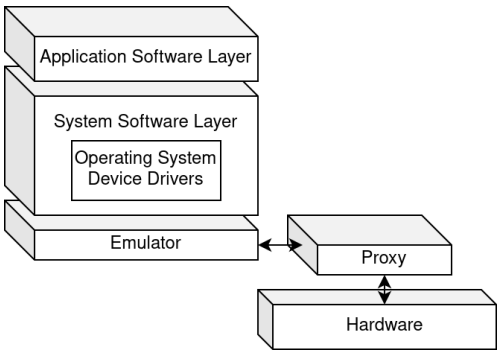

Figure 2.10.: Hardware-in-the-loop setup for fuzzing

[13] https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_130.html
[14] https://www.raspberrypi.com/

Emulation Based Fuzzing

As already introduced above, emulation based fuzzing is a promising approach to fuzzing embedded devices, as it allows for dynamic instrumentation and analysis of the PUT's runtime behavior, enabling a perfect observation capabilities during runtime, which is necessary to guide the fuzzing process. As emulation is a very broad term, we will now introduce the different types of emulation. The most commonly adapted emulation tool is QEMU [79], which is an open source emulator and virtualizer, that is able to emulate different architectures, such as ARM, MIPS, PowerPC, RISC-V, SPARC, x86 and x86-64, as well as different devices, such as network cards, graphics cards, sound cards, USB devices and many more. Muench et al. [69] showed, that emulation based fuzzing even improves the performance in comparison to fuzzing on the hardware itself. This is due to the fact, that the emulating host system is usually more powerful than the target system, thus allowing for much stronger hardware support, such as multiple cores, which can be used to parallelize the fuzzing process, as well as more memory, which can be used to store more inputs and thus increase the fuzzing efficiency.

For a successful emulation, the emulator must be capable of simulating hardware interrupts, direct memory access (DMA) and memory-mapped I/O (MMIO). Hardware interrupts are used by the hardware to signal the CPU, that a specific asynchronous event has occurred, for example a network packet arrived over a network cards serial port, enabling the CPU to directly handle such events. DMA is used by the hardware peripherals to directly access the memory, without CPU handling the access, thus increasing the performance of the system. MMIO lets the CPU to directly access the memory of a hardware peripheral, enabling it to get information about the state of said peripheral or trigger actions on the peripheral. For example, this enables the CPU to get information about a GPIO button being pressed or to trigger an action on a GPIO pin, for example to turn on a LED. DMA accesses are also handled through MMIO. A successful emulation must be able to handle such actions correctly, as they are essential in representing the real device in virtualization. Typically, this so-called firmware re-hosting is a very complex task, requiring a lot of manual configuration, if the hardware to emulate is not already supported by the emulator.

To emulate the hardware for a firmware image, different techniques exist, which are discussed in the following.

User-Mode Emulation tries to emulate only the application software part of the PUT, while still running the PUT on the host's kernel, see 2.11a as an adaption from 2.9. By doing that, only the binary file gets emulated, all system calls to the PUT are then translated to system calls to the host's kernel, which is only possible, if both the host and the PUT have the same kernel or a customized kernel can be crafted and used. As interfaces in the application layer are usually well defined and documented, user-mode emulation is often easy to use and setup, but all potential hardware accesses have to be translated and treated correctly by the emulator, thus leaving lots of room for non-representative results due to system states that are not

(a) User-mode emulation **(b)** Full-System Emulation

Figure 2.11.: Different types of emulation, adapted from Noergaards architecture [68]

consistent with real-world system states. Firmadyne [80] as well as its improved adaption FirmAE [81] use this approach to emulate the PUT using QEMU, which is then used to perform dynamic analysis and instrumentation, for example guiding the fuzzing process. It is important to note, that Firmadyne technically uses QEMUs full-system emulation, but because it utilizes a custom kernel and only really implements user application fuzzing, it is better counted towards user-mode emulation. Another project, that enhanced Firmadyne is FirmAFL [82], which in turn tries to increase the performance of the fuzzing process by utilizing user-mode emulation as long as possible, only switching to full-system emulation, if necessary.

Full-System Emulation emulates, as the name implies, the full system as a guest on the host system, including CPU, kernel, operating system as well as peripherals, see 2.11b as a reference. Using full-system emulation, it is possible to completely instrument the binary firmware, enabling all types of fuzzing, even white-box fuzzing, as the complete system can be instrumented. To achieve full-system emulation, a correct software representation of all hardware-accesses made by the firmware, as well as interrupts made by the hardware, is necessary, so that the firmware binary can then be executed in an environment, that perfectly behaves as if it was executed on the hardware itself.

As already mentioned above, QEMU has the ability to emulate a lot of different architectures in full-system mode, enabling users to quickly start projects by already providing a lot of existing configurations of microcontrollers, as well as peripherals to emulate. Therefore, for a long time, QEMU has been the de-facto standard tool to use for emulation based fuzzing, as a lot of work is based on it [83][84][85][81][80][82]. Another popular project, called libFuzzer [86], emulates the system through so called virtual prototypes, that model the entire hardware using SystemC[15], a system design and verification language. As already described above and especially being shown in 2.10, hardware based fuzzing might also be using full-system emulation to support a hardware-in-the-loop approach, that emulates the firmware or application to test, while forwarding hardware accesses and in return interrupts through peripheral proxying to the actual hardware [83][84][85][87]. While those approaches are very sophisticated and powerful, they are also very complex and require a lot of knowledge about the hardware being tested, thus being very specialized and not

[15] https://systemc.org/overview/systemc/

easily adaptable to other projects.

In some cases it might be better to model the hardware automatically instead of using the actual hardware or a perfect emulation. Such a model can be created by monitoring the actual device and saving accesses to the MMIO, reusing those accesses as well as interrupt timings for dynamic analysis [88]. More sophisticated approaches suggest answering MMIO accesses with data generated by the fuzzing engine[89] and even combining this with dynamic symbolic execution[90].

Unicorn Engine [91] is a lightweight multi-platform, multi-architecture CPU emulator framework based on QEMU, which is able to emulate different architectures, such as ARM, MIPS, as well as a lot of different devices. It is based on QEMU, but is much more lightweight, as it does not emulate the whole system, but only the CPU, as it doesn't translate system calls and POSIX signals to the host system, but rather only translates the CPU instructions. Furthermore Unicorn is built as a framework and therefore offers a very powerful API, which enables the user to instrument the emulated firmware, for example to perform dynamic analysis, such as code coverage analysis, as well as easily control memory registers and set native hooks, that are triggered when certain memory addresses are reached. These features come as a much lighter and faster package, Unicorn even being able to emulate binary code without a complete execution environment, unlike QEMU. All those possibilities make it an ideal tool for embedded fuzzing, AFL++ integrated it to be used instead of QEMU.

2.3.5. Popular Embedded Fuzzers

As already implied, there is a lot of existing research in the field of embedded fuzzing, which is why we will only highlight a few of the projects, which we deemed most relevant to the topic of this thesis. This includes projects, which built the basis for following research but also keystone approaches, that introduced new and promising ideas, which are still used today, as well as projects that can often be found in literature regarding embedded fuzzing.

Firmadyne

As already described above, Firmadyne [80] utilizes QEMU to emulate the user space applications of a firmware image, performing automatic dynamic analysis using this emulation. A custom kernel is used with QEMU to run the extracted firmware binary image, which can then be tested according to the specifications. While technically being a framework focused on the correct emulation of such firmware, it enables dynamic analysis and instrumentation, which can also be used for fuzzing approaches. Nevertheless, the authors evaluate the effectiveness of their framework by testing known exploits on the emulated firmware, to show, that the emulation in fact represents correct real-world states of the device.

Firm-AFL

Firm-AFL [82] is a dedicated fuzzing tool designed to address the complexities of firmware analysis. Building on the foundational concepts of Firmadyne and AFL, Firm-AFL tries to enable a faster fuzzing process based on QEMU. They solve performance issues of full-system emulation by running targeted applications in user-mode emulation as long as possible, only using full-system emulation for necessary system calls. This approach is able to increase the fuzzing speed by up to 10 times. The only real constraint, apart from the obvious dependency of possible emulation with QEMU, is the restriction, that the target must be based on a POSIX-compatible operating system.

μAFL

μAFL [71] is a hardware-in-the-loop approach to fuzz the firmware of microcontrollers. Based on AFL, it uses existing debugging tools intended for development of embedded devices to feed back coverage and crash information to the AFL instance, while introducing a new way of interpreting code coverage from raw debugging information, provided by ARM ETM hardware debugging. By utilizing an existing hardware debugger, the approach is non-intrusive and easy to setup, especially in industrial settings, where a board with respective debugging pins can easily be acquired during development. During evaluation of their tool, the research team was able to find 13 zero-day bugs in the firmware of NXP and STMicroelectronics microcontrollers, which were previously unknown to the vendors.

Fuzzware

Fuzzware [90] presents another interesting and well researched tool, especially for fuzzing monolithic firmware. Fuzzware distinguishes itself through its use of model-based fuzzing techniques, while coupling execution of input with Dynamic Symbolic Execution. The main innovation is the direct fuzzing of MMIO accesses, providing data generated by AFL or AFL++ directly as a response to peripheral accesses. By doing that, Fuzzware is able to precisely emulate a lot of different system states, thus reaching high basic block coverages during the testing process. Fuzzware's methodology is particularly beneficial for testing firmware that operates in tightly coupled hardware-software ecosystems. Presented with the approach and its implementation, the authors conducted a very comprehensive evaluation, extensively fuzzing 21 real-world monolithic firmware samples, previously used in works like μEmu [92] and P2IM [89], being able to achieve an increased code coverage by $\sim 61\%$ and $\sim 44\%$ respectively. Furthermore, they were able to find 12 new bugs in total while fuzzing Zephyr[16] and Contiki-NG[17], two widely used frameworks for embedded firmware.

[16] https://zephyrproject.org/
[17] https://github.com/contiki-ng/contiki-ng

FirmAE

FirmAE [81] is a comprehensive fuzzing framework designed for automated analysis of firmware in embedded devices. It is based on Firmadyne, aiming to improve the emulation capacities of said tool, claiming that many difficulties, presenting itself during the use of Firmadyne, can be avoided with more sophisticated heuristics targeting parameter and interface configuration. They claim that using their heuristics, FirmAE is able to automatically run 79.36% of tested firmware images, while Firmadyne was only able to run 16.28% of the same set of firmwares [81]. This represents a significant advancement in the field by automating the process of firmware emulation and, further down the road, in fuzzing. FirmAE is capable of automatically extracting and emulating firmware images from a wide variety of embedded devices. This automation greatly reduces the manual effort involved in setting up a fuzzing environment, making it accessible to a broader range of researchers and practitioners. Similarly to Firmadyne, the focus of this project is again not the fuzzing process itself, but rather the steps enabling dynamic instrumentation. Nevertheless the authors also implemented a simple fuzzing engine, which is integrated in the framework for testing purposes.

2.4. What is the Usual Process of Embedded System Fuzzing (RQ1)

After a comprehensive literature review, we can now answer Research Question 1. The answering of this question takes many different approaches into account and generalizes them, as the previous research is often focused on a very specific use-case, hence only focusing on a part of the process. Figure 2.12 shows the general process of embedded fuzzing, which is based on the general process of traditional fuzzing, but highlights the differences between the both, mainly the configuration overhead beforehand, as well as the different approaches to embedded fuzzing, which will be discussed in this section.

Target Selection

The process begins with the selection of the target to fuzz. The device to test can be selected based on different criteria, such as the used architecture, operating system, firmware, hardware or even the used peripherals, but for most industry use cases the selection will adhere to a given specification. This can be the testing of a developed product or product in a company The selection of the target is important, as it determines the following steps, such as the configuration of the fuzzing environment, the selection of the fuzzing approach, as well as the selection of the fuzzing tools.

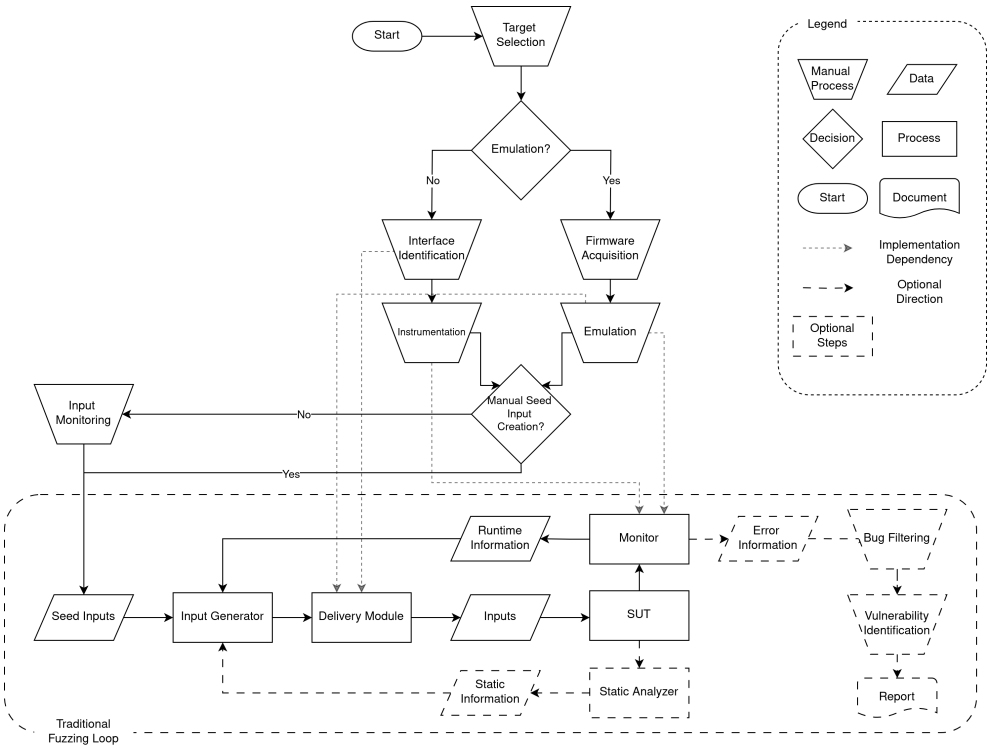

Figure 2.12.: General process of Embedded Fuzzing based on different approaches.

Fuzzing Approach

The second step is to determine, whether an emulation of the system is necessary or beneficial, or if it might be better to use a hardware based approach. This decision is based on the target's architecture, as well as the used operating system. If the target is based on a general purpose operating system, such as Linux, which can be used to perform dynamic analysis and instrumentation, it might be possible to use a hardware based approach, especially if the resources of the device allow for high throughput. If the target is based on a custom operating system or even on monolithic firmware, it might be necessary to use an emulation based approach, as it is not possible to perform dynamic analysis on the target itself. This decision is also based on the used architecture, as it determines the possible emulation approaches, as well as the used fuzzing tools, as not all fuzzing tools support all architectures. Another selection criteria could be the threads to validity.

As mentioned above, an emulation based approach might not represent the correct state, the device would be in, in the real world, thus being error-prone to false positives and negatives. While a hardware based approach might not be able to reach the same throughput as an emulation based approach, it might be more accurate and is still worthwhile to think about it, especially in later stages of development, to ensure the device's security and continuity. Depending on the decision made, different

paths in the process are taken, which are discussed below.

Emulation based Approach If an emulation based approach is chosen, the next step is to configure the emulation environment, depending on the target specification.

1. **Acquire Firmware Image**

   The first step is to acquire the firmware image, which can be done in many different ways, depending on the target. In an industrial setting, this step is often trivial, as the firmware image is readily available, in an open source setting, this might present some difficulties, as especially vendor specific firmware might not be publicly available, as they might contain sensitive information, such as cryptographic keys or other intellectual property.

2. **Extract Firmware Binary**

   The next step is to extract the firmware binary, which can be done using different tools, such as *binwalk* [93] or *OFRAK* [94].

3. **Emulation Environment Setup**

   With the extracted firmware, the emulation environment can be configured depending on the firmware's specifics, for example using Unicorn or QEMU, as the firmware binary is usually not able to run on the host system, as it is not compatible with the host's kernel. This step is especially important, as it determines the possible fuzzing approaches, as well as the possible fuzzing tools, as not all fuzzing tools support all architectures.

Hardware based Approach If the decision falls on a hardware based approach, different steps have to be taken.

1. **Interface Identification**

   Available interfaces for data flow have to be determined, so a fuzzing approach can be chosen. Usually this is done by analyzing the target's architecture, as well as the used peripherals, to determine possible interfaces, such as UART, SPI, I2C, JTAG or similar interfaces[18].

2. **Interface Configuration**

   After the interfaces have been identified, the next step is to configure them, so they can be used to communicate with the device. This might include the usage of a hardware debugger to connect to the device, as well as tools to enable communication over debugging interfaces.

[18] cf. [95] for a small overview of such interfaces

3. **Environment Setup**

   Possibly the fuzzing process is done on a different host machine [70], which utilizes those interfaces to send data to the device and receive information about the PUT's runtime behavior, such as crashes, power consumption or similar metrics. This might present a problem, as often there is no trivial way to instrument the PUT, thus making it difficult to observe the PUT's runtime behavior, which is necessary to guide the fuzzing process. Such difficulties can be mitigated by using a hardware-in-the-loop approach, only sending some information to peripherals to model the correct real world behavior during sophisticated hybrid emulation as described in 2.3.4.

After the desired path has been chosen, the fuzzing process can be started according to the following steps, continuing the step numbers of either approach.

4. **Seed Corpus Creation**

   The initial inputs to the fuzzing process are called the *Seed Corpus*. This corpus has to be created one way or the other, which presents the next decision path to take.

   a) Depending on the specifics, it might be necessary to create a sophisticated seed corpus by hand, to conform to a specific input specification, which especially holds true for model-based fuzzing approaches or when the fuzzer should be directed to specific code areas [21, 20].

   b) Automatic seed generation can also be utilized, either by simply monitoring the program during normal behavior or by using a sophisticated approach [34, 96, 97], even as sophisticated as machine learning algorithms, which will be discussed in 2.5.

   c) Many researchers came to the conclusion, that a random seed corpus is sufficient to start general purpose fuzzing, as the fuzzer will mutate the inputs anyway, thus creating a seed corpus by hand is not necessary [90]. This is especially true for black-box fuzzing, as the fuzzer does not have any knowledge about the PUT's internal structure, thus it is not possible to create sophisticated inputs conforming to a specific input specification, but also for more sophisticated grey-box fuzzers, as the overhead of creating inputs is often not needed.

5. **Fuzzing Loop** After all these steps have been completed, the fuzzing loop can be started. This loop consists of the following steps:

   a) The fuzzer selects an input from the seed corpus, which is then mutated by some kind of Input Generator according to the fuzzer's mutation strategy.

   b) The mutated input is then sent to the PUT using a delivery module, which

depends on the approach chosen, either being a harness utilizing the emulator or a module implementing the identified hardware interfaces.

c) The PUT is then monitored for crashes or other anomalies, which again varies according to the selected approach. If a crash is detected, the fuzzer will save the input, as well as the crash information, especially the state of the device.

d) During each run the fuzzer feeds back information to the input generator, this can include code coverage information, crash information or other metrics, which can then be used to guide and improve the fuzzing process. The improvement in this case depends on the used fuzzer, but for most projects, this includes the adaption of the input being mutated according to increased code coverage, meaning that an input which increased code coverage will be more likely to be mutated again, as it is deemed to be more promising [30, 82].

This loop is then repeated until the fuzzer is stopped, or the PUT is deemed to be bug-free, which it of course never completely is. After the fuzzing process, those saved inputs can be used to analyze the device and reproduce the errors, so it can be determined whether this bug is unique, as well as its severity according to possible impacts or exploitable vulnerabilities. Using this information, a bug report can be created, which in turn can then be sent to the developers, who can then use it to fix the bug. Another optional step would be a possible static analysis of the system state during each fuzzing loop, if the amount of instrumentation allows it. This creates the possibility to feed back more information about the impact of a specific input, which can then be used to guide the fuzzing process.

2.5. Previous Research on Machine Learning Applications in Fuzzing

This section will discuss previous approaches to the application of machine learning techniques to the fuzzing process, focusing on the input generation phase. This section will also answer Research Question 2 and Research Question 3.

The following subsections will be structured as follows: First, we will introduce the concept of machine learning augmented fuzzing, followed by a discussion of the different approaches, focusing in detail on especially relevant work, as well as a discussion of the current state of the art in this field. After that we will highlight some interesting approaches to the application of machine learning in embedded fuzzing, which will conclude in the answering of Research Question 3.

2.5.1. Machine Learning Applications to Input Generation in Traditional Fuzzing (RQ2)

In the recent years, many different ideas and projects were published, applying machine learning to the different stages of the fuzzing process. To improve the input generation phase of fuzzing, mainly two different ideas are adopted, the first one being the application of a machine learning model to generate inputs in each fuzzing loop, the second one being the selection of an operation to perform on the input, which is predicted by the model.

Initially, a common application was the introduction of genetic algorithms to the process [110][111][112]. Such algorithms are created to adhere to the process of biological evolution, naturally sieving out "unfit" candidates to support better ones. The fitness of a candidate, in the case of fuzzing, this would be an input string, is determined by a fitness function, which is used to evaluate the candidate. Over the time of the fuzzing process, the genetic algorithm is then able to select inputs, initially from the seed corpus, later from the corpus updated through random mutation strategies, that possibly increase some kind of metric better than others. Such a metric can technically be defined as any information, that is gainable during the testing process, but historically, some kind of code coverage in combination with information like the number of found crashes or the time an input runs in the system have been used. A very sophisticated project, that is based on genetic algorithms and still in use today, is AFL [29] and its successor AFL++ [31], which we already talked about. While still being a kind of artificial intelligence, genetic algorithms are not really the thing that comes to mind, when you talk about machine learning, therefore the focus of this section will be elsewhere.

As can be seen in table 2.1, many approaches used some kind of recurrent neural network in their work, as they are perfectly suited for predicting sequences. In the column "Fitness Function", some entries are surrounded by brackets, meaning that in this case the corresponding metric is used as an evaluation metric or fitness function used by other heuristics, not in a feedback loop to the machine learning model.

Rajpal et al. present Augmented-AFl [98], where they propose models based on LSTMs, as well as seq2seq, that are able to predict good locations in input strings, that should be mutated to increase code coverage. The models are trained in a supervised fashion, using a dataset of input strings, that are labeled with the code coverage they achieved in past fuzzing runs using AFL. During fuzzing, the model then is used to predict whether an input is good or bad, using a veto technique to discard of badly mutated inputs. The used models were a standard LSTM, a bidirectional LSTM, as well as both, the initial seq2seq approach and the attention based seq2seq approach.

Cheng et al. [97] presented another applications of RNNs in combination with seq2seq, where they trained a RNN to predict program paths, which were then fed to a seq2seq model that created inputs to trigger those paths. Another very popular project utilizing RNNs in combination with seq2seq is Learn&Fuzz by Godefroid et al. [96]. They trained a RNN model to generate syntactically correct inputs for a PDF

reader, trained on a large corpus of valid PDF files. During the evaluation process, they realized, that a problem with training an RNN based model on correct files presented. The model learned to always create perfect input strings, which is not useful for fuzzing. Therefore they introduced a custom sampling strategy, to change the created input strings, so that the fuzzing process could be improved. In a similar fashion, Fan et al. [99] proposed a seq2seq model, that was trained on a large corpus of network traffic, to learn an input grammar and generate input strings accordingly. While they use code coverage as an evaluation metric, the model is solely trained on correctly created input, the focus here is to create input, to create malformed strings, a sampling method is used, that combines various template strings.

Paduraru et al. [100] also used seq2seq models to generate inputs, but in contrast to the other approaches, they used a different technique to train the model. They automated as much as possible, even receiving different input file types as an initial training corpus, learning on each file type provided, thus reducing the work, the user has to do and increasing supported types. Again a custom sampling method is employed to chose what characters are created by the model and which are randomly added. Similar approaches are used by Cummins et al. [101] training a seq2seq model to create correctly formatted input for fuzzing compilers. Sablotny et al. [102] and Nasrabadi et al. [103] again use similar approaches based on LSTMs, trained to generate input data, which is then altered during the fuzzing process, Nasrabadi et al. employed a custom algorithm, that queried the trained model for an input string and then changed parts of the string with characters, that hav a low probability assigned by the model.

A different approach to augment the fuzzing process through machine learning was presented by She et al. [104] and Chen et al. [105]. They used machine learning to model the programs behavior, which was then used to predict path constraints dependent on a given input. She et al. proposed NEUZZ [104], which models the programs behavior as a smooth continuous function, that is calculated using a Neural Network and then used to predict branching behavior presented with a given input. Angora, proposed by Chen et al. [105], calculates a discrete function, that represents the program's path up until a specific branching constraint, using gradient descent to find specific inputs that solve the calculated function. Both approaches are then used to guide a fuzzer to specific code areas, both being successful during evaluation.

In contrast to those generation-based applications of machine learning augmentation, that often are not able to adapt the created inputs during the fuzzing runtime, thus not employing the feedback loop, that is so important for efficient fuzzing, some proposals exist, that use reinforcement learning based fuzzing augmentation.

Becker et al. [106] introduced reinforcement learning to fuzzing by employing the SARSA algorithm [113], a modified RL algorithm that uses the same policy for acting and updating the Q-Values, to mutate network packets for fuzzing the IPv6 protocol. States are defined as the current network packet and its response, actions are defined as the possible mutations of the packet and the reward is defined as a combination of the number of functions called, the number of error invocations and the delay or corruption of the program response. Böttinger et al. [60] leverage deep Q-learning to

improve the fuzzing process. They defined states as the current input, actions consist of rewrite rules to apply to this input and the reward was calculated based on a combination of code coverage and execution time. Both works experimented especially with different reward functions and came to the conclusion, that the correct definition of this function is a crucial step to a successful model, as small changes in the reward calculations had big impacts on the fuzzing process.

Drozd et al. [107] propose a fuzzing framework called FuzzerGym, which integrates reinforcement learning based on OpenAi Gym[19] with libFuzzer [86] to create a sophisticated fuzzing environment based on Deep-Double-Q-Learning. The state in this case is defined as the program state reported by the llvm engine, actions are based on libFuzzers mutator actions, the reward is defined as the code coverage achieved by the action. This paper will be discussed in detail below. Another study, that applied reinforcement learning to fuzzing was conducted by Liu et al. [108], who used a Deep Q-Learning based approach to guide the fuzzing of compilers. In their work, they define the states as all possible substrings of the current input to the program. Actions are then all mutations, that can be applied to those substrings, represented as probabilistic values for each mutation, while the reward is calculated from the unique basic blocks covered by the mutated input.

Paduraru et al. [109] present a framework for reinforcement learning based fuzzing, called RiverFuzzRL, which is able to fuzz binaries using reinforcement learning. They designed RiverFuzzRL in a modular manner, being able to quickly adapt state, action and reward definition, trying to present an open source tool, that can be quickly adapted to the needs of each user, evaluation their idea by using TF-Agents [20] implementations as well as their own implementations of different RL algorithms. This paper will be discussed in detail below.

This summary finally answers Research Question 2, as it shows, that many different approaches to the application of machine learning to the input generation phase of the fuzzing process exist, with many different ideas and concepts being used. The most popular approach seems to be the application of recurrent neural networks, especially LSTMs, being trained before the fuzzing loop to aid with the automatic generation of test cases, while newer research experiments with the application of reinforcement learning to the fuzzing process, especially the application of deep Q-Learning. Those approaches seem promising to also improve fuzzing of embedded devices, as the process can be described in a similar manner when using emulation based fuzzing, which enables for rich instrumentation of the target system, thus creating a good environment for the agent to act on. Following, we will highlight some interesting approaches to the application of machine learning to the fuzzing process of traditional fuzzing in depth, as well as embedded fuzzing, to aid in the final answering of Research Question 3.

[19] https://openai.com/research/openai-gym-beta
[20] https://www.tensorflow.org/agents

2.5.2. Popular augmented traditional Fuzzers

As described in the previous section, many different approaches to the application of machine learning to the fuzzing process exist, which is why we will only highlight a few of the projects, which we deemed most relevant to the topic of this thesis. This includes projects, which built the basis for following research but especially research that highlighted promising applications of reinforcement learning to the domain of fuzzing.

Learn&Fuzz: Machine Learning for Input Fuzzing

Learn&Fuzz by Godefroid et al. [96], as already mentioned above, is an innovative research paper describing a promising application of machine learning to desktop fuzzing, in particular to fuzzing of PDF readers. The central premise of the paper is the application of deep learning techniques to generate models capable of producing correctly formatted inputs to use during fuzzing.

The authors employ a LSTM model, to learn the structure and patterns of valid input files. The training data for this model consists of a dataset of correctly formatted inputs, in the case of the paper PDF files. By training on this data and learning from those examples, the LSTM model is then able to create correctly formatted PDF file components. A key innovation lies in the observation, that the trained model is not perfectly suitable to generate test-cases for fuzzing, as it tends to produce perfect inputs, that don't include unexpected or malformed data. To overcome this problem, Godefroid et al. introduce a custom sampling algorithm, that gets a newly created character from the trained model, as well as the probability of that character being the correct one. If this probability is high enough, they then instead choose to sample another character instead, that has a much lower prediction probability to be the right one, therefore creating unusual or even wrong PDF files. By doing that, unexpected tokens are only introduced in a controlled manner, which is crucial for an sophisticated fuzzing process.

To evaluate their approach, an evaluation metric consisting of code coverage, acceptance rate of the created inputs, as well as the number of bugs found, is introduced. The authors then compare their augmented inputs to the coverage achieved by the non-augmented inputs. During evaluation, they highlight, that the number of epochs used to train the model is crucial, as too few epochs result in a model, that is not able to achieve a very high pass rate, while more than 40 epochs don't result in higher code coverage [96].

Overall, Godefroid et al. introduce a promising approach to the application of machine learning to the fuzzing process, while also highlighting issues, that presented on the way, namely the problem of creating perfect inputs, that don't include malformed data.

Deep Reinforcement Fuzzing

Deep Reinforcement Fuzzing [60] by Böttinger et. al., is another promising paper that explores the application of deep reinforcement learning (DRL) in the realm of desktop fuzzing. The research introduces a concept where DRL techniques are employed to guide and optimize the fuzzing process as an incremental type of model, by utilizing a DRL agent to intelligently select and mutate inputs in a way that maximizes code coverage and the likelihood of discovering software vulnerabilities.

The core methodology involves training a DRL agent using a variant of the Q-learning algorithm [40] introduced above. The agent interacts with a software environment (the program being fuzzed) and learns to select actions (input mutations) based on the observed state of the environment. This state is defined as possible substrings of the current input to the program being tested, while actions represent probabilistic rewrite rules, mapped to those substrings [60]. The reward mechanism is crucial in this setup, as it guides the learning process. The researchers experimented with different reward mechanisms, namely code coverage, execution time, and a combination of both. They found that the combination of both metrics resulted in the best performance, as it allowed the agent to learn to balance the trade-off between code coverage and execution time.

Evaluation was conducted by applying the algorithm to PDF parsers. During evaluation they experimented with different kinds of rewards, while finally concluding that the formulation of the reward function is one of the most important steps to achieve good results. They also found, that the agent was able to learn to balance the trade-off between code coverage and execution time, which is a crucial step to a successful fuzzing process.

By leveraging the capabilities of DRL, fuzz testing can be improved drastically, this baseline could even be used for embedded fuzzing, which will be a basis for chapter 3.

FuzzerGym

Proposed by Drozd et al., FuzzerGym [107] is a framework, that integrates reinforcement learning based on OpenAi Gym[21] with libFuzzer [86] to create a fuzzing environment based on Deep-Double-Q-Learning.

The core methodology of this work is the proposal of a sophisticated cross-language framework, that enables efficient fuzzing augmented with reinforcement learning, while at the same time still maintaining a very fast fuzzing rate. They observe, that libFuzzer is able to sustain an execution speed of more than 100000 executions per second [107], which they claim to not be achievable by just querying a reinforcement learning model for newly generated inputs. To mitigate this time loss by querying the model for each fuzzing loop, they propose an asynchronous approach, where

[21] https://openai.com/research/openai-gym-beta

the model is updated gradually by periodic observations of the system state. To moderate this time disparity, a LSTM layer is employed, that is used to learn time shifted actions, which are updated in a looping manner. The agent is then based on a Deep Double-Q-Network, which uses two Q-tables to prevent overestimation of the Q-values, a common problem in Q-Learning.

The program state is modeled according to the feedback provided by the llvm engine, which is then represented similarly to [98], saving the inputs on a bit-level granularity in contrast to a byte-level one. The authors claim, that while testing both methods, the bit-level granularity achieved better results. To accommodate to the time divergence resulting from the asynchronous approach, actions are implemented in a looping manner. The fuzzing engine executes one mutation action, which is one of the mutations implemented in libFuzzer, until the next observation is made and therefore the RL agent chooses a new mutation strategy to improve the reward. The RL agent then periodically receives batches of input data provided by the fuzzing engine and, based on that in correlation with the coverage feedback, selects a new mutation strategy to improve the reward, which is then employed by the fuzzer until the next decision is made by the agent. Finally the reward is calculated based on the new coverage achieved by the last batch of inputs.

During the evaluation of the proposed architecture, the research team was able to show the improved capabilities of this approach in comparison to the non-augmented libFuzz engine.

RiverFuzzRL

As the first tool, that aims to provide some kind of open-source framework for reinforcement learning based fuzzing, Paduraru et al. present RiverFuzzRL [109]. It is based on the River framework [114], extending it to integrate a modular approach to develop reinforcement learning based fuzzing. River enables dynamic symbolic execution, which is used to gather information about the program state, finally calculating the reward, that the agent receives for a taken action. The authors aim to give potential users as much freedom as possible, highlighting possible configurations in their paper. The state can be defined as freely as possible, to enable different applications, consisting of the last input used on the program, path information in form of basic block sequences, with the possibility to also count the number of occurrences for each basic block, as well as more specialized information like branching behavior using dynamic symbolic execution, hashed basic block addresses and even the possibility for a path embedding of basic block addresses, based on an LSTM [109]. In a similar manner, the actions are designed to be modular as well, while the authors recommend mutation operations similar to [107], implementing the mutations provided by libFuzzer [25] but also providing the possibility to implement custom mutations. Finally, the agents acts on the environment by selecting one of the implemented mutation operations, which is then applied to the current input and send to the target. The agent then observes the executed program under the new input, gathering taint analysis information as well as the information through symbolic execution, which is then fed back to built the next state, as well as used to calculate

the reward. Initially, the authors provide two possible ideas to calculate the reward, the first one only highlighting code coverage, being the newly gained code coverage for the last action, while the second one captures the length of execution paths, increasing the reward for longer paths by calculating the execution time.

The framework is evaluated by showcasing the ease of adaption, reproducing results of similar works, namely [60][107] and [115], focusing on the claimed speed-up their approach brings in comparison to the evaluations of the mentioned works. RiverFuzzRL was able to achieve the maximum lines of code covered 29% faster in comparison to FuzzerGym [107], while not implementing all mutation operations proposed by the latter [109]. Additionally they tested their default implementation and evaluated it according to unique lines of code covered, especially highlighting their policy-gradient based training approach, that is able to find longer paths faster, in contrast to a deep-Q-network based approach.

In conclusion, Paduraru et al. presented a promising approach to open-sourcing reinforcement learning based fuzzing, while also highlighting the importance of the reward function, as well as the state and action definition, which is crucial to the success of the fuzzing process.

2.5.3. Machine Learning Applications to Embedded Fuzzing

Although this is still a very new field, where not much research has been published, a few projects exist, that try to apply ML techniques to different stages during embedded fuzzing. Promising approaches in this field will be highlighted in the following subsections, intending to give an overview of existing ideas in this area to help answer Research Question 3. Unfortunately, as already mentioned above, the corresponding code to the papers is often not available, which makes it hard to reproduce the results, as well as to adapt the ideas to other projects by analyzing the implementation specifics. Additionally it is important to note, that RiverFuzzRL [109] is technically also able to receive AArch32/ARM32 binaries as an input[22], enabling it to also fuzz applications compiled for embedded devices. But as this has not been tested in the evaluation and the tool has already been discussed above, it will not be discussed in this section.

Fw-Fuzz

Gao et al. introduce a cross-platform framework [70], that introduces a genetic algorithm to guide the fuzzing of network applications running on the firmware of embedded devices, while proposing a novel instrumentation mechanism to observe system state and detect faults. A hardware-in-the-loop approach is presented, where test cases are produced on a host system, while the generated inputs are then sent

[22] https://github.com/unibuc-cs/river#how-to-use-the-concolic-sage-like-tool

to the actual device. GDB-Server[23] is utilized on the device to instrument the target program. Breakpoints are set to code locations, deemed interesting, which are then triggered by the execution of the target program and used to analyze the state of the program. This instrumentation then provides code coverage feedback to the genetic algorithm model, that creates the test cases according to a fitness function, consisting of the code coverage, the number of times a path is executed as well as the time it took to find a path. According to this fitness function, the genetic algorithm then selects the best test cases to be mutated, which are then sent to the device to be executed.

The new approach is evaluated by fuzzing different architectures and comparing the results to other fuzzing approaches, namely Boofuzz [116] and Peach [117], as well as evaluation based on found vulnerabilities and performance overhead. Fw-fuzz was able to gain an edge coverage improvement of 33.7% over Boofuzz and 38.4% over Peach [70], while also finding five new zero-day vulnerabilities in the tested applications. To measure the performance overhead of the proposed framework, CPU and memory loads were compared between the fuzzing process and the normal execution of the target program, resulting in an average 5% increase in memory load as well no significant increase in CPU load [70], making the framework a promising approach for embedded fuzzing.

CG-Fuzzer

Yu et al. present CG-Fuzzer [118], introducing a Generative Adversarial Network (GAN) [119] to create input strings for fuzzing industrial IoT protocols.

On a high level, GANs consist of two neural networks, being called a Generator and a Discriminator. The Generator is trained to generate data from random inputs, that seems plausible but is not real, while using the feedback of the Discriminator. The latter is trained on real data using the Generator's output as negative samples during training. Over time, the Generator produces data that is increasingly plausible, so that the Discriminator can't differentiate between real data and generated data.

In the case of CG-Fuzzer, a SeqGAN [120] is used to generate input data for the fuzzing engine. It uses a LSTM model to generate data, while a Convolutional Neural Network (CNN) [121] is used as a Discriminator. The Generator is trained using a reward system, that employs code coverage of the target system on the generated input data, using a policy gradient similar to reinforcement learning. This novel approach uses both the reward, as well as the Discriminator to train the LSTM to create input data, that is then additionally randomly mutated, to be able to achieve high code coverage. Yu et al. evaluated the algorithm on different metrics, namely the rate of test case recognition by the target, the number of exceptions triggered and the diversity of the generated inputs. They show that CG-Fuzzer is able to trigger more exceptions than other tools, namely GANFuzz [122], SeqFuzzer [123] and Peach [117], while at the same time being faster than Peach and having a compa-

[23] https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_130.html

rable execution time to the other tools. Additionally, they show that CG-Fuzzer is able to generate more diverse inputs than the other tools, diversity is defined as the degree of mutation between test cases. Although the diversity is very high, the test case recognition rate and code coverage remains higher than those of the other tools, concluding in a very promising result for CG-Fuzzer.

2.5.4. How will the ML approach understand that it is on the right path to finding vulnerabilities in embedded systems (RQ3)?

The last subsections already indirectly answered this research question, but to finally answer it directly, we have to differentiate between two cases, namely the application of an incremental model, like the reinforcement learning approaches discussed above [106][60][107][108][109][118][124], and the second case being a static model, that is used to automatically generate test cases conforming to the input specifications of a target system, similar to the other discussed papers [98][97][96][99][100][101][102][103]. In the case of the incremental model, the agent is able to learn from the feedback provided by the target system, which is usually code coverage, but can also be execution time or other metrics. This feedback is then used to update the model, which is then able to adapt to changes in the target system, efficiently guiding the fuzzing process. In the case of the static model, the model is trained on a dataset of valid inputs, which is then used to generate new inputs, that are then mutated according to some kind of meta-heuristics to create malformed inputs. In the latter case, the model is not able to adapt to changes in the target system, as it is not trained incrementally, all adaptions to the code coverage or other metrics are handled by other heuristics, whose necessity is reasoned for nicely by Godefroid et al. [96], highlighting the conflict of interest between a perfect model and the production of malformed fuzzing data. Therefore, the static model is not able to understand, that it is on the right path to finding vulnerabilities, it is simply used to generate a large corpus of correctly formed inputs, the modification of such inputs and guidance during the process is handled elsewhere. Nevertheless, such heuristic can also include probing the model for information, like conducted by Godefroid et al. [96] whose sampling procedure involves swapping a character that appears next with high probability with another one the model is confident about not appearing, but usually a feedback loop to the model does not exist.

The approaches used by [98][97][104] and [105] stand somewhat in the middle of the two ideas. While still employing a static model, said model is trained on pairs of inputs and code coverage, enabling it to model the programs behavior and predict, what certain inputs are able to reach or how to modify those inputs, to reach a certain code area.

This shows the need for an incremental model, that can react to defined metrics, to improve the selection or creation of inputs to send to the target. All revised literature implies, that a feedback loop must contain some kind of code or path coverage, to guide the fuzzing process, as this is the metric traditional fuzzing already needs to

effectively test a target, as it is the goal of automated testing, to test as much code as possible. Therefore, a sophisticated feedback loop must be employed, to guide the fuzzing process, which in the case of [107] contains the current input in correlation with gained code coverage, [109] proposes that more advanced observations should be used, including the current input, path coverage, hashed basic block addresses and even the possibility for a path embedding of basic block addresses, based on an LSTM. Böttinger et al. [60] propose to also include a metric capturing execution time in the reward function, also implied to be useful by [70] and [118]. The latter even employ one of the most advanced feedback loops, using code coverage feedback as well as a Discriminator to train the Generator creating inputs, although technically not employing an incremental approach, as the Generator learns to create inputs from scratch, which are randomly mutated.

In conclusion the answer to Research Question 3 is, that the ML approach will understand that it is on the right path to finding vulnerabilities in embedded systems, by using a sophisticated feedback loop, that captures as much information as possible in a reward function, to guide the fuzzing process. The information available will be the individual limitation, as advanced instrumentation must be possible to capture the information, which is not always the case, especially in embedded systems. But as the research shows, the application of machine learning to the fuzzing process is a promising approach, improving the capabilities of automatic testing, especially in embedded systems, as the process can be described in a similar manner as traditional fuzzing when using emulation based techniques, which enable for rich instrumentation of the target system, thus creating a good environment for the agent to act on. Nevertheless, the literature also implies, that implementing the reward function is a crucial step in the effectivity of the model, requiring experimentation and testing to find the best possible solution.

Project	ML Algorithm	Fitness Function
Augmented-AFL [98]	RNN w/LSTM + Seq2Seq	Input - code coverage pairs
Cheng et al. [97]	RNN w/ seq2seq	path coverage - Seed Corpus
Learn&Fuzz [96]	RNN w/ LSTM	(code coverage, pass rate, bugs found)
Fan et al. [99]	Seq2Seq	(code coverage, bugs found, performance)
Paduraru et al. [100]	Seq2Seq	(code coverage, execution time)
Cummins et al. [101]	RNN w/LSTM	(compile time defects, runtime defects, crash rate)
Sablotny et al. [102]	RNN w/LSTM	(code coverage)
Nasrabadi et al. [103]	RNN w/LSTM	(code coverage, bugs found)
NEUZZ [104]	FF-NN Smoothing	Input - code coverage pairs
Angora [105]	Gradient descent	path coverage + constraints
Becker et al. [106]	RL/ SARSA	# functions called + error invocation + delay/corruption of program response
Böttinger et al. [60]	RL/ Deep Q-Learning	code coverage + execution time
FuzzerGym [107]	RL/ Double-Q-Learning	code coverage
Liu et al. [108]	RL/ Deep Q-Learning	code coverage
RiverFuzzRL [109]	RL	code coverage + execution time + dynamic symbolic execution

Table 2.1.: Overview of different machine learning augmentations of input generation for fuzzing

3. Proposed Approach

Based on the literature review in the last sections, this section will now present our proposed approach to augment embedded fuzzing with machine learning. First, we will present the general approach and then explain the steps to take to implement it. Unfortunately, given the time constraint of this thesis, it was not possible to implement the approach, as there were many challenges, which will also be highlighted in this section. Nevertheless, we will present some implementation guidelines and possible frameworks to use, based on the experiences gained and recommendations found in the white and grey literature. This section as a whole will answer Research Question 4

3.1. Fuzzing Augmentation

To highlight, how the observed techniques can be applied to embedded systems fuzzing, we will first present a general approach to augment the embedded fuzzing process, introduced in 2.4, by adapting the fuzzing loop. Consequently we will present a more specific approach based on a chosen technique we deemed promising for embedded systems fuzzing.

3.1.1. General Approach

Figure 3.1 shows a possible augmentation of the original fuzzing loop that was introduced in 2.4, only focusing on the changes to the "Traditional Fuzzing Loop" illustrated in Figure 2.12. Like already highlighted in the previous chapter, the use of an incremental model to guide the input generation process during fuzzing is a promising approach. To enable the integration of such a model, the most important step is to know, which information the chosen fuzzer to be augmented is able to provide during runtime. As already highlighted in 2.5, the feedback loop is an essential component of a successful machine learning model for input generation, therefore enough information to feed back must be available. If the chosen fuzzer is not able to provide such information, or another reason requires it, it might be best to use a static model, that is trained on a dataset that consists of pairs of input and corresponding code coverage, gathered during earlier fuzzing runs. Such a static model would then be able to provide seed inputs to the fuzzing loop, that can then be mutated using another kind of heuristic during each fuzzing loop. However, this approach would not be able to actively learn from the environment and therefore

would not be able to improve over time. Because of that, we will focus on the integration of an incremental model, that is able to learn from the environment and adapt to changes.

The augmented fuzzing process then works as described in the following.

1. **Seed Corpus**

   Starting with a seed corpus, that can either be some random data or a more sophisticated set of inputs, that are known to trigger certain behavior in the target. Such a seed corpus might even be created by another machine learning (ML) model, depending on the use case.

2. **Incremental ML Model**

   The fuzzer then starts the fuzzing loop, feeding the seed inputs to the incremental ML model, which in turn applies mutation strategies on those inputs, based on the experience it has accumulated.

3. **System under Test**

   Those mutated inputs are then fed to the system under test (SUT) over a specific delivery module. This can be an execution of said inputs using an emulator, or a transfer of the inputs to the target device, if a hardware based approach is chosen, but the specifics of this delivery module depend on the use case as well as the chosen fuzzer and method selected.

4. **Observation**

   The Observation component monitors the execution of the input on the SUT, gathering information about the code coverage, crashes and other information, like symbolic execution or static analysis of an extracted system state, if such information can be gathered. This information is then collected by a parsing component, which is responsible for combining the relevant information in a manner that is understandable by the machine learning model. This might include hashing of basic blocks, keeping count of visited paths, execution time, or other approaches of data retention and representation, that enables the ML model to understand the system state and from which the reward can be calculated, which the ML model tries to improve.

5. **Loop**

   The ML model then uses this information to update its internal state and to choose the next input as well as the next mutation strategy to apply. The process then jumps to step 2 again and is then continued repeatedly until a certain stopping criterion is met.

6. **Stopping Criterion**

The stopping criterion can be chosen based on the use case and the available resources, like a certain amount of time has passed, a certain amount of inputs has been generated or a certain amount of code coverage has been reached. When the loop is stopped, the generated inputs are then stored in the corpus and the process can be repeated, starting with the ML model, which is now able to use the new inputs as a starting corpus to improve its internal state and therefore the input generation process.

This represents the basic workflow of an augmented fuzzing process, but as already mentioned, the specifics of each step depend on the use case and the chosen fuzzer. For example, the ML model might be able to learn from the execution time of the SUT, similarly to [96], which might then lead to the preference of inputs or input mutations, that lead to fast crashes of the SUT, instead of inputs that lead to timeouts or that are able to trigger deep code sections. Therefore extensive testing is necessary, as well as a good idea of the use case, to be able to choose the right fuzzer and the right augmentation method.

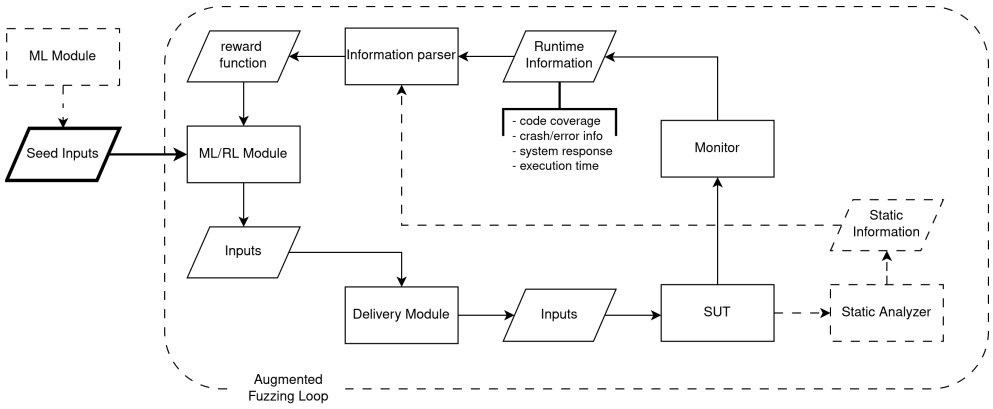

Figure 3.1.: Proposed augmented process of Embedded Fuzzing, only highlighting the loop from 2.12

3.1.2. Example Application of the Proposed Process

The specifics of the augmentation process depend on the use case and the chosen fuzzer, therefore we will present a more specific approach on how to augment the input generation in embedded fuzzing, based on the general approach presented above as a result of our literature research.

Fuzzer Selection

As described in Chapter 2.3.5, many different projects for embedded system fuzzing exist. However, many of them are not actively maintained anymore or are not open

source. Therefore, we decided to use Fuzzware [90] as a basis for our prototype. Fuzzware is a sophisticated extension of AFL and AFL++, that is specialized on fuzzing embedded firmware. It is written in C++ and Python and has a modular architecture, which, makes it accessible for extension with machine learning algorithms written in python. The main reason for choosing Fuzzware is that it is still actively maintained and presents very promising results. Furthermore, it is based on AFL++, which is one of the most popular fuzzers for general fuzzing and through its capability to accept custom mutators[1], AFL++ is the most suitable basis for augmentation with machine learning. The capability of fuzzing monolithic firmware images, with an added possibility to automatically generate configurations for new firmware images, makes it a suitable candidate for our use case, although it is important to note, that currently the automatic generation of firmware configurations is still in a testing phase and automatically generated configurations have to be manually verified before starting longer testing runs[2]. Furthermore, Fuzzware has been evaluated extensively on many different test targets, as well as Zephyr-OS[3] and Contiki-NG[4], which are popular frameworks to build embedded firmware. For each of those targets, the authors of Fuzzware conducted two 24 hour fuzzing runs for their evaluation, enabling an already existing baseline for a potential evaluation of a prototype in the future.

Target Selection

Corresponding to the selection of the fuzzer, the selection of the target in our case is also based on the work of the authors of Fuzzware, but also makes sense when viewing market statistics. As the ARM-Cortex-M architecture is a very popular architecture and is projected to remain important in the future [67], the targets are chosen to be ARM-Cortex-M based. Furthermore, as mentioned above, Fuzzware has already been evaluated on a lot of targets, providing configurations as well as evaluation results, those can be reused as evaluation targets.

Machine Learning Model Selection

To combine the research done in the previous sections, we will focus on the use of reinforcement learning to augment the input generation of our fuzzing process. As already discussed in Chapter 2.2.1 and 2.5, reinforcement learning is a suitable approach for our problem, since it is able to learn from the environment as well as the actions taken and doesn't necessarily rely on a huge dataset to be trained beforehand. Furthermore, it is able to learn from delayed rewards, which might be necessary in the case in fuzzing, since the reward is only given after the execution

[1] https://aflplus.plus/docs/custom_mutators/
[2] https://github.com/fuzzware-fuzzer/fuzzware/blob/main/docs/target_configuration.md
[3] https://zephyrproject.org/
[4] https://github.com/contiki-ng/contiki-ng

of the test case and if an approach similar to FuzzerGym [107] is chosen, those rewards will be delayed. As shown in 2.5, a few works already applied Reinforcement Learning to fuzzing [60][106][109], but only [118] focused specifically on embedded systems, although *RiverFuzzRL* [109] also works with ARM binaries but has not been tested on them. The findings of those works will build the basis for our approach. Similar to [60] we propose a Deep-Q-Learning method, since it has already been proven to achieve good results in traditional fuzzing, but this can also be adapted to a model-based method for further testing, like Paduraru et al. [109] and Yu et al. [118] propose. Adhering to the approaches of [109] the problem then consists of an environment encapsulating the actions and observations, on which the agent acts. How this environment gets designed, will be discussed in the next paragraph.

Proposed Augmentation

To define the environment on which our agent acts, we adhere to findings of similar work, mainly [109][107][118].

1. **Seed Corpus**

   Fuzzwares fuzzing loop starts with a seed corpus of three inputs, each one being 512 bytes long. Those three inputs are set to be all zero bits, all one bits and the third one being a concatenation of 32-bit values, each shifting 1 bit. As found in [107] and [98], viewing input on a bit-granular level seems to perform better, therefore we define the input as an "array-of-bits" [107] in the following manner:

   $$Input : [0, 255]^{512} \mapsto [0, 1]^{4096} \tag{3.1}$$

2. **Incremental ML Model**

   As an incremental ML model, we propose a reinforcement learning agent, that acts on the environment by taking actions, that modify the input and then execute it on the system under test. The actions an agent can take then represent operations similar to those in [107] and therefore [25], including adding bits or bytes at given locations, changing bits or bytes, erasing them or even adding words from a token dictionary, as well as a byte shuffle operation. All proposed mutation strategies can be found in 3.1 and can of course be adapted or extended during evaluation of a future implementation.

3. **System under Test**

   The system under test (SUT) is the firmware image, that is loaded into the Fuzzware emulator, which is then executed using the input generated by the agent. During runtime, Fuzzware is able to provide the following information

regarding the execution of an input, which can be used as observations for the agent:

- The last input that was executed
- New basic blocks that were covered during the execution of the last input
- Trace of the last input as a list of triggered basic block addresses through dynamic symbolic execution using angr [125][126]
- The number of crashes induced in the firmware
- The time it took to execute the last input

4. **Observation**

   Using that information, we propose an observation space, that captures the last input, new basic block coverage as the number of newly covered basic blocks, a list containing the path the last input took as basic block addresses and the number of occurrences for each basic block, as well as the number of new crashes found for each input. This information is then used to represent the state, the system under test is in after the execution of the last input.
   The reward is then calculated as the number of newly covered basic blocks, as well as the number of new crashes found, which is then used to update the agents internal state and observation space.

5. **Loop**

   The agent acts on the environment by invoking a *step()* function, which includes the action on the current input, as well as executing the input using the Fuzzware emulator, waiting for the execution to finish and then gathering the information described above.
   The agent then uses this information to update its internal state and to choose the next input as well as the next mutation strategy to apply. Inputs, that increase code coverage, will be added to the corpus of promising inputs for further mutation.

   As Q-Learning was able to achieve good results in the past, it should also be used for this approach, therefore the agent updates the Q-values after each step. Nevertheless, by defining the environment in such a way, this approach should be possible to use with other RL algorithms as well, which should also be part of a testing process after implementation.

6. **Stopping Criterion**

   The described process is repeated until a certain stopping criterion is met, like a certain amount of time has passed, a certain amount of inputs have been generated or a certain amount of code coverage has been reached. The stopping

criterion can be chosen based on the use case and the available resources, but the passed time is the main criterion used by Fuzzware.

The described specific approach is visualized in Figure 3.2.

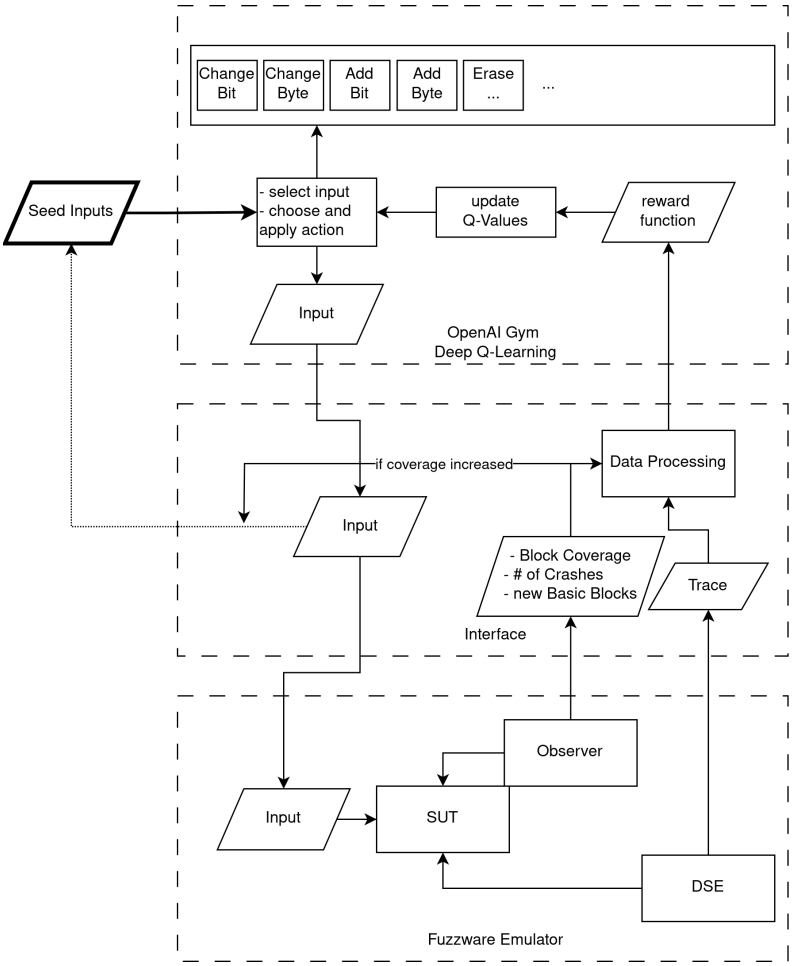

Figure 3.2.: Proposed specific augmented process of Embedded Fuzzing, only highlighting the loop from 2.12

As analyzed in [107], this synchronous approach of course slows down the fuzzing process, since the agent has to wait for each execution to finish. Therefore this approach builds the baseline for further testing approaches, including a sophisticated introduction of delayed rewards in combination with a LSTM-based model, as proposed in [107]. This seems to also be applicable to Fuzzware, but is a lot more complex to implement, therefore initial testing should be done by applying the proposal above. To also highlight possible applications of [107], we propose to use the same approach to augment the input generation process, but instead of using a LSTM-based model, we propose to try a Transformer-based model [52], as Transformers

Mutator	Description
ChangeBit	Changes a single bit at a random or given position
ChangeByte	Changes a single byte at a random or given position
AddBit	Adds a single bit at a random or given position
AddByte	Adds a single byte at a random or given position
EraseBit	Erases a single bit at a random or given position
EraseByte	Erases a single byte at a random or given position
AddToken	Adds a random token from a dictionary at a random or given position
ShuffleBytes	Shuffles the bytes of the input
AddRepeatedBytes	Adds a random number of repeated bytes at a random or given position
CrossOver	Replaces a part of an input with parts from another one, that already performed well

Table 3.1.: Overview of different proposed mutation strategies

perform astonishingly well for problems in the domain of natural language processing [127][128][129]. In this case, the agent only receives the system state periodically, choosing a new mutation strategy for the fuzzer to apply until the next observation is made. The agent then updates the weights of a Transformer-based neural network, that gets the last fuzzing input as an input and outputs a new mutation strategy, that will likely increase the reward. The system observation state is fed to a Double-Q-Learning-based RL agent, that adjusts the weights of the neural network to maximize the expected reward.

This section presented a general approach to augment the fuzzing process, as well as a more specific approach, based on the findings of our literature review, therefore answering Research Question 4, exploring how the input generation phase of the embedded fuzzing process can be improved using machine learning. The next section will present some implementation guidelines and possible frameworks to use, based on the experiences gained and recommendations found in the white and grey literature.

3.2. Implementation Guidelines

As already mentioned, during this work, it was not possible to create a prototype of the proposed approach, as the implementation presented many challenges. Nevertheless, we will present some implementation guidelines and possible frameworks to use, based on the experiences gained while trying to implement said prototype, as well as recommendations found in the white and grey literature. Additionally, challenges that arose during implementation, which are the main reason the prototype could not be implemented in this work, will be highlighted in the next section.

3.2.1. Selection of Tools

During our research, we found many different tools, used for implementation of similar works. In this section, we will present possible tools and frameworks to use, based on our research and the recommendations found in the white and grey literature.

Framework Selection

For reinforcement learning, many different sophisticated and widely adapted python frameworks exist [130][131][132][133], but because we need the possibility to easily adapt a custom environment for the agent to act on, OpenAi Gym[5] or its successor Gymnasium[6], is a suitable framework to use. Gymnasium is a toolkit used for the development and comparison of reinforcement learning algorithms. It provides a simple interface to a variety of environments, which are already implemented and can be used out of the box but most importantly it enables the fast creation of custom environments for an agent to act on, which is necessary for our use case.

The environment can be created and customized in a modular manner, by defining the observation space, the action space and the step function, as well as other utility functions, like resetting the environment. The step function is called by the agent to act on the environment, finally returning the necessary information to calculate the reward and update the internal state. The observation space can be defined as a tuple of spaces, which can be a discrete space, a box space or a multi binary space, similar to the action space[7]. Additionally some wrappers to normalize reward functions exist, which can also be utilized. This enables the fast creation of custom environments, that can be used by the agent to act on.

For our approach, the observation space can be described as a dictionary space[8], consisting of Box or Tuple spaces, that represent the necessary information mentioned

[5] https://openai.com/research/openai-gym-beta
[6] https://gymnasium.farama.org/index.html
[7] https://gymnasium.farama.org/api/spaces/fundamental/
[8] https://gymnasium.farama.org/api/spaces/composite/#dict

above. Box spaces are defined as n-dimensional continuous spaces in $\mathbb{R}^n$, so they are useful for representing the path information in form of basic block hashes, as well as mapping basic block addresses to the number of occurrences. The action space can be defined as a discrete space, consisting of the index of different mutation strategies, or even a Tuple space, that consists of a discrete space for the index of the mutation strategy and a Box space for the position of the mutation. The step function finally executes the chosen action on the input and runs the Fuzzware emulator with the given input, utilizing the interface module seen in 3.2 to update the observation space and return it together with the reward. The initial reward for testing should be calculated by the increase in code coverage, the last input gained, but further testing should be done by also incorporating the number of new crashes found, as well as the execution time of the input.

Input representation

As already mentioned above, the input should be represented as an "array-of-bits" [107], which can be done using the BitVector library[9]. This representation can then be used to apply the mutation strategies, as the API provides methods to use bitwise operations on the data. If this proves to be slow or unsuitable during this approach, the input can also be represented as a plain list, mapping bit indices to the respective value, or finally in byte representation, either as a bytearray or as a dictionary, mapping byte indices to the respective value. Generally it seems to be useful, to define the data structure as a class, providing methods to apply the mutation strategies, as well as methods to convert the data to different representations, if necessary. This enables the fast creation of different representations, that can be used for evaluation.
To set inputs in context to one another, a tree like structure might be useful, linking mutated inputs to the "parent" they were mutated from, as well as the mutation strategy that was applied. This enables the fast creation of a corpus of promising inputs, that can be used for further mutation, as well as the possibility to trace back the mutation strategies that were applied to a certain input. This might also be useful for evaluation purposes, to analyze the mutation strategies that were applied to inputs that lead to a high reward.

Action Implementation

Depending on the chosen input representation, actions should be implemented accordingly. A possible solution might accept a list of parameters as an argument, containing the input to change, the index, where the mutation should be applied as well as additional information, like the value to add or the value to change to. A possible simple implementation of such action functions can be seen in Listing 1. Those examples receive an array as an input, that maps bit indices to the respective values, as well as the index of the bit to change for the `change_bit()` function and the

[9] https://pypi.org/project/BitVector/

```python
from numpy import random

class Actions:
    @staticmethod
    def bit_change(input, index=None):
        if index is None:
            index = random.randint(0, len(input) - 1)
        input[index] = 1 - input[index]
        return input

    @staticmethod
    def byte_change(input, index=None, value=None):
        if index is None:
            index = random.randint(0, len(input) - 1)
        if value is None:
            value = random.randint(0, 255)
        bin_value = bin(value)[2:].zfill(8)
        for i in range(8):
            try:
                input[index + i] = int(bin_value[i])
            except IndexError:
                pass
        return input
```

Listing 1: Simple examples of action functions receiving an array as an input, mapping bit indices to values

starting index of the byte as well as the value to change to for the `change_byte()` function. If those parameters are not given, a random location and a random value is chosen respectively. Those are of course only simple examples, but can be used as a starting point for further implementation.

Machine Learning Model Implementation

The literature review in Chapter 2.2.1 and 2.5 showed, that many different approaches to implement reinforcement learning exist. As already mentioned, we propose to use a Deep-Q-Learning approach as a starting point, as it has already been proven to achieve good results in traditional fuzzing. Which framework to use, is up to the user, but a good approach would be to use tensorflow agents[10], as this library provides different reinforcement learning algorithms to experiment with. Using that framework, the `tf_agents.agents.dqn.dqn_agent` can be used together with `tf_agents.networks.sequential` to implement a Deep-Q-Learning approach. The network can then be implemented using fully connected keras dense layers (`tf.keras.layers.Dense()`) with a final dense layer with a

[10] https://www.tensorflow.org/agents

number of units adhering to the number of actions, to output q-values for each action. The policy then has to be created to account for all the information presented in this chapter, evaluating it against a random policy, that chooses action at random. Additionally, to evaluate such an implementation, the average return should be computed and gathered, as well as the fuzzing specific metrics. To store experiences a replay buffer must be used, the official `tf_agents` documentation suggests using the `tf_agents.replay_buffers.reverb_replay_buffer`. If everything is set up, the training of the agent can begin, which already introduces the fuzzing process, due to the environment definition. The agent will learn to produce better input over time, the network weights can then be exported and reused for other fuzzing runs.

Fuzzer Interface

The environment has to be tightly coupled to work with the selected fuzzer, in our case Fuzzware. Therefore, the environment has to be able to invoke the fuzzer, as well as to gather the necessary information to update the internal state of the agent. In the case of Fuzzware, this can be done by modifying the Fuzzware Pipeline to also include the complete logic for our Deep-Q-Learning approach, which requires to rewrite a lot of the project. Another possibility would be to use custom mutators for AFL++[11], which would enable the use of the already existing Fuzzware pipeline, building a custom mutator that invokes the environment and returns the mutated input. This would require to implement the environment as a python module, which can then be invoked by the custom mutator, but this approach seems to bring some challenges. Fuzzware uses unicorn as a basis for emulation, invoked by a custom harness in combination with AFL++. To get inputs to unicorn, the forkserver mechanism[12] is utilized, using shared memory for data transfer and communication. Therefore, to implement a custom mutator, this mechanism has to be changed, which requires in depth knowledge and understanding of the current implementation.

To combine those approaches, it might be necessary to split the machine learning logic from Fuzzware, to ensure continuity and no interrupts during training, as the forkserver method of AFL++ seems to only work on a turn to turn basis, sleeping in between, which would make it impossible for an agent to be trained. Therefore some inter-process communication seems to be necessary to enable the agent to act on the environment, which might be possible using gRPC[13] as a cross-platform data streaming solution, which would then lead to a custom mutator requesting new input from the RL module. Independent on the chosen approach, the environment has to be able to invoke the emulation and get back all necessary information described above. The implementation of such an interface takes a lot of time and effort, consisting of a thorough code-review to understand the existing project, as well as the implementation of the interface itself, which might be the most challenging part of the implementation process.

[11] https://aflplus.plus/docs/custom_mutators/
[12] cf. [134] and [135]
[13] https://grpc.io/

AMD ROCm

A challenge that might occur depending on the used hardware, is the installation of the AMD ROCm[14] framework, which is necessary to use an AMD GPU for training the machine learning model. The installation of the framework is not trivial and might require some time and effort, depending on the used hardware and the operating system. Therefore, it might be useful to use a docker container[15] to install the framework, as this might be easier to set up and maintain. Nevertheless, we highlight some installation steps to install ROCm in Appendix A.

3.3. Challenges and Lessons Learned during Implementation

As already mentioned, during the attempt to implement a prototype of the proposed approach, many challenges occurred, which are the main reason, why the prototype could not be implemented during this work. In this section, we will highlight those challenges, to give an overview of the complexity of the implementation process.

3.3.1. Old Projects

As already highlighted in Chapter 2.3.5, there are countless different projects for embedded system fuzzing, but many of them are not actively maintained anymore.

Fuzzer Selection

Therefore, although many promising approaches have been published, the fuzzer selection is a key element in the process of embedded fuzzing, as it defines the amount of manual labour that has to go into the creation of a suitable fuzzing environment. Different works like Firm-AFL [82] present promising evaluation results, but as the last contribution to this specific project is five years old, many dependency requirements can no longer be met, necessitating a lot of manual patching and code review, to get the project to work as intended. As a matter of fact, the first project we chose to use as a baseline, was destined to be Firm-AFL, but even after a lot work, manually applying patches to dependencies, to apply backwards compatibility, we were not able to make the project work in an appropriate time span, therefore we decided to go for another project.

[14] https://rocm.docs.amd.com/en/latest/
[15] https://hub.docker.com/r/rocm/tensorflow

Possible Solution

This highlights the importance of choosing a suitable fuzzer, as it defines the amount of work that has to go into the implementation of the proposed approach. Therefore, such a project should always be chosen to be either a proposal, that was recently published, or a large open-source project, that is still maintained well and has had recent contributions.
Viewing the existing literature also seems to confirm this as an existing problem, as especially works applying machine learning to the fuzzing process arise from incremental research based on own previous work [109] or even building a completely new fuzzer around their approach [118][60][108], mitigating the dependance on maintenance of open-source projects.

3.3.2. Large Existing Projects

Another observed challenge included the size and complexity of existing projects, that try to present sophisticated approaches for fuzzing embedded devices. This was the main challenge, that finally prevented the implementation of a prototype to evaluate the approach proposed in this work, as more time would have been necessary to understand the existing code base, add functionalities and and especially conduct a sophisticated evaluation, testing different methods and experimenting with approaches. Nevertheless, many insights have been gathered, that can be applied in future work.

Problem Domain Complexity

Especially for embedded fuzzing, many different components have to be considered, as the fuzzing process is not only limited to the execution of the input on the target, but also the whole emulation harness, the communication between components, control logic and many more, leading to astonishingly big projects, increasingly hard to understand.
As highlighted in this thesis, the fuzzing of embedded devices is a highly specialized field, necessitating many different steps an prerequisites to enable a sophisticated fuzzing process. Due to the vast variety of embedded systems, in addition to the fuzzing logic, often a large corpus of different configuration files has to be supplied and maintained, to enable the targeting of different devices and architectures.

The code base for Fuzzware for example, has around 500000 lines of code using the languages C/C++, including the header files, and Python[16]. While certainly not every line of code is important, this gives an overview of the complexity of such projects. This leads to the involvement of a plethora of different components, that have to be adapted to change the functionality of such fuzzers, increasing the complexity of augmentations.

[16] counted with cloc (`https://github.com/AlDanial/cloc`) and find in combination with wc -l

Unintended Adaption

While the usage of such projects is often well documented, the adaption to other use cases or the extension to fundamentally new features was often not intended or foreseen, therefore also not well documented, if such use cases are documented at all.
This makes it challenging to adapt such projects to new use cases, as a lot of code review is necessary to understand the existing code base, as well as to differentiate between new or changed code, that changes the functionality of already existing components, and unmodified parts of segments. As an example, Fuzzware uses AFL, AFL++ and Unicorn to offer its functionalities, but they changed some baseline functionality of those, making it difficult to find correct and working approaches to augment this existing project with machine learning approaches.

Possible Solution

Similar to the previous challenge 3.3.1, the complexity and size of existing projects can be mitigated by enabling incremental research, exploring the possibilities of machine learning in the fuzzing process in multiple steps, building on previous work, to lay a foundation to learn from. Furthermore, it might still be useful to build a new fuzzer around the proposed approach, or at least fork an existing project to change deep laying functions, as this enables for deep knowledge of functionalities, as well as the possibility to build a modular architecture, built especially to accommodate machine learning augmentations of the fuzzing process. This of course requires a lot of work, but might mitigate suboptimal solutions, that arise from the adaption of existing projects, that were not built with machine learning in mind.

3.3.3. Lack of Open-Source Projects

Especially in the realm of machine learning application to the fuzzing process, regardless of traditional or embedded fuzzing, researchers tend to not publish their code, that was created to implement prototypes to apply their ideas.

Existing Projects

As an example, we could only find the published code for two projects, that proposed promising applications of machine learning techniques to the fuzzing process, namely NEUZZ [104] and RiverFuzzRL [109], the latter being published because of the same reasoning we had, as the authors were not able to find a suitable open-source project to build their approach on. This is a problem, as it makes it difficult to build on existing work, as well as to compare different approaches, as the code is not

available. This is especially problematic, as the implementation of such approaches is often not trivial, as already highlighted in the previous sections.

Possible Solution

To mitigate this challenge, we strongly recommend to publish code, that was created to implement prototypes, if a working prototype can be contributed, as this enables other researchers to build on existing work and extend ideas, as well as to compare different approaches, which is a key element of scientific research, instead of having to build their own tools through incremental research over a span of multiple years. This may also include the mitigation of the previous discussed challenges, as it provides the opportunity to build a sophisticated fuzzing project with the possibility to extend it with machine learning approaches in mind, enabling the fast adaption and extension to accommodate new ideas and approaches.

4. Summary and Future Work

This section will summarize the insights gathered and contributions done by this thesis and will give an outlook on future work to further explore and improve the field of embedded fuzzing. Additionally the limitations and challenges of this thesis will be highlighted.

4.1. Summary and Contributions

By conducting a thorough review of white and grey literature on the topic of traditional as well as embedded fuzzing, this thesis has provided a comprehensive overview of the current state of the art in this field. The insights gathered on traditional fuzzing have then been compared to the fuzzing of embedded devices, highlighting differences of the approach to conduct the latter as well as challenges and resulting differences in techniques applied to this field, especially highlighting different emulation approaches and the importance of correct state representation. This concluded in a comprehensive overview of steps necessary to conduct automated testing on embedded devices, which can be used as a framework to conduct this technique.

Furthermore, by analyzing the current state of the art of machine learning applications to traditional fuzzing, this thesis has provided a thorough overview of ideas and techniques to improve the input generation phase of the fuzzing process, finding possible solutions to adapt those techniques to embedded fuzzing as well. This was done by additionally analyzing existing applications of machine learning algorithms to the realm of embedded fuzzing, finding methods to guide the fuzzing process as well as to improve the input generation phase, especially highlighting the importance of a sophisticated feedback loop as well as a correct definition of a fitness function.

Finally, using the insights gathered by a comprehensive literature review and analysis of existing tools, we proposed a possible augmentation of the fuzzing process, to improve the input generation during embedded fuzzing. This was done by proposing a general approach as well as a specific use case, how the project Fuzzware might be augmented with an incremental machine learning model, namely reinforcement learning, to improve input generation and therefore optimize the automated testing process. Additionally we provided implementation possibilities based on existing research and tools, highlighting challenges that occur during the implementation

process, especially the time and knowledge overhead necessary to augment an existing tool with machine learning capabilities.

4.2. Limitations and Challenges

As the nature of this thesis presented the work with a time constraint, all work provided remains theoretical and no thorough implementation was possible. While there exist approaches using similar techniques and frameworks, none of them has been applied to the chosen fuzzer and open-source implementations for the respective application to embedded fuzzing do not exist. During the analysis of possible implementation steps, it became increasingly clear, that the implementation of a reinforcement learning model to augment the fuzzing process is a considerably complex task, requiring a lot of time and deep knowledge in different areas, possibly requiring a team of specialists to work together. In addition to the implementation overhead, testing and optimizing of said augmentation again requires a lot of time and resources, to conduct a thorough evaluation of proposed approaches on real-world targets. Therefore, the implementation of the proposed augmentation remains a challenge for future work.

Another limitation important to note is the fact, that all proposed ideas are based on an emulation of the hardware. While this is a common approach, and especially using the MMIO fuzzing proposed by Fuzzware should be representative of the real device, it remains an emulation. Therefore, the results gathered by fuzzing an emulated device might not be perfect representative of every state the real hardware might be in, especially if the emulation is not perfect. Thus, the results found by the proposed augmentation might not be exploits, that work on the real-world system, consequently it might not find all possible bugs and vulnerabilities present in the actual device.

Nevertheless this work provides a comprehensive introduction and overview to the realm of traditional and embedded fuzzing, providing ideas and approaches to improve the fuzzing process by augmenting it with machine learning techniques. Therefore, this thesis provides a solid foundation for future work in this field.

4.3. Future Work

As the implementation of the proposed augmentation is important to evaluate ideas and finally improve existing fuzzing projects, this remains a challenge for future work. Additionally, the proposed augmentation is not limited to the fuzzing project Fuzzware, but can be applied to other fuzzers as well, as long as they provide a suitable feedback information. Therefore, the proposed augmentation should be applied to other fuzzers as well, which might even be a more suitable starting point for the implementation and testing, especially if the projects are faster to understand and

not as big as Fuzzware. A suitable alternative might certainly be FirmAE [81] or even plain AFL++ in unicorn mode[1] augmented with a custom harness and mutator.

Furthermore, the proposed approach is not limited to Q-Learning or even plain reinforcement learning, as shown by [118], therefore different approaches should be evaluated as well, building incremental models in different ways. Of course the applications of machine learning are not limited to the input generation phase, but can be applied to other parts of the fuzzing process as well, namely the re-hosting process like conducted in [88], initial seed file generation or exploitation analysis of inputs found during the fuzzing process. Therefore, it might be interesting to evaluate different approaches to augment the fuzzing process with machine learning techniques.

Finally, to speed up the research in this area, it might be necessary and useful to provide an open-source framework, implementing different ideas and approaches, to enable other researchers to build upon existing work and further improve the fuzzing process without having to start over fresh again or incrementally build their own framework.

[1] https://aflplus.plus/docs/fuzzing_binary-only_targets/

A. Appendix 1 - ROCm Installation

As the installation of ROCm might present some problems, especially for systems not based on Ubuntu, we will describe the process for an Arch Linux based system. For Ubuntu based distributions, an official installation guide exists[1].

To initially install ROCm, the respective packages have to be installed:

```
$ sudo pacman -Syyu
$ sudo pacman -S rocm-hip-sdk rocm-opencl-sdk
```

After that, following the official installation instructions[2], the shared objects have to be added to the library path:

```
$ sudo tee --append /etc/ld.so.conf.d/rocm.conf <<EOF
$ /opt/rocm/lib
$ /opt/rocm/lib64
$ EOF
$ sudo ldconfig
```

The current user has to be added to the *video* and *render* group to be able to access the GPU:

```
$ sudo usermod -a -G video $USER
$ sudo usermod -a -G render $USER
```

Additionally the rocm binary has to be added to the `PATH`:

```
$ export PATH=$PATH:/opt/rocm/bin:/opt/rocm/opencl/bin
```

Finally, the following commands have to be executed before using ROCm (the environment variables can also be exported via the `~/.bashrc` file):

```
$ export HSA_OVERRIDE_GFX_VERSION=10.3.0
$ export ROCM_PATH=/opt/rocm
$ for a in /sys/bus/pci/devices/*; do echo 0 \
    | sudo tee -a $a/numa_node; done
```

[1] https://rocm.docs.amd.com/projects/install-on-linux/en/latest/how-to/native-install/ubuntu.html

[2] https://rocm.docs.amd.com/projects/install-on-linux/en/latest/how-to/native-install/post-install.html

Bibliography

[1] Deepak Kumar et al. "All Things Considered: An Analysis of IoT Devices on Home Networks". In: 28th USENIX Security Symposium (USENIX Security 19). 2019, pp. 1169–1185. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-deepak (visited on 11/10/2023).

[2] JSOF Tech. *Ripple20*. JSOF. 2020. URL: https://www.jsof-tech.com/disclosures/ripple20/ (visited on 11/10/2023).

[3] Armis. *URGENT/11*. Armis. 2019. URL: https://www.armis.com/research/urgent-11/ (visited on 11/10/2023).

[4] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Communications of the ACM* 63.6 (May 21, 2020), pp. 46–56. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3357033.

[5] Paul Kocher et al. *Spectre Attacks: Exploiting Speculative Execution*. Jan. 3, 2018. DOI: 10.48550/arXiv.1801.01203. arXiv: 1801.01203 [cs]. preprint.

[6] Internationale Elektrotechnische Kommission, ed. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems - Part 1: General Requirements*. Ed. 2.0,2010-04. International Standard / IEC 62443-1-1. Geneva: IEC Central Office, 2010. 127 pp. ISBN: 978-2-88910-524-3.

[7] Internationale Elektrotechnische Kommission, ed. *Industrial Communication Networks: Network and System Security. Pt. 1,1: Terminology, Concepts and Models*. Ed. 1.0, 2009-07. International Standard / IEC 62443-1-1. Geneva: IEC Central Office, 2009. 81 pp. ISBN: 978-2-88910-710-0.

[8] International Organization for Standardization. *ISO/IEC 27001:2013 Information Technology - Security Techniques - Information Security Management Systems - Requirements*. 2013. URL: https://www.iso.org/standard/54534.html (visited on 11/10/2023).

[9] International Organization for Standardization. *ISO/IEC/IEEE 12207:2017 Systems and Software Engineering - Software Life Cycle Processes*. ISO. 2021. URL: https://www.iso.org/standard/63712.html (visited on 01/11/2024).

[10] International Organization for Standardization. *ISO 22301:2019 Security and Resilience - Business Continuity Management Systems - Requirements*. ISO. 2019. URL: https://www.iso.org/standard/75106.html (visited on 01/11/2024).

[11] International Organization for Standardization. *ISO/IEC/IEEE 29119-1:2022 Software and Systems Engineering - Software Testing - Part 1: General Concepts*. ISO. 2022. URL: https://www.iso.org/standard/81291.html (visited on 01/11/2024).

[12] International Organization for Standardization. *ISO 26262-1:2018 Road Vehicles - Functional Safety - Part 1: Vocabulary*. ISO. 2018. URL: https://www.iso.org/standard/68383.html (visited on 01/11/2024).

[13] International Organization for Standardization. *ISO/SAE 21434:2021 Road Vehicles - Cybersecurity Engineering*. ISO. 2021. URL: https://www.iso.org/standard/70918.html (visited on 01/11/2024).

[14] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Communications of the ACM* 33.12 (Dec. 1, 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[15] Marcel Böhme. *STADS: Software Testing as Species Discovery*. Apr. 3, 2018. arXiv: 1803.02130 [cs]. URL: http://arxiv.org/abs/1803.02130 (visited on 01/14/2024). preprint.

[16] Max Eisele et al. "Embedded Fuzzing: A Review of Challenges, Tools, and Solutions". In: *Cybersecurity* 5.1 (2022), p. 18. DOI: 10.1186/s42400-022-00123-y.

[17] Patrice Godefroid. "Random Testing for Security: Blackbox vs. Whitebox Fuzzing". In: *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. RT '07. New York, NY, USA: Association for Computing Machinery, Nov. 6, 2007, p. 1. ISBN: 978-1-59593-881-7. DOI: 10.1145/1292414.1292416.

[18] Patrice Godefroid, Michael Y Levin, and David Molnar. "Automated Whitebox Fuzz Testing". In: *NSDD* 8 (2008), pp. 151–166. URL: https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf.

[19] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft." In: *Queue* 10.1 (Jan. 11, 2012), pp. 20–27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081.

[20] Vijay Ganesh, Tim Leek, and Martin Rinard. "Taint-Based Directed Whitebox Fuzzing". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009 IEEE 31st International Conference on Software Engineering. May 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[21] Marcel Böhme et al. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas Texas USA: ACM, Oct. 30, 2017, pp. 2329–2344. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134020.

[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 24, 2016, pp. 1032–1043. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978428.

[23] Istvan Haller et al. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations". In: 22nd USENIX Security Symposium (USENIX Security 13). 2013, pp. 49–64. ISBN: 978-1-931971-03-4. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller (visited on 11/15/2023).

[24] Yuyue Zhao et al. "Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning". In: *Information Security and Cryptology*. Ed. by Zhe Liu and Moti Yung. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 134–153. ISBN: 978-3-030-42921-8. DOI: 10.1007/978-3-030-42921-8_8.

[25] *libFuzzer – a Library for Coverage-Guided Fuzz Testing. — LLVM 18.0.0git Documentation*. URL: https://llvm.org/docs/LibFuzzer.html (visited on 01/14/2024).

[26] Jiongyi Chen et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing". In: *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2018. ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23159.

[27] Nilo Redini et al. "Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). May 2021, pp. 484–500. DOI: 10.1109/SP40001.2021.00066.

[28] Michal Zalewski. *Google/AFL*. Google, 2016. URL: https://github.com/google/AFL (visited on 01/14/2024).

[29] Lcamtuf. *American Fuzzy Lop*. 2017. URL: https://lcamtuf.coredump.cx/afl/ (visited on 01/14/2024).

[30] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. WOOT'20. USA: USENIX Association, Aug. 11, 2020, p. 10.

[31] Marc Heuse et al. *AFL++*. Version 4.00c. Jan. 2022. URL: https://github.com/AFLplusplus/AFLplusplus (visited on 12/21/2023).

[32] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing". In: *Proceedings 2017 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2017. ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23404.

[33] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). San Jose, CA: IEEE, May 2016, pp. 110–121. ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.15.

[34] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017 IEEE Symposium on Security and Privacy (SP). May 2017, pp. 579–594. DOI: 10.1109/SP.2017.23.

[35] Stuart Russell and Peter Norvig. *Artificial Intelligence, Global Edition : A Modern Approach*. Pearson Deutschland, May 13, 2021. 1168 pp. ISBN: ISBN 9781292401133. URL: https://elibrary.pearson.de/book/99.150005/9781292401171.

[36] *AI vs. Machine Learning: How Do They Differ?* Google Cloud. URL: https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning (visited on 11/20/2023).

[37] Pariwat Ongsulee. "Artificial Intelligence, Machine Learning and Deep Learning". In: *2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE)*. 2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE). Nov. 2017, pp. 1–6. DOI: 10.1109/ICTKE.2017.8259629.

[38] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research* 4 (May 1, 1996), pp. 237–285. ISSN: 1076-9757. DOI: 10.1613/jair.301.

[39] Yuxi Li. *Deep Reinforcement Learning: An Overview*. Nov. 25, 2018. arXiv: 1701.07274 [cs]. URL: http://arxiv.org/abs/1701.07274 (visited on 11/20/2023). preprint.

[40] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. arXiv.org. Dec. 19, 2013. URL: https://arxiv.org/abs/1312.5602v1 (visited on 09/25/2023).

[41] Matthew Hausknecht and Peter Stone. *Deep Recurrent Q-Learning for Partially Observable MDPs*. arXiv.org. July 23, 2015. URL: https://arxiv.org/abs/1507.06527v4 (visited on 01/12/2024).

[42] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. Dec. 8, 2015. DOI: 10.48550/arXiv.1509.06461, arXiv: 1509.06461 [cs], preprint.

[43] *Machine Learning, Explained — MIT Sloan*. Jan. 11, 2024. URL: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained (visited on 01/12/2024).

[44] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (July 1959), pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210.

[45] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers. II—Recent Progress". In: *IBM Journal of Research and Development* 11.6 (Nov. 1967), pp. 601–617. ISSN: 0018-8646. DOI: 10.1147/rd.116.0601.

[46] *What Are Recurrent Neural Networks? — Data Basecamp*. Dec. 14, 2021. URL: https://databasecamp.de/en/ml/recurrent-neural-network (visited on 01/28/2024).

[47] David Rummelhart and James L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. 1987, pp. 318–362. URL: https://ieeexplore.ieee.org/document/6302929.

[48] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network". In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 01672789. DOI: 10.1016/j.physd.2019.132306. arXiv: 1808.03314 [cs, stat].

[49] A. Graves et al. "A Novel Connectionist System for Unconstrained Handwriting Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.5 (May 2009), pp. 855–868. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2008.137.

[50] G. E. Dahl et al. "Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition". In: *IEEE Transactions on Audio, Speech, and Language Processing* 20.1 (Jan. 2012), pp. 30–42. ISSN: 1558-7916, 1558-7924. DOI: 10.1109/TASL.2011.2134090.

[51] Milos Miljanovic. "Comparative Analysis of Recurrent and Finite Impulse Response Neural Networks in Time Series Prediction". In: 3.1 (2012).

[52] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 1, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs]. preprint.

[53] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1, 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[54] Saul Dobilas. *LSTM Recurrent Neural Networks — How to Teach a Network to Remember the Past*. Medium. Mar. 5, 2022. URL: https://towardsdatascience.com/lstm-recurrent-neural-networks-how-to-teach-a-network-to-remember-the-past-55e54c2ff22e (visited on 01/28/2024).

[55] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. Dec. 14, 2014. DOI: 10.48550/arXiv.1409.3215. arXiv: 1409.3215 [cs]. preprint.

[56] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. May 19, 2016. DOI: 10.48550/arXiv.1409.0473. arXiv: 1409.0473 [cs, stat]. preprint.

[57] *Part 1: Key Concepts in RL — Spinning Up Documentation*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html (visited on 11/20/2023).

[58] Christopher Watkins. "Learning From Delayed Rewards". PhD thesis. Cambridge University, Jan. 1, 1989. 241 pp. URL: https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[59] Christopher J. C. H. Watkins and Peter Dayan. "Q-Learning". In: *Machine Learning* 8.3 (May 1, 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[60] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. "Deep Reinforcement Fuzzing". In: 2018 IEEE Security and Privacy Workshops (SPW). IEEE Computer Society, May 1, 2018, pp. 116–122. ISBN: 978-1-5386-8276-0. DOI: 10.1109/SPW.2018.00026.

[61] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Cham: Springer International Publishing, 2010. ISBN: 978-3-031-01551-9. DOI: 10.1007/978-3-031-01551-9.

[62] *Microcontrollers Get a Lift from Automotive After 2021 Rebound*. Design And Reuse. URL: https://www.design-reuse.com/news/51679/mcu-market-history-and-forecast.html (visited on 11/16/2023).

[63] Wassen Mohammad, Adel Elomri, and Laoucine Kerbache. "The Global Semiconductor Chip Shortage: Causes, Implications, and Potential Remedies". In: *IFAC-PapersOnLine*. 10th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2022 55.10 (Jan. 1, 2022), pp. 476–483. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2022.09.439.

[64] Roddy Urquhart. *What Does RISC-V Stand For?* Semiconductor Engineering. Mar. 29, 2021. URL: https://codasip.com/2021/03/17/what-does-risc-v-stand-for/ (visited on 01/17/2024).

[65] *MIPS32 Architecture – MIPS*. URL: https://mips.com/products/architectures/mips32-2/ (visited on 01/17/2024).

[66] Arm Ltd. *Architecture*. Arm — The Architecture for the Digital World. URL: https://www.arm.com/architecture (visited on 01/17/2024).

[67] Nitin Dahad. *Embedded Survey 2023: More Software/ Hardware/ IP Reuse*. Embedded.com. June 18, 2023. URL: https://www.embedded.com/embedded-survey-2023-more-ip-reuse-as-workloads-surge/ (visited on 01/17/2024).

[68] Tammy Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Embedded Technology Series. Amsterdam Boston: Elsevier/Newnes, 2005. ISBN: 978-0-7506-7792-9.

[69] Marius Muench et al. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices". In: *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2018. ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23166.

[70] Zicong Gao et al. "Fw-fuzz: A Code Coverage-guided Fuzzing Framework for Network Protocols on Firmware". In: *Concurrency and Computation: Practice and Experience* 34.16 (July 25, 2022), e5756. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.5756.

[71] Wenqiang Li et al. "muAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware". In: *Proceedings of the 44th International Conference on Software Engineering*. May 21, 2022, pp. 1–12. DOI: 10.1145/3510003.3510208. arXiv: 2202.03013 [cs].

[72] Kurt Rosenfeld and Ramesh Karri. "Attacks and Defenses for JTAG". In: *IEEE Design & Test of Computers* 27.1 (Jan. 2010), pp. 36–47. ISSN: 0740-7475. DOI: 10.1109/MDT.2010.9.

[73] Mdsec Research. *Mdsecresearch/UARTFuzz*. Sept. 28, 2023. URL: https://github.com/mdsecresearch/UARTFuzz (visited on 01/13/2024).

[74] Rong Fan, Jianfeng Pan, and Shaomang Huang. "ARM-AFL: Coverage-Guided Fuzzing Framework for ARM-Based IoT Devices". In: *Applied Cryptography and Network Security Workshops*. Ed. by Jianying Zhou et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 239–254. ISBN: 978-3-030-61638-0. DOI: 10.1007/978-3-030-61638-0_14.

[75] Leila Delshadtehrani et al. "PHMon: A Programmable Hardware Monitor and Its Security Use Cases". In: 29th USENIX Security Symposium (USENIX Security 20). 2020, pp. 807–824. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/delshadtehrani (visited on 01/13/2024).

[76] Dokyung Song et al. "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary". In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23176.

[77] Philip Sperl and Konstantin Böttinger. "Side-Channel Aware Fuzzing". In: *Computer Security – ESORICS 2019*. Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 259–278. ISBN: 978-3-030-29959-0. DOI: 10.1007/978-3-030-29959-0_13.

[78] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. *Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces*. Nov. 11, 2015. DOI: 10.48550/arXiv.1511.03609. arXiv: 1511.03609 [cs]. preprint.

[79] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: 2005 USENIX Annual Technical Conference (USENIX ATC 05). 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator (visited on 01/11/2024).

[80] Daming D. Chen et al. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware". In: *Proceedings 2016 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2016. ISBN: 978-1-891562-41-9. DOI: 10.14722/ndss.2016.23415.

[81] Mingeun Kim et al. "FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis". In: *Annual Computer Security Applications Conference*. ACSAC '20: Annual Computer Security Applications Conference. Austin USA: ACM, Dec. 7, 2020, pp. 733–745. ISBN: 978-1-4503-8858-0. DOI: 10.1145/3427228.3427294.

[82] Yaowen Zheng et al. "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation". In: 28th USENIX Security Symposium (USENIX Security 19). 2019, pp. 1099–1114. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng (visited on 01/11/2024).

[83] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. "Prospect: Peripheral Proxying Supported Embedded Code Testing". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. New York, NY, USA: Association for Computing Machinery, June 4, 2014, pp. 329–340. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590301.

[84] Hertz, Jesse and Newsham, Tim. *Whitepaper – Project Triforce: Run AFL On Everything (2017)*. Whitepaper. NCC Group, 2017, p. 43. URL: https://research.nccgroup.com/2022/09/27/whitepaper-project-triforce-run-afl-on-everything-2017/ (visited on 01/12/2024).

[85] Jonas Zaddach et al. "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares". In: *Proceedings 2014 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2014. ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23229.

[86] Vladimir Herdt et al. "Verification of Embedded Binaries Using Coverage-guided Fuzzing with SystemC-based Virtual Prototypes". In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. GLSVLSI '20: Great Lakes Symposium on VLSI 2020. Virtual Event China: ACM, Sept. 7, 2020, pp. 101–106. ISBN: 978-1-4503-7944-1. DOI: 10.1145/3386263.3406899.

[87] Karl Koscher, Tadayoshi Kohno, and David Molnar. "SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems". In: 9th USENIX Workshop on Offensive Technologies (WOOT 15). 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher (visited on 01/13/2024).

[88] Eric Gustafson et al. "Toward the Analysis of Embedded Firmware through Automated Re-hosting". In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). 2019, pp. 135–150. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson (visited on 06/12/2023).

[89] Bo Feng, Alejandro Mera, and Long Lu. "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling". In: 29th USENIX Security Symposium (USENIX Security 20). 2020, pp. 1237–1254. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/feng (visited on 01/13/2024).

[90] Tobias Scharnowski et al. "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing". In: USENIX Security Symposium. 2022. URL: https://www.semanticscholar.org/paper/Fuzzware%3A-Using-Precise-MMIO-Modeling-for-Effective-Scharnowski-Bars/17b0422509e7a7cc1de5a9614c82f4a4adc43ad3 (visited on 12/21/2023).

[91] Nguyen, Anh Quynh and Dang, Hoang Vu. *Unicorn – The Ultimate CPU Emulator*. 2015. URL: https://www.unicorn-engine.org/ (visited on 01/11/2024).

[92] Wei Zhou et al. "Automatic Firmware Emulation through Invalidity-guided Knowledge Inference". In: 30th USENIX Security Symposium (USENIX Security 21). 2021, pp. 2007–2024. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou (visited on 01/14/2024).

[93] ReFirmLabs. *ReFirmLabs/Binwalk*. ReFirm Labs, Jan. 11, 2024. URL: https://github.com/ReFirmLabs/binwalk (visited on 01/11/2024).

[94] *OFRAK: Unpack, Modify, and Repack Binaries*. URL: https://ofrak.com/ (visited on 01/11/2024).

[95] Shakir. *Hardware Debug Ports: A Definitive How-To Guide*. Payatu. Jan. 25, 2023. URL: https://payatu.com/blog/iot-security-part-14-introduction-to-and-identification-of-hardware-debug-ports/ (visited on 01/16/2024).

[96] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: Machine Learning for Input Fuzzing". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '17. Urbana-Champaign, IL, USA: IEEE Press, Oct. 30, 2017, pp. 50–59. ISBN: 978-1-5386-2684-9. DOI: 10.48550/arXiv.1701.07232.

[97] Liang Cheng et al. "Optimizing Seed Inputs in Fuzzing with Machine Learning". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). May 2019, pp. 244–245. DOI: 10.1109/ICSE-Companion.2019.00096.

[98] Mohit Rajpal, William Blum, and Rishabh Singh. *Not All Bytes Are Equal: Neural Byte Sieve for Fuzzing*. Nov. 9, 2017. DOI: 10.48550/arXiv.1711.04596. arXiv: 1711.04596 [cs]. preprint.

[99] Rong Fan and Yaoyao Chang. "Machine Learning for Black-Box Fuzzing of Network Protocols". In: *Information and Communications Security*. Ed. by Sihan Qing et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 621–632. ISBN: 978-3-319-89500-0. DOI: 10.1007/978-3-319-89500-0_53.

[100] Ciprian Paduraru and Marius-Constantin Melemciuc. "An Automatic Test Data Generation Tool Using Machine Learning". In: *Proceedings of the 13th International Conference on Software Technologies*. 13th International Conference on Software Technologies. Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 472–481. DOI: 10.1007/978-3-030-12146-4_22.

[101] Chris Cummins et al. "Compiler Fuzzing through Deep Learning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '18: International Symposium on Software Testing and Analysis. Amsterdam Netherlands: ACM, July 12, 2018, pp. 95–105. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213848.

[102] Martin Sablotny, Bjørn Sand Jensen, and Chris W. Johnson. "Recurrent Neural Networks for Fuzz Testing Web Browsers". In: *Information Security and Cryptology – ICISC 2018*. Ed. by Kwangsu Lee. Cham: Springer International Publishing, 2019, pp. 354–370. ISBN: 978-3-030-12146-4. DOI: 10.1007/978-3-030-12146-4_22.

[103] Morteza Zakeri Nasrabadi, Saeed Parsa, and Akram Kalaee. "Format-Aware Learn&Fuzz: Deep Test Data Generation for Efficient Fuzzing". In: *Neural Computing and Applications* 33.5 (Mar. 2021), pp. 1497–1513. ISSN: 0941-0643, 1433-3058. DOI: 10.1007/s00521-020-05039-7. arXiv: 1812.09961 [cs].

[104] Dongdong She et al. *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*. July 12, 2019. DOI: 10.48550/arXiv.1807.05620. arXiv: 1807.05620 [cs]. preprint.

[105] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018 IEEE Symposium on Security and Privacy (SP). May 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[106] Sheila Becker et al. "An Autonomic Testing Framework for IPv6 Configuration Protocols". In: *Mechanisms for Autonomous Management of Networks and Services*. Ed. by Burkhard Stiller and Filip De Turck. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 65–76. ISBN: 978-3-642-13986-4. DOI: 10.1007/978-3-642-13986-4_7.

[107] William Drozd and Michael D. Wagner. *FuzzerGym: A Competitive Framework for Fuzzing and Learning*. July 19, 2018. arXiv: 1807.07490 [cs]. URL: http://arxiv.org/abs/1807.07490 (visited on 01/09/2024). preprint.

[108] Xiaoting Li et al. "FuzzBoost: Reinforcement Compiler Fuzzing". In: *Information and Communications Security: 24th International Conference, ICICS 2022, Canterbury, UK, September 5–8, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Sept. 5, 2022, pp. 359–375. ISBN: 978-3-031-15776-9. DOI: 10.1007/978-3-031-15777-6_20.

[109] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. "RiverFuzzRL - an Open-Source Tool to Experiment with Reinforcement Learning for Fuzzing". In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). Apr. 2021, pp. 430–435. DOI: 10.1109/ICST49551.2021.00055.

[110] Jared D. DeMott and R. Enbody. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing". In: 2007. URL: https://www.semanticscholar.org/paper/Revolutionizing-the-Field-of-Grey-box-Attack-with-DeMott-Enbody/f9d7ad9ec1f7a082a84130d9b7071ddfc6db30c5 (visited on 01/15/2024).

[111] Guang-Hong Liu et al. "Vulnerability Analysis for X86 Executables Using Genetic Algorithm and Fuzzing". In: *2008 Third International Conference on Convergence and Hybrid Information Technology*. 2008 Third International Conference on Convergence and Hybrid Information Technology. Vol. 2. Nov. 2008, pp. 491–497. DOI: 10.1109/ICCIT.2008.9.

[112] Fabien Duchene. "Fuzz in the Dark: Genetic Algorithm for Black-Box Fuzzing". In: Nov. 27, 2013. URL: https://inria.hal.science/hal-00978844.

[113] Saul Dobilas. *Reinforcement Learning with SARSA — A Good Alternative to Q-Learning Algorithm*. Medium. Dec. 20, 2023. URL: https://towardsdatascience.com/reinforcement-learning-with-sarsa-a-good-alternative-to-q-learning-algorithm-bf35b209e1c (visited on 01/15/2024).

[114] Bogdan Ghimis, Miruna Paduraru, and Alin Stefanescu. "RIVER 2.0: An Open-Source Testing Framework Using AI Techniques". In: *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*. LANGETI 2020. New York, NY, USA: Association for Computing Machinery, Nov. 8, 2020, pp. 13–18. ISBN: 978-1-4503-8123-9. DOI: 10.1145/3416504.3424335.

[115] Jinkyu Koo et al. "PySE: Automatic Worst-Case Test Generation by Reinforcement Learning". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). Xi'an, China: IEEE, Apr. 2019, pp. 136–147. ISBN: 978-1-72811-736-2. DOI: 10.1109/ICST.2019.00023.

[116] Joshua Pereyda. *Jtpereyda/Boofuzz*. Jan. 17, 2024. URL: https://github.com/jtpereyda/boofuzz (visited on 01/17/2024).

[117] PeachTech. *What Is Peach?* URL: https://peachtech.gitlab.io/peach-fuzzer-community/WhatIsPeach.html (visited on 01/17/2024).

[118] Zhenhua Yu et al. "CGFuzzer: A Fuzzing Approach Based on Coverage-Guided Generative Adversarial Networks for Industrial IoT Protocols". In: *IEEE Internet of Things Journal* 9.21 (Nov. 2022), pp. 21607–21619. ISSN: 2327-4662. DOI: 10.1109/JIOT.2022.3183952.

[119] Ian J. Goodfellow et al. *Generative Adversarial Networks*. June 10, 2014. DOI: 10.48550/arXiv.1406.2661. arXiv: 1406.2661 [cs, stat]. preprint.

[120] Lantao Yu et al. *SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient*. Aug. 25, 2017. DOI: 10.48550/arXiv.1609.05473. arXiv: 1609.05473 [cs]. preprint.

[121] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dec. 2, 2015. DOI: 10.48550/arXiv.1511.08458. arXiv: 1511.08458 [cs]. preprint.

[122] Zhicheng Hu et al. "GANFuzz: A GAN-based Industrial Network Protocol Fuzzing Framework". In: *Proceedings of the 15th ACM International Conference on Computing Frontiers* (May 8, 2018), pp. 138–145. DOI: 10.1145/3203217.3203241.

[123] Hui Zhao et al. “SeqFuzzer: An Industrial Protocol Fuzzing Framework from a Deep Learning Perspective”. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). Apr. 2019, pp. 59–67. DOI: 10.1109/ICST.2019.00016.

[124] Yuqi Chen et al. “Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2019, pp. 962–973. DOI: 10.1109/ASE.2019.00093. arXiv: 1909.05410 [cs].

[125] Yan Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). May 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[126] Yan Shoshitaishvili et al. “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware”. In: *Proceedings 2015 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2015. ISBN: 978-1-891562-38-9. DOI: 10.14722/ndss.2015.23294.

[127] Anton Chernyavskiy, Dmitry Ilvovsky, and Preslav Nakov. “Transformers: “The End of History” for Natural Language Processing?” In: *Machine Learning and Knowledge Discovery in Databases. Research Track*. Ed. by Nuria Oliver et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 677–693. ISBN: 978-3-030-86523-8. DOI: 10.1007/978-3-030-86523-8_41.

[128] Anthony Gillioz et al. “Overview of the Transformer-based Models for NLP Tasks”. In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. 2020 15th Conference on Computer Science and Information Systems (FedCSIS). Sept. 2020, pp. 179–183. DOI: 10.15439/2020F20.

[129] Saidul Islam et al. *A Comprehensive Survey on Applications of Transformers for Deep Learning Tasks*. June 11, 2023. DOI: 10.48550/arXiv.2306.07303. arXiv: 2306.07303 [cs]. preprint.

[130] Vladimir Lyashenko. *The Best Tools for Reinforcement Learning in Python You Actually Want to Try*. neptune.ai. July 21, 2022. URL: https://neptune.ai/blog/the-best-tools-for-reinforcement-learning-in-python (visited on 12/21/2023).

[131] *Top 6 Reinforcement Learning Tools to Use*. URL: https://www.turing.com/kb/best-tools-for-reinforcement-learning (visited on 01/12/2024).

[132] Ram Sagar. *Top 7 Python Libraries For Reinforcement Learning*. Analytics India Magazine. Jan. 13, 2020. URL: https://analyticsindiamag.com/python-libraries-reinforcement-learning-dqn-rl-ai/ (visited on 01/12/2024).

[133] Mauricio Fadel Argerich. *5 Frameworks for Reinforcement Learning on Python*. Medium. June 6, 2020. URL: https://towardsdatascience.com/5-frameworks-for-reinforcement-learning-on-python-1447fede2f18 (visited on 01/12/2024).

[134] mirroer. *Afl/Docs/Technical_details.Txt at Master · Mirrorer/Afl*. GitHub. URL: https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt (visited on 01/22/2024).

[135] lcamtuf. *Fuzzing Random Programs without Execve()*. lcamtuf's old blog. Oct. 14, 2014. URL: https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html (visited on 01/22/2024).

List of Figures

List of Tables

Typeset January 29, 2024