M.ENG PROJECT REPORT - 699A

UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Enhancing Fuzz Testing with LLM-Generated Content

Report submitted by:
Eshani Nandy
(Student ID: 21116915)

Project Supervisor:
Prof. Sebastian Fischmeister

Project Advisor:
Cameron Hadfield

Date: August 12, 2025

Contents

1 Introduction

Fuzz testing is a widely used technique to identify security flaws in software systems by repeatedly sending malformed or unexpected inputs to a target application. It has proven effective in discovering memory corruption, logic bugs, and input validation issues across both industry and academia [1, 2]. However, when applied to embedded systems such as IoT devices, fuzzing faces unique challenges. These systems often operate on constrained hardware with limited interfaces and typically require manual resets after a crash, making large-scale automated fuzzing difficult [3, 4]. Furthermore, many IoT devices run as black-box binaries without source code access or protocol documentation, restricting the ability to design structured test cases. Traditional fuzzers, which often rely on random input generation without an understanding of expected input formats, tend to achieve low code coverage and may miss vulnerabilities [5]. These constraints create a need for strategies that can generate meaningful, protocol-aware inputs and support automated execution on physical hardware, even in the absence of source code.

1.1 Problem Statement

The problem addressed in this work is: How can large language models be integrated into fuzz testing to create a reproducible benchmark for evaluating fuzzing strategies on embedded IoT firmware running on physical hardware? Without smarter input generation, hardware fuzzing efforts risk remaining shallow and overlooking critical vulnerabilities hidden deeper in the firmware logic. [4, 6].

1.2 Aims and objectives

This project aims to evaluate whether large language models (LLMs) can improve the effectiveness of fuzz testing on embedded IoT firmware running on real hardware. To achieve this, we first investigate available fuzzing benchmarks and select a firmware that can be ported onto a Raspberry Pi for hardware-based testing. We reviewed several benchmarks, including IoTFuzzBench [7], ProFuzzBench [8], and IoTVulBench [9], to determine their suitability for physical deployment. Based on this analysis, we chose the OWASP IoTGoat [10] firmware as the benchmark due to its hardware compatibility and documented vulnerabilities. Next, we perform

baseline fuzz testing using the Boofuzz framework, targeting known vulnerable services within the firmware. Next, a local LLM is integrated into the fuzzing setup to generate semantically rich test inputs that follow the expected command structure of the target service. We extended the Boofuzz framework to inject these LLM-generated test cases and to detect failure conditions such as service crashes or device unresponsiveness. The goal is to demonstrate that LLM-guided fuzzing can discover faults more effectively or efficiently than conventional approaches.

1.3 Related work

Recent studies have begun integrating large language models (LLMs) into fuzz testing to overcome limitations of traditional fuzzers. For instance, Meng et al. propose an LLM-guided protocol fuzzing approach that taps into an LLM's knowledge of protocol specifications written in natural language [4]. Their system, CHATAFL, interacts with an LLM (e.g., ChatGPT) to generate grammars for protocol message formats and suggest valid message sequences. This allows the fuzzer to explore complex, stateful communication flows even without a formal machine-readable specification. In evaluations on real-world network servers, it achieved significantly higher state coverage than conventional protocol fuzzers and uncovered nine previously unknown vulnerabilities, far more than baseline tools.

Similarly, Xia et al. presents Fuzz4All, which uses LLMs to perform universal fuzzing across different input languages [6]. Traditional grammar-based fuzzers are typically tailored to one input format (e.g., C, JSON), but Fuzz4All leverages an LLM as a general-purpose generator and mutator. It uses an autoprompting technique to guide the LLM in producing valid and diverse inputs across many target formats. Tested on nine real-world systems covering six languages, the LLM-driven fuzzer achieved superior coverage and found 98 new bugs, 64 of which were previously unknown, across projects like GCC, Clang, Z3, OpenJDK, and others.

These works highlight the promise of LLM-assisted fuzzing in conventional software domains. However, they do not address fuzzing of embedded systems, where devices operate under resource constraints and often lack formal specifications. Our project extends the principles of LLM-guided fuzzing into this domain, using LLMs to automatically generate meaningful test cases for embedded firmware with opaque or poorly documented interfaces.

1.4 Fuzzing Benchmarks Evaluation and Selection

Evaluating fuzzing techniques on embedded systems necessitates standardized benchmarks that reflect real-world firmware behavior, security vulnerabilities, and interface diversity. Traditional fuzzing benchmarks often target user-space programs or protocol servers in isolation, falling short when applied to IoT firmware. Several recent benchmarks have emerged that aim to fill this gap, each with distinct strengths and limitations which are discussed in Table 1.1.

1.4.1 IoTFuzzBench (2023)

IoTFuzzBench is a reproducible evaluation framework tailored for fuzz testing IoT firmware in emulated environments [7]. It includes 14 real firmware images from commercial devices, each embedding approximately 30 known vulnerabilities and running on QEMU-based virtual machines orchestrated via Docker. Its design focuses on black-box fuzzing through network interfaces like Telnet, HTTP, or DNS, making it a pragmatic option when source code or instrumentation is unavailable.

Key strengths of IoTFuzzBench include its structured protocol interfaces, well-suited for LLM-assisted input generation, and its coverage of real-world firmware. It allows early-stage evaluation without requiring physical hardware, but its limited firmware set and emphasis on emulation constrain scalability and realism for hardware fuzzing.

1.4.2 ProFuzzBench (2021)

ProFuzzBench is a benchmark suite focused on fuzzing stateful network protocols. Rather than firmware, it includes real-world server applications such as FTP, SMTP, and SSH, deployed in containerized environments [8]. It tracks code coverage and crash reports while supporting advanced fuzzers like AFLNet and ChatAFL.

ProFuzzBench is particularly relevant to LLM-driven fuzzing, as shown in studies like CHATAFL [4], where LLMs improved coverage and protocol state exploration. However, ProFuzzBench is best viewed as complementary, it lacks the constraints and authenticity of full IoT firmware.

1.4.3 IoTVulBench/ IoTBenchSL (2025)

IoTVulBench is the most comprehensive and modern benchmark available for embedded firmware fuzzing [9]. It features over 100 real firmware images, many sourced from commercial IoT devices, each annotated with known vulnerabilities and automated exploit validation scripts. Its CI/CD pipeline manages QEMU emulation, test orchestration, and crash recording. Importantly, it uncovered 32 previously unknown vulnerabilities, including 21 CVEs, through its integrated fuzzing tools.

IoTVulBench is particularly aligned with LLM-guided fuzzing research due to its structured interfaces (e.g., HTTP, Telnet), real vulnerabilities, and automation features. Despite its emulation focus, it allows for extrapolation to hardware scenarios with appropriate care.

1.4.4 Benchmark Selection for Hardware Fuzzing

When contrasting IoTVulBench with ProFuzzBench and IoTFuzzBench, the distinctions become clear. ProFuzzBench excels in protocol fuzzing and is well suited to testing LLM-driven tools on standard services, but lacks firmware realism. IoTFuzzBench provides firmware authenticity but at a smaller scale. IoTVulBench combines both worlds: a large and diverse firmware corpus, real embedded service exposure, and automation tooling. These qualities make it a compelling choice for evaluating LLM-guided fuzzing strategies on realistic IoT firmware. Despite its advantages, deploying IoTVulBench on actual hardware remains complex due to architecture constraints, storage limitations, and packaging requirements. Since none of the surveyed benchmarks are readily deployable on our target hardware platform, we instead selected IoTGoat, a smaller-scale but hardware-ready vulnerable firmware developed by OWASP [10]. Its compatibility with devices like the Raspberry Pi 2 and inclusion of multiple exploitable services make it well-suited for testing fuzzing strategies in real-world embedded environments. In Section 1.4.5, we explore IoTGoat in more detail, including the factors that make it a suitable replacement for the larger emulation-focused benchmarks in our hardware fuzzing experiments.

1.4.5 IotGoat as a Fuzzing Benchmark

While existing benchmarks like IoTFuzzBench and IoTVulBench offer realistic firmware samples and automated emulation environments, they are not easily deployable on physical devices due to hardware compatibility issues, packaging constraints, and

Table 1.1: Comparison of IoT Fuzzing Benchmarks

Benchmark	Pros	Limitations
IoTFuzzBench (2023)	• Real IoT firmware for authentic embedded contexts • Easy-to-deploy Docker/QEMU infrastructure • Pre-defined vulnerabilities and seeds for comparison • Compatible with LLM-guided input generation over network protocols	• Only 14 firmware samples • Hardware deployment requires complex manual porting
ProFuzzBench (2021)	• Excellent for protocol-level evaluation • Proven baseline for LLM-guided approaches • Automated, scriptable environment • Ideal for testing stateful input generation	• No embedded firmware or hardware representation • Not directly portable to physical devices
IoTVulBench / IoTBenchSL (2025)	• 100+ firmware samples with real vulnerabilities • Reproducible fuzzing pipeline with CI integration • Support for structured protocols, ideal for LLM-generated input • Demonstrated practical impact via discovered CVEs	• Requires adaptation for physical deployment • Hardware compatibility and flashing are non-trivial

architecture mismatches. To address the practical need for fuzzing on real embedded systems, IotGoat, a deliberately vulnerable firmware developed by OWASP, is selected as the benchmark for hardware fuzzing.

IoTGoat is based on OpenWrt and is designed to emulate the insecure configurations found in commercial IoT devices. It includes common vulnerabilities such as hardcoded credentials, exposed services like Telnet and HTTP, and a backdoor shell service named shellback. This makes it ideal for testing LLM-guided fuzzing techniques targeting command-parsing interfaces.

Unlike larger benchmarks that require virtualization and may not run correctly outside emulators, IoTGoat runs natively on a Raspberry Pi 2, making it a practical and realistic target for black-box fuzzing experiments on physical hardware. The firmware can be directly flashed onto an SD card, and it supports SSH and serial access for monitoring and interaction.

Given its combination of real attack surfaces, low setup overhead, and hardware compatibility, IoTGoat serves as an effective platform to evaluate fuzzing frameworks under real-world conditions. It allows this project to measure how well LLM-generated inputs can trigger faults in embedded environments and whether such approaches outperform or complement traditional fuzzing workflows.

1.5 Fuzzing Framework

A fuzzing framework provides the infrastructure to automate the generation and delivery of malformed or unexpected inputs to a target system, enabling the discovery of bugs, crashes, or security vulnerabilities [1]. Such frameworks are essential for scaling fuzz testing efforts and integrating advanced input generation techniques. some text about what fuzzing is and why we need it

1.5.1 Boofuzz

Boofuzz is an open-source network protocol fuzzing framework written in Python [11]. It evolved from the now-unmaintained Sulley framework and provides a modular, scriptable platform for fuzz testing services that accept structured inputs over network sockets or serial connections. Boofuzz is widely used in academic and industry settings due to its extensibility, ease of use, and built-in monitoring support.

The framework allows testers to define protocol grammars as sequences of tokens (e.g., command prefixes, delimiters, payloads) and automatically mutates them to explore unexpected behaviors or trigger crashes. It supports TCP, UDP, and serial

connections, and integrates pre- and post-send callbacks for custom monitoring or state tracking.

In this project, Boofuzz is used as the base fuzzing framework to transmit test inputs to the vulnerable IoTGoat firmware. It manages session creation, input delivery, and crash detection via connection state monitoring. By default, Boofuzz fuzzes each field individually using built-in mutation strategies. However, in the LLM-guided setup, Boofuzz is modified to send entire inputs generated by the language model without performing internal mutations. This hybrid structure retains Boofuzz's robust session handling while allowing intelligent test case generation to be offloaded to a large language model.

Boofuzz also includes a lightweight web interface for monitoring the progress of a fuzzing session, tracking input lengths, crash statistics, and target responsiveness in real time.

1.6 Large Language Models (LLMs)

Large Language Models (LLMs) are pre-trained neural networks capable of performing language-related tasks such as text generation, reasoning, and code synthesis [12, 13]. Their ability to interpret prompts and generate structured outputs makes them useful for software testing tasks, including fuzzing.

Since many embedded services use text-based protocols (e.g., HTTP, Telnet), and their grammars are publicly documented, LLMs can leverage their training on internet-scale corpora to produce valid or semi-valid test inputs. This makes them promising tools for generating meaningful fuzzing payloads that explore deeper system behavior.

1.6.1 LLMs for fuzzing

In this project, we explore whether LLMs can improve the effectiveness of fuzz testing on embedded firmware. Specifically, we integrate GPT4All, an open-source framework for running LLMs locally, to generate fuzz inputs. GPT4All supports a range of quantized models optimized for offline inference, making it suitable for deployment on resource-constrained platforms like the Raspberry Pi [14]. Unlike cloud-based LLMs, GPT4All operates entirely on the local device, ensuring reproducibility, privacy, and independence from external APIs.

This local integration enables structured, prompt-based test case generation without manual crafting or extensive seeding. Prior work such as CHATAFL has

shown that LLMs can enhance protocol fuzzing by generating valid message sequences and inferring grammar-like structures [4]. Our goal is to evaluate whether a similar approach can yield comparable benefits in a constrained, hardware-based fuzzing setup.

2 Methodology

This chapter outlines the step-by-step methodology adopted to develop, deploy, and test the LLM-guided fuzzing system on a real embedded target. The overall process consists of selecting a hardware-compatible benchmark, flashing the firmware, identifying vulnerable interfaces, designing a fuzzing harness using Boofuzz, and finally augmenting the system with GPT4All to explore the potential of LLM-generated test inputs.

2.1 Porting IoTGoat to Raspberry Pi 2

The OWASP IoTGoat firmware was selected as the benchmark due to its deliberately insecure design and ease of deployment on physical hardware. While many fuzzing benchmarks, such as IoTFuzzBench and IoTVulBench, focus on emulated environments, IoTGoat provides prebuilt .img files for the Raspberry Pi 2, enabling real device fuzzing. The experiments used a Raspberry Pi 2 Model B Rev 1.1, featuring a Broadcom BCM2835 SoC with a quad-core ARMv7 processor (v7l). Hardware specifications are shown in Figure 2.1.

The "IoTGoat-raspberry-pi2.img" firmware was downloaded from OWASP's GitHub releases [15] and flashed to a 16 GB microSD card using Raspberry Pi Imager. The Pi was powered via micro-USB and connected over Ethernet for headless access.

2.2 Establishing SSH Access to IotGoat Firmware

After flashing the IoTGoat firmware, the Raspberry Pi 2 (RPi2) was powered on and connected via Ethernet to a host machine. As IoTGoat does not assign IPs dynamically, the host was manually configured with a compatible static IP to initiate communication.

Default SSH credentials were not documented, so passwords for the `root` and `iotgoatuser` accounts were recovered by analyzing the extracted firmware image offline using standard techniques. This enabled successful SSH access to the device.

To simplify ongoing communication, the RPi2's IP address was updated to match the host subnet. While not a complex step, it ensured consistent connectivity throughout the fuzzing experiments.

Once configured, the SSH connection was used to monitor and control the target

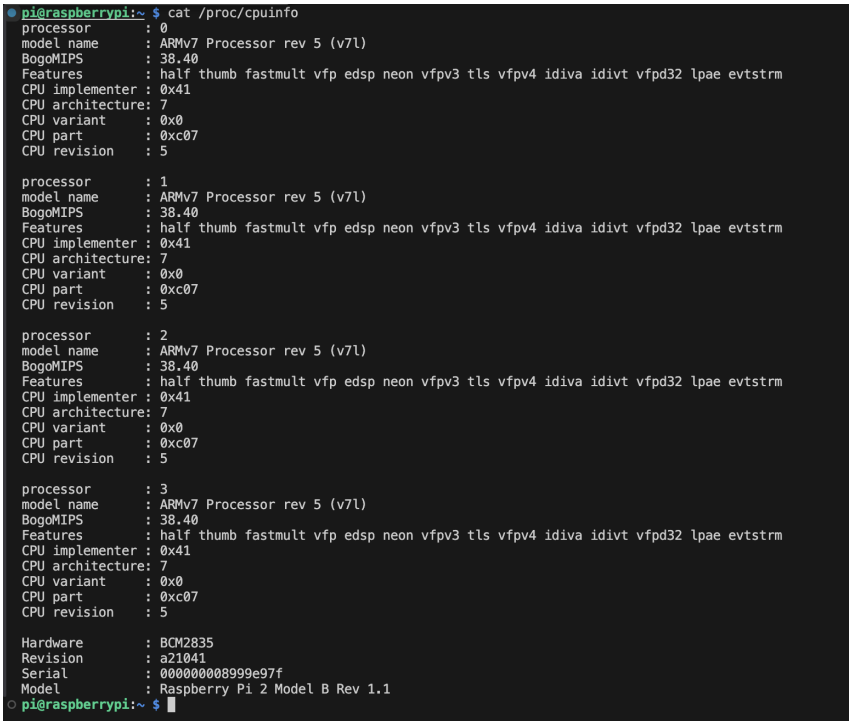

Figure 2.1: Raspberry Pi 2 specifications

system. This setup allowed tools like BooFuzz to operate reliably without interruptions from dynamic IP changes.

2.3 Exploring IoTGoat and Identifying Vulnerable Ports

With SSH access successfully established, the next step was to identify which services were actively listening on the IoTGoat system. This was done by logging in as the recovered `iotgoatuser` account and enumerating active TCP and UDP ports using standard system utilities. Although executed without root privileges, the output was sufficient to identify several open ports, as shown in Figure 2.2.

The following ports were confirmed open:

- **22**: SSH (Dropbear)
- **53**: DNS (dnsmasq)
- **80/443**: Web interface (HTTP/HTTPS)
- **5515**: Shellback backdoor (unauthenticated root shell)
- **65534**: Telnet
- **5000, 1900, 5351, 47068**: Likely auxiliary services (e.g., UPnP, SSDP)

Among these, port **5515**, linked to the shellback binary, was selected for focused

```
iotgoatuser@IoTGoat:~$ netstat -tulnp
netstat: can't scan /proc - are you root?
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign
Address         State       PID/Program name
tcp        0      0 0.0.0.0:5515            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:53              0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:443             0.0.0.0:*               LISTEN      -
tcp        0      0 :::5000                 :::*                    LISTEN      -
tcp        0      0 :::80                   :::*                    LISTEN      -
tcp        0      0 :::53                   :::*                    LISTEN      -
tcp        0      0 :::22                   :::*                    LISTEN      -
tcp        0      0 :::443                  :::*                    LISTEN      -
tcp        0      0 :::65534                :::*                    LISTEN      -
udp        0      0 0.0.0.0:53              0.0.0.0:*                           -
udp        0      0 192.168.2.2:5351        0.0.0.0:*                           -
udp        0      0 0.0.0.0:1900            0.0.0.0:*                           -
udp        0      0 192.168.2.2:47068       0.0.0.0:*                           -
udp        0      0 :::53                   :::*                                -
```

Figure 2.2: Running `netstat -tulno` as the `iotgoatuser` to enumerate open ports.

fuzz testing. Several characteristics made it a particularly attractive target:

- It exposes an unauthenticated root shell to any client, posing a critical security vulnerability by design.
- It accepts arbitrary input in an interactive shell environment, increasing the likelihood that malformed or unexpected inputs could trigger crashes.
- It lacks a formal communication protocol, making it ideal for creative fuzzing approaches, especially those leveraging LLM-generated input without protocol grammar constraints.
- As the service runs with root privileges, any induced crash or instability could lead to full system compromise or reboot, amplifying the impact of successful fuzzing.

These properties positioned the shellback service as a high-value, realistic target for evaluating the effectiveness of intelligent fuzzing strategies guided by large language models (LLMs).

2.4 Manual Exploration of the Shellback Service on Port 5515

To understand the behavior of the `shellback` service prior to fuzzing, a series of manual tests were performed over a direct TCP connection to port `5515`. The service responded with a banner as shown in Figure 2.3:

A shell-like interface was presented, and a variety of command-line inputs were

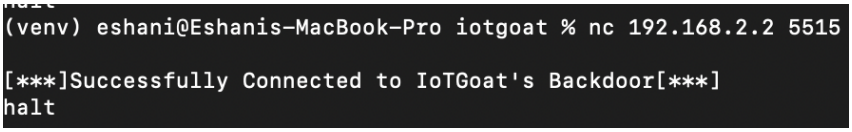

Figure 2.3: Successful entry into backdoor through port 5515

tested to assess how the service parsed and handled incoming data as shown in Table 2.1.

Table 2.1: Examples of LLM-generated fuzzing payloads

Category	**Example Payloads**
Standard shell commands	`ls`, `whoami`, `id`, `echo test`, `cat /etc/passwd`
Shell metacharacters and command chaining	`; ls -al`, `| echo owned`, `$(ls)`, `$(reboot)`
Special characters and format string probes	`%x%x%x%x`, `%n`, `AAAA...AAAAAAAAAAAAAAA`, `\xff\xff\xff\xff`
Obfuscated command variants and encodings	`Reb\toot`, `shut_down`, `h\u0061lt`

These manual inputs served as a foundation for later automated fuzzing, including guiding prompt design for LLM-generated test cases. They were chosen to test the shell's tolerance to malformed syntax, encoded characters, and common payload patterns used in vulnerability discovery.

The interactive nature of the shellback service and the fact that it executed input without authentication made it an ideal candidate for fuzzing and payload mutation experiments, as discussed in subsequent sections.

2.5 Boofuzz-Based Fuzzing

To establish a baseline for fuzz testing, the Boofuzz framework was used to target the shellback backdoor service running on TCP port `5515` of the IoTGoat firmware. Boofuzz is a Python-based fuzzing engine that allows custom protocol definitions and automated mutation of input structures.

2.5.1 Test Input Construction

The shellback service accepts command-style input over a TCP socket. Based on manual testing and inspection, each input was structured as:

```
<prefix> <payload>\n
```

Boofuzz was configured to generate test inputs using predefined lists of command prefixes and payloads. The command prefixes were inspired by common administrative operations, while the payloads included benign strings, risky commands, and obfuscated variants (Table 2.2).

Table 2.2: Prefixes and Payloads Used in Fuzzing

Prefixes	**Payloads**
`ping`, `exec`, `run`, `exit`, `get`, `set`, `info`, `config`, `update`, `version`	• Standard: `admin`, `root`, `ls`, `echo test`, `echo fuzzed` • Format strings: `%x%x%x%x`, `%n` • Long/binary: `A*64`, `\x00`, `\xff * 4` • Obfuscated: `1; ls`, `&& whoami`, `| echo owned` • Shutdown: `reb00t`, `shutDOWN`, `REBOOT`, `H@LT`, `halt`, etc.

2.5.2 Automation and Execution

The fuzzing session was fully automated via Python scripts. Boofuzz logged all test cases and responses, and exposed a local web interface to observe session progress. Thousands of combinations were generated and transmitted during this phase.

2.6 LLM-Guided Fuzzing with Boofuzz and GPT4All

To introduce semantic and syntactic diversity in test inputs, Boofuzz was adapted to incorporate GPT4All as a generator of fuzzing payloads. The goal was to assess whether LLM-driven inputs could uncover additional edge cases or faults.

2.6.1 LLM Integration Setup

The GPT4All framework was installed locally with the Nous Hermes 2 Mistral 7B DPO Q4_0 gguf model. It was chosen for its strong instruction-following and code generation abilities, making it well-suited for producing shell-style inputs used in fuzzing the IoTGoat shellback backdoor. The model has been shown to perform well on structured tasks and is commonly used in scripting and shell automation contexts [16]. Similar instruction-tuned models have also demonstrated effectiveness in LLM-guided fuzzing of protocol and grammar-free interfaces [4].This enabled offline LLM access for reproducibility and unrestricted usage. A Python interface was written to query the model at runtime.

2.6.2 LLM-Guided Input Generation and Execution Control

Two input generation approaches were tested to evaluate which style yielded more effective fuzzing payloads:

- **Fixed Prefix + LLM Payload:** A known prefix (e.g., `ping`, `exec`) was combined with a payload generated by GPT4All. The following prompt was used:

  > Generate ONE shell-style payload (no prefix) for a TCP backdoor on port 5515. The service understands simple commands. Known crash string: 'halt'. Use obfuscation, pipes, subshells, nulls, and malformed syntax. Return exactly the payload, no explanation.

- **Fully LLM-Generated Inputs:** The model generated the entire input line, including both prefix and payload. The prompt used was:

  > Generate one test input line for a vulnerable shell service. Format: `<prefix> <payload>\n`. Use shell-style syntax and obfuscation.

These approaches were compared based on output diversity and observed impact on the target. Temperature (0.3–1.2), output length, and duplicate filtering were adjusted to encourage variation across test cases. All inputs were generated at runtime by querying the LLM during each fuzzing cycle.

Boofuzz was used to transmit inputs, manage TCP sessions, and log results. An Arduino-controlled relay was integrated into the test setup to manage the Raspberry Pi's power state automatically. The relay switched the Pi on just before the fuzzing session began, monitored the TCP connection to detect crashes or hangs, and initiated a power cycle whenever the shellback service became unresponsive. The relay was configured to turn the Raspberry Pi off in case the fuzzing comes to a natural end. This setup eliminated the need for manual resets, ensured consistent recovery from crashes, and enabled long-duration, fully autonomous fuzzing campaigns [17].

3 Results

Our goal was to uncover exploitable vulnerabilities in the services exposed by the IoTGoat firmware when deployed on physical hardware. The fuzzing experiments focused on the shellback service on TCP port 5515, previously identified in Section 2.3 as an unauthenticated root shell with no input validation. This section presents the results of testing the IoTGoat device, with a focus on the shellback service running on TCP port 5515. We begin with manual testing, which confirmed the presence of an unauthenticated root shell and established baseline behavior for command handling. We then present results from Boofuzz-based fuzzing, which uncovered multiple crash-inducing inputs and abnormal system behavior. This is followed by LLM-guided fuzzing, where GPT4All-generated inputs were used to replicate or extend the crashes found by Boofuzz. We compare the effectiveness of the Boofuzz and LLM-guided approaches and conclude with a consolidated list of vulnerabilities discovered through the LLM-guided process.

3.1 Manual Test Result of the IoTGoat Backdoor

Before automated fuzzing, manual interaction with the shellback service confirmed its ability to execute arbitrary commands with root privileges and revealed early signs of potential vulnerabilities, such as command injection and persistent filesystem writes (Figure 3.1).

Several shell commands were manually tested to probe the system's behavior and identify vulnerabilities:

- `ping hello`: returned `ping: bad address 'hello'`, indicating shell command forwarding.
- `echo hello`: worked as expected, confirming standard shell output.
- Random strings (e.g., `djbkjgbhkrdbghjzeruihgjkdbfkfb`): returned "`not found`", consistent with shell parsing behavior.
- `get fjhgjhrguvfjfv`, `commands`: also returned "`not found`", confirming lack of custom CLI handler.
- `echo $(id)`: successfully executed and returned `uid=0(root) gid=0(root)`, verifying full root access and shell substitution parsing.
- `crashme %x %x %x %n AAAAA...`: returned `command not found`, but showed signs of format string evaluation when embedded into other inputs.

```
        -p HEXBYTE      Pattern to use for payload
ping hello
ping: bad address 'hello'
echo hello
hello
djbkjgbhkrdbghjzeruihgjkdbfkfb
sh: djbkjgbhkrdbghjzeruihgjkdbfkfb: not found
get fjhgjhrguvfjfv
sh: get: not found
commands
sh: commands: not found
echo `whoami`
sh: whoami: not found

echo $(id)
uid=0(root) gid=0(root)
crashme %x %x %x %n AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
sh: crashme: not found
touch /fuzzed
ls /
%x
adminrootlsecho
adminrootshutdownlsecho
bin
boot
crash%s%x
crash%x%x%x%x%nAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA????1
dev
dnsmasq_setup.sh
etc
fuzzed
fuzzed%x%x%x%x%nAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA????1
lib
mnt
ootrebootshutDOWNsh
otREBOOTReb
overlay
proc
rom
root
sbin
sys
testecho
tmp
usr
utdownshut_downsHuTdOwNhaltH@LTha1thalthalt
var
www
cd root
sh: cd: line 21: can't cd to root: No such file or directory
ls -lrt
-rw-r--r--    1 root     root             5 Jul  7 17:44 testfile
pwd
/root
```

Figure 3.1: Manual testing session showing interaction with the shell-back service on port 5515.

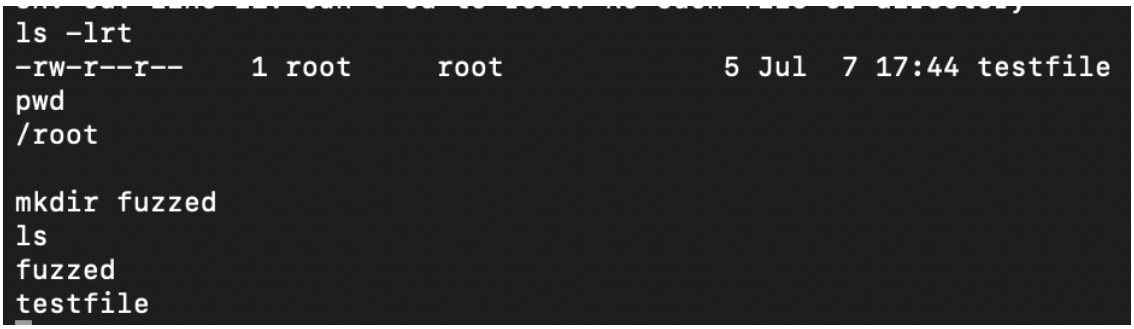

Figure 3.2: Root access through backdoor without authentication

- `mkdir fuzzed`: created a directory in the root directory without any authentication, as seen in Figure 3.2. These suggest prior test strings had been interpreted and possibly persisted.

Furthermore, access to the root directory and ability to write files demonstrated a critical flaw: the backdoor executes all commands as root with no access control or sandboxing. Commands like `1; reboot`, `halt`, and `shutdown` were shown to cause system unresponsiveness in fuzzing sessions (discussed later), and preliminary attempts confirmed that reboot and shutdown commands are available.

These observations support the conclusion that the shellback service:

- Is a fully functional root shell listener with no authentication.
- Accepts and executes arbitrary Linux commands.
- Supports shell metacharacters like `$()`, backticks, and pipes.
- Provides persistent access to the full filesystem.

This manual analysis informed the structure of fuzzing campaigns and provided early confirmation that the service is both exploitable and highly privileged.

3.2 Observed Behaviors on Port 5515 During Boofuzz Fuzzing

Boofuzz systematically generated mutated inputs for the shellback service, reproducing and expanding on the vulnerabilities observed in manual testing, and triggering multiple confirmed service crashes and anomalous filesystem artifacts.

3.2.1 Crash Symptoms and Detection

A typical crash was identified when BooFuzz displayed repeated log entries indicating failure to reconnect to the target, such as: " *Cannot connect to target; retrying. Note: This likely indicates a failure caused by the previous test case... Restarting*

the target process using CallbackMonitor No reset handler available... sleeping for 5 seconds".

These logs (e.g., Figure 3.3) indicated that the backdoor shell process had crashed or become unresponsive, often requiring a full device reboot manually. On the physical connection, the crash was confirmed by the lights on the Raspberry Pi Ethernet port turning off.

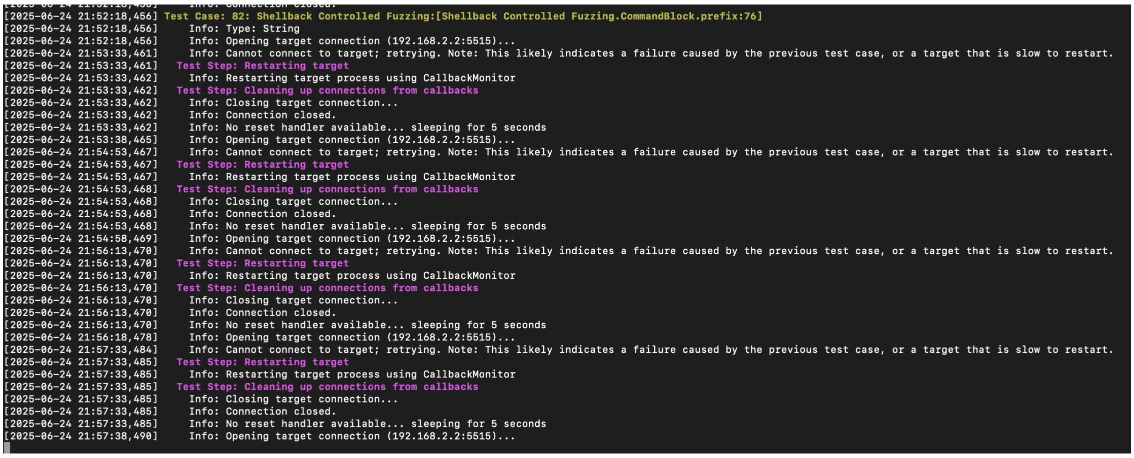

Figure 3.3: BooFuzz repeatedly restarting the target after crash

Crash-Inducing Inputs

Several crafted payloads triggered visible disruption of the shellback service, either resulting in test case hang, disconnection, or creation of anomalous files in the root filesystem (Figure 3.4).

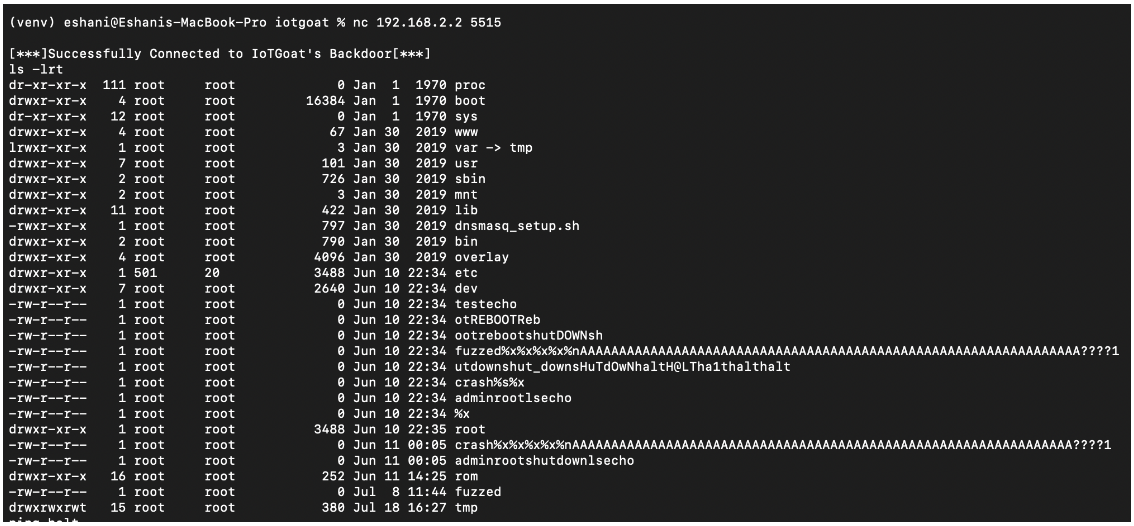

Figure 3.4: Artifacts left on the filesystem after crash-inducing inputs, confirming root command execution and format string evaluation.

Three key crash-inducing inputs were identified:

```
pingpingping...ping adminrootshutdownlsecho crash%x%x%x%x%nAAAAAAAA
    ...AAAAAAAA\x00????1; reboot‘halt‘
```

- Repeated `ping` prefix (likely mutated)
- Shell keywords: `admin`, `root`, `shutdown`, `echo`
- Format string attack: `%x%x%x%x%n`
- Buffer overflow candidate: 64+ `A`'s
- Null byte: `\x00`
- Command injection: `1; reboot`, `` `halt` ``

```
ping??????????????????????????????????????
    adminrootshutdownlsecho crash%x%x%x%x%nAAAAAAAA...AAAAAAAA\x00
    ????1; reboot‘halt‘
```

- Fuzzed or corrupted prefix, replaced with question marks
- Payload structure same as Test Case 1

```
ping;id adminrootshutdownlsecho crash%x%x%x%x%nAAAAAAAA...AAAAAAAA\
    x00????1; reboot‘halt‘
```

- Injected semicolon to separate `ping` from an actual command (`id`)
- Remainder includes same risky components

Crash Behavior Summary

Test Case 3: Semi-valid Command Injection All crash-inducing inputs shared a common structure:

- Command-like prefix (valid or malformed)
- Shell metacharacters or control tokens (`;`, `` ` ``, `\x00`)
- Obfuscated system commands (e.g., `shutdown`, `reboot`, `halt`)
- Format string markers
- Long or malformed padding (e.g., `A`*64)

The Boofuzz framework was able to trigger system hangs, dropped shell connections, creation of persistent malformed files on the filesystem, and multiple command execution scenarios without authentication. Notably, inputs containing explicit shutdown-related commands (e.g., `shutdown`, `reboot`, `halt`) consistently

preceded service crashes, suggesting that execution of these system commands may have directly caused the disruption. As a control, safe commands like `ls /` that produce consistent output did not result in crashes, reinforcing that the observed failures were tied specifically to dangerous shell instructions. Crashes were confirmed and reproducible.

3.3 LLM-Guided Fuzzing Results

Integrating GPT4All into the fuzzing process, we tested two prompting strategies: one where the LLM generated only the payload for a fixed command prefix, and another where it generated both prefix and payload. The fixed-prefix approach reliably reproduced Boofuzz-triggered crashes, while the fully LLM-generated inputs were less consistent; neither approach revealed new vulnerabilities, but both increased input diversity.

1. Fixed Prefix + LLM-Generated Payload In this approach, Boofuzz controlled the command prefix (e.g., `ping`, `exec`, `get`), and GPT4All was prompted to generate the payload segment only. This technique produced the most effective results: several inputs generated in this mode successfully reproduced the same crashes discovered during Boofuzz fuzzing. These inputs consistently demonstrated the same vulnerability traits such as command injection or buffer instability. An example output from this method is shown in Figure 3.5.

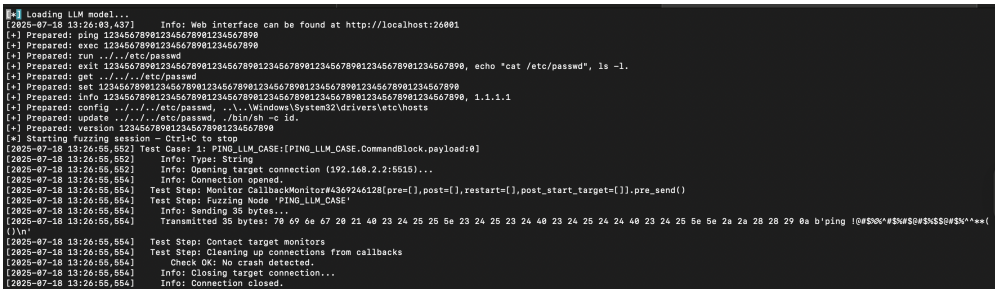

Figure 3.5: Sample test cases generated using the fixed-prefix + LLM-generated payload strategy

2. Fully LLM-Generated Inputs (Prefix + Payload) Here, the LLM was tasked with generating the entire test input line, including both prefix and payload. The prompt format used was:

> Generate one test input line for a vulnerable shell service. Format: `<prefix> <payload>\n`. Use shell-style syntax and obfuscation.

While this method gave the LLM full freedom to generate inputs, the results were inconsistent. Although some outputs followed the intended structure (`<prefix> <payload>`), many were malformed or highly repetitive. As the experiment aimed only to compare structural styles rather than optimize prompts, no advanced prompt engineering techniques were applied. Consequently, this strategy underperformed compared to the fixed-prefix setup, often leading to less diverse or less effective test cases.

Among the two strategies, the fixed-prefix method paired with LLM-generated payloads proved the most productive. It balanced structural control with generative creativity, resulting in test cases that were not only syntactically valid but semantically rich enough to exercise the shellback service. However, no additional vulnerabilities were discovered beyond those already uncovered by Boofuzz. This suggests that while the LLM enhanced coverage and semantic diversity, it did not fundamentally expand the vulnerability surface.

Model Limitations The experiments were conducted using an offline model, Nous-Hermes-2-Mistral-7B-DPO.Q4_0.gguf, executed locally via GPT4All. The diversity and quality of LLM-generated inputs are highly dependent on the underlying model. More advanced or fine-tuned LLMs may produce a broader range of syntax, command structures, or obfuscation styles, potentially increasing the chances of triggering unexpected behavior in the target service. Such models might also generate semantically richer inputs that mimic real-world attacks more closely, thereby improving fuzzing coverage and the likelihood of crash discovery.

3.4 Summary and Comparative Discussion

The shellback service on TCP port 5515 was found to exhibit several critical vulnerabilities. Through a combination of manual probing, Boofuzz mutation-based fuzzing, and LLM-guided input generation, the attack surface of this backdoor was comprehensively mapped. This section summarizes the findings and compares the effectiveness of each approach.

3.4.1 Fuzzing Approaches: Boofuzz vs LLM

This section compares the results obtained from Boofuzz's mutation-based fuzzing with the LLM-guided fuzzing (Fixed Prefix + LLM-generated Payload). The comparison focuses on their ability to reproduce crashes, trigger unique system behaviors, and generate diverse payloads.

Boofuzz Results The Boofuzz framework, using a curated set of prefixes and manually selected risky payloads (e.g., format strings, control characters, shell commands), was able to trigger:

- System hangs and dropped shell connections
- Creation of persistent malformed files on the filesystem
- Multiple command execution scenarios without authentication

Crashes were confirmed and reproducible. Several inputs, such as "*ping???? adminrootshutdownlsecho %x%x%x%x%n AAAAAAAA... \x00 1; reboot*", were responsible for destabilizing the shellback service.

LLM-Guided Fuzzing Results The Fixed Prefix + LLM Payload approach reliably reproduced the crashes identified by Boofuzz. LLM-generated payloads incorporated format strings, null bytes, and shell command chaining, effectively mimicking malicious patterns while producing a higher diversity of variants compared to the original Boofuzz set.

While no additional vulnerabilities were discovered beyond those already found by Boofuzz, the LLM approach demonstrated value as a payload diversification mechanism, which may improve the chances of uncovering edge-case failures in extended fuzzing campaigns.

3.4.2 Observed Shell Behavior

Interactive testing revealed that the shellback service behaves as a permissive, line-based shell interpreter without any authentication. Key behavioral observations include:

- **Command Execution Without Validation**: Any string, including shell metacharacters, is treated as a shell command. Known commands such as `ls` and `echo` executed successfully, while invalid ones (e.g., `randomtext`) returned standard shell errors, confirming that input is passed unfiltered to a BusyBox shell.
- **Lack of Input Sanitization**: Inputs like `shutdown`, `reboot`, or garbage strings were all processed by the shell, with no verification or bounds checking. This allows a wide range of injection vectors.
- **Evidence of Persistent Filesystem Writes**: Files named `adminrootlsecho`, `crash%x%x%x%x%n...`, and `fuzzed` were later discovered in the root directory. These names directly reflect fuzzed payloads, suggesting the backdoor may interpret inputs as redirection targets or write outputs to disk.
- **Format String Injection Success**: File names containing format specifiers indicate that the shell parsed format strings, confirming format string vulnerability

potential even if no memory disclosure was directly observed.

- **Root Access Without Login**: The service provided unrestricted root shell access immediately upon connection, without any password.

These observed behaviors form a baseline catalog of exploitable conditions that can serve as reference failure modes for assessing the effectiveness of future fuzzing tools. Each observation above corresponds to a specific weakness class; Table 3.1 summarizes the mapped vulnerabilities and their CWE identifiers.

Table 3.1: Summary of Vulnerabilities in Shellback Backdoor

Vulnerability	**Observed Through**	**Evidence**	**CWE**
Unauthenticated Shell Access	Manual, All Tools	Access granted via TCP without login; root shell available immediately	CWE-306, CWE-250
Command Injection	Boofuzz, LLM	Inputs like `ping; reboot`, `echo \`halt\`` executed directly	CWE-78
Format String Injection	Boofuzz, LLM	Inputs like `%x%x%x%n` created filenames with format specifiers	CWE-134
Buffer Overflow Potential	Boofuzz, LLM	Payloads with `A*64 + \x00` destabilized the shell or caused hangs	CWE-120
Improper Input Validation (Malformed Prefix Acceptance)	Boofuzz, LLM	Inputs like `pingpingping`, `??? admin` processed without error	CWE-20
Filesystem Injection	Manual, Boofuzz	Files matching payload strings created in `/`, suggesting output redirection or misinterpretation	CWE-73

4 Conclusions and Future Work

This project set out to evaluate whether large language models (LLMs) can enhance fuzz testing on embedded systems. The Shellback backdoor on port 5515 proved to be an ideal case study due to its:

- Lack of authentication
- Direct BusyBox shell command parsing
- Writable persistent filesystem
- Observable behavioral and filesystem changes from fuzz input

The vulnerabilities identified here establish a reproducible set of reference failure modes for evaluating future fuzzing tools. While LLM-guided fuzzing did not uncover new classes of vulnerabilities beyond those found with Boofuzz, it reliably generated structured, crash-inducing variants of known inputs. This suggests that with more advanced or fine-tuned models, grammar-aware exploit generation may further expand vulnerability coverage.

4.1 Future Work

The LLM experiments used a fixed-prefix plus LLM-generated payload strategy, which was the most reliable among tested methods. Future work could explore more capable LLMs (e.g., GPT-4, Claude, LLaMA 3) and alternate prompting strategies such as few-shot or crash-aware prompting to produce more diverse and targeted inputs.

Due to time constraints, other open ports (e.g., DNS on 53, HTTP on 80/443, Telnet on 65534) were not fuzzed. Expanding coverage to these services could reveal additional vulnerabilities and strengthen IoTGoat's value as a comprehensive embedded fuzzing benchmark.

In summary, broader fuzzing coverage, refined prompting, and integration of advanced LLMs could make LLM-guided fuzzing a robust approach for embedded security testing.

References

[1] V. J. M. Manès, H. Han, C. Han, S. Lee, I. Yun, Y. Jang, W. Lee, and T. Kim, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[2] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Smart greybox fuzzing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, 2018, pp. 1032–1043.

[3] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

[4] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proc. Network and Distributed System Security Symposium (NDSS)*, 2024.

[5] W. Guo *et al.*, "Can large language models generate fuzzing inputs?" in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2023.

[6] Y. Xia, Y. Wang, Y. Zhou, Z. Xu, L. Lu, and Y. Jiang, "Fuzz4all: Universal fuzzing with large language models," in *Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE)*, 2024.

[7] Y. Li, B. Chen *et al.*, "Iotfuzzbench: Towards a benchmark for iot fuzzing," *Electronics*, vol. 12, no. 14, p. 3010, 2023.

[8] R. Natella and V. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2021, pp. 662–665.

[9] Y. Cheng, X. Li, Z. Mao, W. Fan, W. Huang, and W. Liu, "IoTBenchSL: A Streamlined Framework for the Efficient Production of Standardized IoT Benchmarks with Automated Pipeline Validation," *Electronics*, vol. 14, no. 5, p. 856, 2025. [Online]. Available: https://doi.org/10.3390/electronics14050856

[10] OWASP, "Iotgoat: A deliberately insecure iot firmware," 2023, available: https://owasp.org/www-project-iot-goat/.

[11] J. Riser and Contributors, "Boofuzz: Network protocol fuzzing for humans," 2023, available: https://github.com/jtpereyda/boofuzz.

[12] T. B. Brown, B. Mann, N. Ryder, and et al., "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[13] M. Chen, J. Tworek, H. Jun, and et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/pdf/2107.03374.pdf

[14] Nomic AI, "Gpt4all: Local language models and documentation," https://docs.gpt4all.io/, 2023, accessed: 2025-07-21.

[15] OWASP, "Iotgoat releases - github," https://github.com/owasp/iotgoat/releases, 2025, accessed: 2025-07-21.

[16] N. Research, "Nous hermes 2 mistral 7b dpo," https://huggingface.co/NousResearch/Nous-Hermes-2-Mistral-7B-DPO, 2024, accessed August 2025.

[17] E. Nandy, "LLM-Guided-Fuzzing: Automation and Fuzzing Scripts for IoTGoat Shellback," https://github.com/eshaninandy/LLM-Guided-Fuzzing, Aug. 2025, accessed: 2025-08-09.