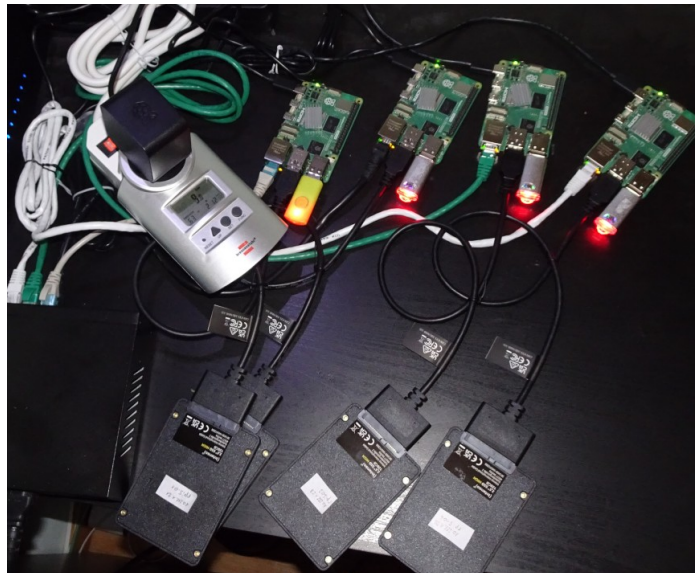


FUZZING ANDROID APPLICATIONS : SCALABLE INFRASTRUCTURE FOR AUTOMATED SECURITY ANALYSIS



Bachelor's Thesis presented by

Stefan ANTUN

for the obtention of the degree Bachelor of Science HES-SO in

**Computer Science and Communication Systems specialized in
Computer Security**

September 2025

Professor HES in charge

Roland Sako Sekou

TABLE OF CONTENTS

Acknowledgments.....	vi
subject Statement.....	vii
Summary.....	viii
Acronyms List.....	ix
Table of Figures.....	xi
Table List.....	xii
Appendices List.....	xiii
Introduction.....	1
1. Chapter 1 : State of the Art.....	3
1.1. What is Fuzzing ?.....	3
1.2. Fundamentals of Fuzzing.....	4
a) Application Fuzzing.....	4
b) Protocol Fuzzing.....	4
c) File Format Fuzzing.....	4
1.3. Fuzzing Approaches.....	5
a) White-box Testing.....	5
b) Grey-box Testing.....	5
c) Black-box Testing.....	5
1.4. White-box vs Black-box Fuzzing.....	5
1.5. Symbolic vs Concolic Fuzzing.....	7
1.6. Fuzzing Methodologies.....	7
a) Random/Mutation-based.....	7
b) Generation-based.....	8
c) Evolution-based.....	8
1.7. Virtualization.....	8
1.8. Security Needs.....	9
a) CIA Triad.....	9
b) Non-Repudiation.....	9
c) State Management.....	9
d) Zero Trust.....	10

e) Risk Management.....	10
2. Challenges.....	11
2.1. Android.....	11
a) ARM64 vs x86_64.....	11
b) Bionic libc vs glibc.....	11
c) Memory Management and Patched Linux Kernel.....	11
d) Hardware Attestation.....	12
2.2. Long Running Fuzzing Sessions.....	12
2.3. Power Users.....	12
2.4. Unknown Unknowns.....	12
3. Hardware Platform.....	13
3.1. Fuzzing in the Cloud – No Go.....	13
3.2. Raspberry Pi 5.....	13
3.3. Storage.....	15
a) NAS – No Go.....	15
b) GlusterFS.....	16
4. Software Platform.....	17
4.1. AFL++.....	17
4.2. GNU/Linux.....	17
4.3. Podman Containers.....	17
4.4. Android Cuttlefish.....	18
4.5. SSH.....	18
4.6. Wireguard.....	19
5. Security.....	20
5.1. CIA Triad.....	20
5.2. Non-Repudiation.....	22
5.3. State Management.....	23
a) Podman.....	23
b) Ansible.....	24
5.4. User Management and Access Control.....	24
5.5. Network Access.....	25
5.6. Monitoring and Alerting.....	25
6. Virtualization Topology.....	26
7. Network Topology.....	28

8. Deployment Workflow.....	32
8.1. Sysadmin.....	32
a) Time to Deploy.....	33
8.2. Student.....	35
a) Time to Deploy.....	35
8.3. Student White Box Fuzzing Workflow.....	36
8.4. Student Black Box Fuzzing Workflow.....	36
9. Fuzzing Performance.....	37
9.1. Our Android Cluster.....	37
9.2. RPi 5 but GNU/Linux.....	38
9.3. Hetzner Cloud.....	39
9.4. Commentary on Results.....	40
10. Difficulties.....	41
10.1. RPi Troubles.....	41
10.2. Podman Troubles.....	41
10.3. Wireguard Troubles.....	42
10.4. Android Troubles.....	42
10.5. GlusterFS Troubles.....	42
10.6. NDK Troubles.....	42
10.7. AFL Troubles.....	43
10.8. GDB Troubles.....	43
10.9. Ghidra Troubles.....	43
11. License.....	44
12. Areas for Improvement.....	45
13. Conclusions.....	46
Appendices.....	48
Bibliography.....	56

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to everyone who contributed to the completion of this bachelor's thesis.

First and foremost, I extend my sincere thanks to my supervisor, Roland Sako Sekou, for his guidance, expertise, and unwavering support throughout this project. His insights and feedback were invaluable in shaping the direction and quality of this work.

I am also deeply grateful to the faculty and staff of the ISC program at HES-SO for providing the academic environment, resources, and encouragement that made this research possible.

A heartfelt thank you to the global open-source community—to the developers, contributors, and educators who freely share their tools, tutorials, and knowledge online. Without their dedication to creating and maintaining open resources, this project would not have been feasible.

Lastly, I wish to thank my family and friends for their patience, understanding, and constant encouragement. Their support was a steady source of inspiration and strength during this demanding journey.

**FUZZING ANDROID APPLICATIONS: SCALABLE INFRASTRUCTURE FOR AUTOMATED
SECURITY ANALYSIS****ORIENTATION : COMPUTER SECURITY**

Description: This work consists of designing and deploying a scalable fuzzing infrastructure for the security analysis of Android applications, with a particular focus on native components (C/C++ libraries) running on ARM architecture. The main objective is the automated identification of vulnerabilities in these components, in different scenarios depending on whether the source code is available or not. The project addresses a need expressed by a pentesting team wishing to industrialize its tests, while also enabling reuse in an academic setting (Android labs for cybersecurity courses). The emphasis is on automation, orchestration, and documentation.

Work required :

- Study of fuzzing tools adapted to Android ARM (AFL++, libFuzzer, honggfuzz, etc.)
- Comparison of white-box and black-box approaches depending on source code availability
- Deployment of a scalable environment (AVD, Docker, Proxmox, etc.)
- Integration of fuzzing tools and automation of test campaigns
- Creation of a reusable guide for Android cybersecurity labs

Candidate :

ANTUN STEFAN

Study Program : ISC

Professor in charge :

SAKO SEKOU ROLANDBachelor's thesis subject to an internship agreement in a
company: noBachelor's thesis subject to a confidentiality
agreement: no

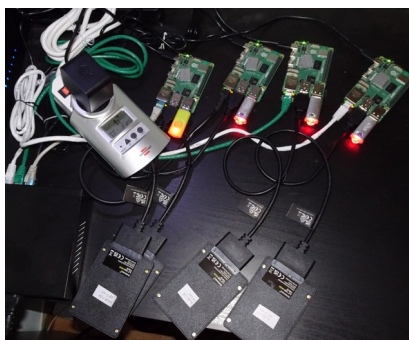
SUMMARY

This bachelor's thesis presents a scalable infrastructure for automated fuzzing of Android applications, focusing on native ARM64 components to address security testing challenges in both industrial pentesting and academic settings. Android's unique architecture—combining Bionic libc, a modified Linux kernel, and ARM64—complicates traditional fuzzing, particularly when source code is unavailable. The proposed solution integrates AFL++ with Android Cuttlefish and rootless Podman containers, deployed on a Raspberry Pi 5 cluster to enable distributed white-box and black-box vulnerability analysis.

The infrastructure achieves 83,101 executions per second across four nodes, though performance benchmarks reveal a 79% slowdown compared to GNU/Linux due to poor optimization. Automation via Ansible reduces deployment time to around four hours for administrators and around 40 minutes for end users, while WireGuard VPN and GlusterFS ensure secure, resilient networking and storage. Custom patches for AFL++, GDB, and the Android NDK resolve ARM64 compatibility issues, and comprehensive documentation supports reuse in cybersecurity labs, including workflows for tools like JADX, Ghidra, and Frida.

Key challenges—such as the Raspberry Pi 5's lack of nested virtualization—were addressed through containerized Android VMs. The system's cost efficiency (CHF 700 for four Pis, 9W per node) and modular design make it accessible for research and education, despite trade-offs like limited non-repudiation.

Future improvements could target performance optimizations (e.g., Cuttlefish snapshots or cloud hybrids), centralized user management (LDAP), or AI-driven fuzzing techniques. By open-sourcing the framework, this work provides a practical, reproducible foundation for Android security analysis, lowering barriers for researchers and professionals alike.



Candidate :

ANTUN STEFAN

Study Program : ISC

Professor in charge :

SAKO SEKOU ROLAND

Bachelor's thesis subject to an internship agreement in a company : no

Bachelor's thesis subject to a confidentiality agreement : no

ACRONYMS LIST

IaC	Infrastructure as Code
VM	Virtual Machine
ROM	Read-Only Memory
KVM	Kernel-based Virtual Machine
AFL++	American Fuzzy Lop Plus Plus
CVD	Cuttlefish Virtual Device
RAM	Random Access Memory
CPU	Central Processing Unit
GNU	GNU's Not Unix
MITM	Man-In-The-Middle
CA	Certificate Authority
IP	Internet Protocol
GDB	GNU Debugger
NDK	Native Development Kit
GPL	GNU General Public License
RPi	Raspberry Pi
IPC	Instructions Per Cycle / Inter-Process Communication
VPS	Virtual Private Server
ADB	Android Debug Bridge
API	Application Programming Interface
SDK	Software Development Kit
SSH	Secure Shell
TLS	Transport Layer Security
SSL	Secure Sockets Layer
DNS	Domain Name System
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
OS	Operating System
CLI	Command Line Interface

GUI	Graphical User Interface
VPN	Virtual Private Network
IoT	Internet of Things
LXC	Linux Containers
CIA	Confidentiality, Integrity, Availability
SSD	Solid State Drive
HDD	Hard Disk Drive
LLVM	Low Level Virtual Machine
FS	File System
LDAP	Lightweight Directory Access Protocol
CMDB	Configuration Management Database

TABLE OF FIGURES

Figure 1: Virtualization Topology - Control.....	26
Figure 2: Virtualization Topology - Execution Flow.....	27
Figure 3: Network Topology - Control.....	28
Figure 4: Network Topology - GlusterFS.....	29
Figure 5: Network Topology - Container to WAN.....	30
Figure 6: Screenshot of distributed fuzzing results on our Android cluster.....	37
Figure 7: Screenshot of fuzzing results on RPi 5.....	38
Figure 8: Screenshot of fuzzing results on VPS.....	39

TABLE LIST

Table 1: Virtualization Tools.....8

APPENDICES LIST

Appendix 1 – Simple Fuzzing Code.....	49
Appendix 2 – GDB Debugging Aid.....	50
Appendix 3 – Custom Shared Memory Implementation.....	51

INTRODUCTION

The proliferation of Android applications—now exceeding 1.5 million on the Google Play Store alone—has made the platform a prime target for cyberattacks. Many of these applications rely on native components (C/C++ libraries) for performance-critical tasks, exposing them to vulnerabilities such as memory corruption, buffer overflows, and logic flaws. Traditional security testing methods, while effective, often struggle to keep pace with the scale and complexity of modern Android apps, especially when source code is unavailable or when testing must be automated for industrial or academic use.

Fuzzing has emerged as a powerful technique for uncovering hidden vulnerabilities by bombarding software with malformed inputs. However, applying fuzzing to Android presents unique challenges: the ARM64 architecture, the use of Bionic libc instead of glibc, and the platform’s heavily modified Linux kernel all complicate the deployment of standard fuzzing tools. Additionally, the need for scalable, resilient infrastructure—capable of handling long-running test campaigns and supporting both white-box and black-box approaches—remains largely unmet in existing solutions.

This thesis addresses these challenges by designing a scalable fuzzing infrastructure for Android applications, focusing on native components. The solution integrates open-source tools (AFL++, Podman, Android Cuttlefish) to create a flexible, automated environment for both white-box and black-box fuzzing.

This project presented a steep learning curve, as fuzzing techniques and Android internals were not covered in the curriculum. The work required self-directed study of advanced topics, including ARM64 reverse engineering, distributed fuzzing architectures, and Android virtualization—skills developed iteratively through experimentation and open-source community resources.

All code and documentation are publicly available at <https://githopia.hesge.ch/stefan.antun/travailbachelor2025> and <https://github.com/ByteBrigand/Bachelor-IT-HEPIA-2025>.

Key contributions include:

- A comparative analysis of fuzzing approaches (white-box, grey-box, black-box) and their suitability for Android’s ARM64 ecosystem.
- A distributed fuzzing architecture leveraging Raspberry Pi 5 clusters, GlusterFS for shared storage, and WireGuard for secure network isolation.
- Automation scripts and documentation to facilitate reuse in academic labs and professional pentesting workflows.
- Performance benchmarks comparing fuzzing efficiency on Android virtual devices, GNU/Linux, and cloud-based ARM64 instances.

By bridging the gap between theoretical fuzzing techniques and practical deployment on Android, this work aims to provide security researchers and educators with a robust, cost-effective toolkit for identifying vulnerabilities in real-world applications.

1. CHAPTER 1 : STATE OF THE ART

1.1. WHAT IS FUZZING ?

What is fuzzing in the context of IT security? The Oxford English Dictionary has no such definition for our uses. Other dictionaries and Thesauruses neither. It's a rather new, fuzzy word and we don't know for sure who coined it. There are articles and blog posts on the world wide web trying to specify what it is, but there's still some fuzz around it.

A page [22] on gitlab.com offers this definition:

Fuzz testing, or application fuzzing, is a software testing technique that allows teams to discover security vulnerabilities or bugs in the source code of software applications. Unlike traditional software testing methodologies – SAST, DAST, or IAST – fuzzing essentially “pings” code with random inputs in an effort to crash it and thus identify faults that would otherwise not be apparent.

A page [13] on sciencedirect.com offers another definition :

Fuzzing is a process in computer science where random data is passed to an application to detect anomalies, particularly in parts that accept user input, which can help identify vulnerabilities such as buffer overflows.

An article [14] on owasp.org offers this definition :

Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion.

We find yet another definition on mitre.org [8] :

Fuzzing is a software security and functionality testing method that feeds randomly constructed input to the system and looks for an indication that a failure in response to that input has occurred.

I'll throw in my definition for good measure :

Fuzzing is the art of producing unexpected behavior in a program by feeding it semi-random data.

The process of fuzzing seems to date to Barton Miller, when electric noise on a phone line connected to a Unix system caused random inputs which caused programs to crash, which led him to develop systematic program tests by feeding them with random inputs. He described this in his 1990 paper [3].

Another project [17] of his from 1988 cites an assignment to create a fuzzing tool.

While he uses the terms « fuzz » and « fuzz generator » he never used it as a verb, in the likes of « fuzzing » or « to fuzz », at least not in the two articles cited above that most cite as the source of the verb « fuzzing ». Lets give him the benefit of the doubt.

1.2. FUNDAMENTALS OF FUZZING

A) APPLICATION FUZZING

Targets user-facing interfaces: APIs, CLIs, GUIs, or libraries. The goal is to inject malformed, unexpected, or edge-case inputs to expose crashes, memory leaks, or logic flaws.

The main challenge is bypassing input validation, sandboxing, and handling stateful applications like databases.

B) PROTOCOL FUZZING

Tests network protocols by sending malformed or unexpected packets to servers/clients. Targets include HTTP, TLS, DNS, and custom protocols. Effective for finding parsing errors, buffer overflows, and logic flaws in protocol implementations.

C) FILE FORMAT FUZZING

Targets applications that parse files (e.g., PDF, JPEG, DOCX). Malformed files are generated to trigger crashes or memory corruption. Often used to find vulnerabilities in parsers, libraries, or media players.

1.3. FUZZING APPROACHES

A) WHITE-BOX TESTING

White-box fuzzing leverages full access to the target’s source code and internal logic. This allows for precise input generation, symbolic execution, and systematic exploration of execution paths. By analyzing the code structure, white-box fuzzers can infer input constraints, optimize test cases, and even solve path conditions to reach deep or hidden vulnerabilities.

B) GREY-BOX TESTING

Grey-box fuzzing combines elements of both white-box and black-box methods by using runtime feedback, such as code coverage, to guide input generation. This approach dynamically adapts to the target’s behavior without requiring full source code access.

C) BLACK-BOX TESTING

Black-box fuzzing tests the target without any internal knowledge, relying on random, mutated, or model-based inputs to provoke crashes or unexpected behavior. It is simple to implement and works on any executable, including closed-source or binary-only software. While easy to deploy, it is less efficient than grey-box or white-box methods, as it lacks insight into the target’s structure and may miss subtle or complex bugs.

1.4. WHITE-BOX VS BLACK-BOX FUZZING

White-box programs are generally easier to build a harness for because their variable types are known, variables are named, and comments often clarify the code’s intent. This makes it simpler to infer the kind of input data required. However, white-box fuzzing faces significant challenges when it comes to compilation. Programs may have numerous dependencies, all of which must be compiled as well, and the process can be complex due to the sheer number of compilation flags. For example, GDB’s makefile exceeds 60,000 lines, and many bugs only surface in specific build configurations. In this regard, black-box fuzzing holds a clear advantage: the code is already compiled and in its final state, so only the harness needs to be built.

Analyzing compiled programs is feasible with tools like the NSA’s Ghidra, and large language models (LLMs) are now being trained for decompilation (e.g., LLM4Decompile [23]), though their effectiveness remains untested by us. The main difficulty lies in the fact that CPUs are type-agnostic and they store data as binary, and type information is only determined at compile time. As a result, reverse engineers must rely on intuition and experience to deduce variable types and their contents.

The situation becomes even more complex with packed structs, where multiple variables are stored in a single memory block for performance reasons. For instance, an 8-bit variable might split into 3 bits for one field and 5 bits for another, making it difficult to decipher the structure and meaning of the data.

To mitigate these challenges, we can instrument a program during runtime, observing register states at each instruction. This helps infer variable types and their contents, ultimately enabling the construction of a harness for fuzzing.

1.5. *SYMBOLIC VS CONCOLIC FUZZING*

Symbolic Fuzzing attempts to explore all possible execution paths in a program to find crashes. While powerful, it suffers from path explosion, which is the exponential growth of paths in complex programs, which can render the approach impractical. For simpler programs with few loops, symbolic fuzzing can quickly uncover most or all bugs. For example:

```
def check_num(x):
    if x > 10:           # Path condition 1
        if x < 20:       # Path condition 2
            crash()      # Target
```

Here, symbolic execution reveals that any value between 11 and 19 (inclusive) triggers the crash. Without this insight, a traditional fuzzer might waste time testing millions of inputs, unaware that only nine values cause the crash.

Concolic Fuzzing combines symbolic execution with concrete values to guide semi-random fuzzing. This hybrid approach helps manage path explosion by solving path constraints more efficiently. For example:

```
def check_num(x):
    y = x * x
    if y > 100:           # Path condition 1
        if x < 20:       # Path condition 2
            crash()      # Target
```

While the quadratic operation complicates pure symbolic execution, concolic fuzzing uses concrete values to evaluate y , making it easier to solve the constraints and identify the same crashing inputs (11 to 19).

1.6. *FUZZING METHODOLOGIES*

A) *RANDOM/MUTATION-BASED*

Starts with valid inputs (seeds) and mutates them (bit flips, deletions, insertions). Simple to implement, but may miss complex bugs due to lack of structure awareness.

B) GENERATION-BASED

Creates inputs from scratch using models of the expected format (e.g., grammar rules for file formats or protocol messages). More precise, but requires detailed knowledge of the target.

C) EVOLUTION-BASED

Uses genetic algorithms to evolve inputs over time, favoring those that trigger new code paths or crashes. Combines mutation and generation, often guided by feedback (e.g., code coverage).

1.7. VIRTUALIZATION

Virtualization abstracts hardware, allowing multiple OS instances (VMs) to run on a single physical machine.

There are two main types of hypervisors. A type 1 (bare-metal) hypervisor runs directly on hardware (e.g., VMware ESXi, Hyper-V, KVM) while a type 2 (hosted) runs on top of an OS (e.g., VirtualBox, QEMU).

Modern CPUs accelerate virtualization by offloading tasks (e.g., memory management, I/O) to hardware, improving performance.

Emulation simulates hardware/software not natively supported (e.g., QEMU emulating ARM on x86). Slower than virtualization but enables cross-platform testing.

Tool	Type	Use Case
KVM	Type 1	Linux kernel-based virtualization
QEMU	Type 2/Emulator	Cross-platform emulation
Proxmox	Type 1	Enterprise virtualization
Hyper-V	Type 1	Windows virtualization
VMware	Type 1/2	Enterprise/desktop virtualization

Table 1: Virtualization Tools

Containerization isolates applications and their dependencies in lightweight, portable containers (e.g., Docker, LXC). Shares the host OS kernel, unlike VMs. Faster startup, lower overhead, but weaker isolation.

1.8. SECURITY NEEDS

At the time of writing, infrastructure security is mostly guided by five principles :

A) CIA TRIAD

The CIA Triad is made of three pillars :

- Confidentiality: Ensuring that sensitive data is accessed only by authorized individuals or systems.
- Integrity: Ensuring data remains accurate, consistent, and unaltered during transmission or storage.
- Availability: Ensuring data and network resources are accessible to authorized users when needed.

B) NON-REPUDIATION

Non-Repudiation is the security concept where every action can be attributed to someone. This requires logging every action, logging who performed the action, and making sure the logs can't be altered or deleted.

C) STATE MANAGEMENT

State is the complete description of a system at a given point in time. A backup is description of the state of the system at a past point in time. The current state of the art is to describe the state of the system entirely in code. Tools that help achieve this are mostly shell scripts, containerization technologies like Docker and orchestration solutions like Ansible. Whenever someone applies a change to a system that is undocumented, the system is said to be in unknown state. When deploying Infrastructure-as-Code tools it is therefore primordial to stick to them, to only apply changes to the code and never directly to the machine, which would negate the benefit of IaC and force the use of backups.

D) *ZERO TRUST*

Zero Trust is the security concept of not trusting the internal perimeter of a network and assuming that malicious actors have already gotten inside. It means that one has to perform strong authentication and authorization even when inside of a network. It means encrypting internal communications.

E) *RISK MANAGEMENT*

Risk management is a most fundamental concept of computer security. It involves defining threats, whether human or not, assigning a degree of importance and priority, and managing them.

2. CHALLENGES

2.1. ANDROID

A) ARM64 vs x86_64

The first major challenge is that Android operates with an ARM64 instruction set.

Most major tools at our disposal only support x86_64, or AMD64, which requires modifying and adapting them to work on ARM64.

While there is Android for x86_64 for development, given most developers have an Intel or AMD workstation allowing them to virtualize it without emulation, it is not sufficient for our purposes because the same code compiled for ARM64 will have bugs that are not present in x86_64 and viceversa.

B) BIONIC LIBC VS GLIBC

Android uses Bionic libc libraries instead of the popular glibc on GNU/Linux.

This complicates compilation toolchains and means that some libraries won't be available. They may also produce different fuzzing behavior if a program makes calls to Bionic libraries. Which is going to be virtually all cases unless a program has no input and no output.

C) MEMORY MANAGEMENT AND PATCHED LINUX KERNEL

Android has a different memory allocation strategy and a heavily patched Linux kernel. This forces us to run the fuzzing inside an Android OS, given vulnerabilities that manifest on Android may not manifest on GNU/Linux.

D) **HARDWARE ATTESTATION**

The boot process requires an Android-specific hardware attestation which is poorly documented. Meaning one may have difficulty running Android as a VM and may face challenges emulating the hardware attestation.

Moreover, far too many financial apps check the hardware attestation with Google and refuse to run if not on « official » hardware, notably payment solutions like Google Pay. Even non financial apps are joining in on the lockdown, like the one from McDonalds [4] for placing a food order... who's lovin' that ?

2.2. LONG RUNNING FUZZING SESSIONS

Fuzzing sessions can span weeks if not months. While some tools like AFL++ have resume-ability, others do not. Availability and resilience is of utmost importance. There's the challenge of implementing countermeasures against hardware failures, as well as what to do in case a security patch demands a reboot.

2.3. POWER USERS

Given the system is destined to be used by security researchers, the system will have to be extra secure to contain them.

2.4. UNKNOWN UNKNOWNNS

Given the complexity of the project and the sheer quantity of unknown unknowns, which we don't really know how many but we assume too many, we'll have to proceed in iterative fashion, not knowing if our plans would face significant setbacks and whether we'll have to pivot to something else.

3. HARDWARE PLATFORM

3.1. FUZZING IN THE CLOUD – No Go

We tried renting cloud servers and quickly discovered that cloud providers don't enable nested virtualization. We tried Android Docker containers but those didn't work either. We opted to buy ARM64 hardware.

3.2. RASPBERRY PI 5

The Raspberry Pi 5 was chosen as the hardware platform because it has an ARM64 architecture, the same architecture as Android smartphones, allowing us to fuzz binaries in black box fashion without resorting to emulation which would have delivered catatonic performance.

We didn't want to use Android phones for fuzzing mostly because of performance and cost. A smartphone does not have an exposed IC onto which we can attach a heat sink and a cooling fan. Old used smartphones don't support the newest Android version, and those that do, are not guaranteed to keep supporting them. Therefore sooner or later we'd reach a point where we'd need to replace the smartphones. Another problem is the supply availability and cost. A RPi 5 with 16 GB of RAM costs around CHF 120, pushing CHF 160 with accessories. A smartphone of similar power (on paper) would cost CHF 360, like the Google Pixel 7a, at the time of writing. On paper because that power could not be sustained for long. Given fuzzing uses all cores at all times, the phone would quickly have to scale down its frequency to not overheat, given the CPU is enclosed, has no heat sink and no fan. Not having a phone limits the things that can be fuzzed, like the broadband modem, or the camera. Although these are vendor specific and out of scope of this project.

We could have sourced old used phones, but we'd still have to pay at least CHF 100 for each, they would not perform as well, given they too are enclosed, and we would not have been able to source the same make and model in large quantities on the used market. Therefore we'd end up with phones of different manufacture, for which a custom Android ROM would have had to be

built, each having a different hardware and bootloader. Some require contacting the manufacturer and waiting months for them to unlock the bootloader, if they are even willing to do so.

Another strong contender was the Ampere Altra platform. We didn't pick it on the basis of budget. At the time of writing, mifcom.ch sells an Ampere Altra Q64-22 workstation with a 64x 2.2 GHz processor for CHF 2999 [1].

We'd have to buy at least four for clustering, for a total of ~12k CHF.

Why a minimum of four? Because three is the minimum to avoid the split-brain issue, and four to allow for availability. If we bought only three, the entire cluster would have to be stopped when one node goes down, because only two hosts would remain and it'd be impossible to know who's in the right when a conflict (split-brain) ensues. Four is not ideal either because there's a risk that a network problem disconnects two from the other two, and when they reconnect they'd have a split-brain issue. This feels like a rare enough problem to take the risk and therefore we opted for four nodes.

Having to spend only CHF ~700 for the four RPi 5s allows us to fuzz on ARM64 with less capital outlay.

Unfortunately the RPi 5 doesn't have IPMI or other remote management solution, unlike the Ampere Altra offering.

In terms of performance per Watt, Jeff Geerling had done some benchmarks and found 4.7 Gflops/W for the Ampere Altra Q64-22 [5] and 2.8 Gflops/W for the RPi 5 [6]. So it seems, at least on paper, that long term fuzzing on the Ampere Altra is more efficient in terms of electricity cost. Whether Gflops performance translates to fuzzing performance is not known to us and we don't have an Ampere Altra to benchmark against.

Even though RPi 5 is specified as GICv4 [21], after buying it we discovered it does not support nested virtualization, or we couldn't get it to work. Passing the 'kvm-arm.mode' [21] parameter to the kernel did not enable it.

An attempt to build a custom kernel for the RPi 5 was made, but there was no flag for nested virtualization, and modifying the kernel's source code (or even the firmware) is out of scope of this project, given how time consuming that would be and the risk of finding out that the RPi 5 can't fundamentally do nested virtualization with hardware acceleration.

A benchmark was performed to test nested virtualization performance on the RPi 5 using the sysbench utility.

Running "sysbench cpu --threads=1 run" on the host, 2738 events per second were measured.

In a VM the same benchmark resulted in 2728 events per second.

In a VM inside a VM : 439 events per second.

Therefore Android has to run as the first level of VM, which greatly limits our options for the infrastructure.

3.3. STORAGE

A) NAS – No Go

A single NAS with network boot on the Pis would of prejudiced availability, given the NAS would of become the single point of failure.

Having a redundant storage system, separate from the RPi 5s, would of obliterated any cost, time and complexity savings given how few Pis we bought.

Given Ampere Altra seems to be more efficient and much more powerful, it would be preferable to buy a few Ampere Altra servers than to buy hundreds of RPi 5s, if one is to deploy a powerful cluster. Which would still keep the number of hosts relatively low.

It seems unlikely at this time that the university would buy 10 or more Ampere Altra servers in the near future, which could benefit from network boot and iSCSI storage purely for cost savings once such a number of hosts is hit. Given fuzzing does not do synchronous database operations, a Fibre Channel will not be needed. Therefore centralized redundant storage is of low priority at this time and will not be implemented.

Therefore each host gets its own SSD over a cheap USB adapter. We added a SATA SSD over a USB to SATA adapter to each Pi, because performance and durability of SD cards is abysmal. Unfortunately most cheap adapters don't support the TRIM command but it's a compromise worth making at this stage. We could of bought a SATA or NVMe hat for the RPi but the USB adapter was chosen on budget considerations.

B) GLUSTERFS

We opted to use GlusterFS as the shared storage solution. It creates a kind of striped disk across many hosts. It has a replica system where files are kept on multiple machines for fault tolerance. We opted for a replica value of two. If a disk becomes unavailable, a copy of its data is present on other machines, which is automatically redistributed to keep the same replica level. While a replica of three would have allowed two failures at the same time, we find that for the time being a replica of two should provide adequate availability.

4. SOFTWARE PLATFORM

4.1. AFL++

AFL++ was chosen as the fuzzing platform because it can do distributed fuzzing on its own, simply by having a shared directory between all hosts. It also comes with Frida mode, useful for black box fuzzing. It also comes with many plugins incorporating other fuzzers, like libFuzzer.

4.2. GNU/LINUX

Given Android runs on a modified Linux kernel, given Raspberry provides support for the Pi with a modified Debian distribution, given AFL++ was developed to be run on GNU/Linux, and given time is of the essence, it was decided to stay on GNU/Linux, rather than trying ARM64 Windows, macOS, *BSD, ChromeOS or something else.

An unsuccessful attempt was made to run Android directly on the RPi 5. Unfortunately stock Android demands security features that the RPi does not provide and modifying Android is out of scope of this project.

The same security features made it daunting to run Android through QEMU. Security features which we don't fully understand, but involve some kind of hardware attestation.

4.3. PODMAN CONTAINERS

Given a fuzzing session could span months, it's important to minimize disruptions like reboots. This is achieved by containerization, network isolation and having the least amount of software possible on the host platform, minimizing attack surface and therefore the likelihood that a security vulnerability forces us to update and reboot.

Initially the idea was to deploy a virtualization solution like Proxmox, which would have made it very easy to manage the infrastructure. We spent considerable time getting Proxmox to work on the RPi 5. But the lack of nested virtualization and QEMU support for Android torpedoed the idea.

A project called start-avm [19] seemed like the perfect solution, which allows running Android through QEMU and provides the security features that Android requires, emulating hardware attestation. The repository is only 2 years old and no longer works with the most recent version of Android, version 15. An unsuccessful attempt was made to modify start-avm to get it to work with Android 15.

A considerable advantage of using podman containers is the build checkpointing system. A checkpoint is made after each command, allowing one to apply a modification without having to rebuild the entire container. This makes development much quicker than on hardware.

Another advantage of a Podman container versus a Docker container is that the former is rootless by default, while the latter is meant to be run as root. This aids us in separating users and applying the principle of least privilege.

4.4. ANDROID CUTTLEFISH

A breakthrough was made with the discovery of Android Cuttlefish, a virtualization solution that uses CVD (Cuttlefish Virtual Devices) and leverages KVM.

CVD code is shipped with AOSP and is built into a ready-to-deploy solution when using "make distrib".

It took some work to get it to work in an unprivileged podman container by manually creating the network interfaces it needed and we were successful. Therefore we'll use rootless podman containers, one for each user, where they run their own Android VMs by leveraging Android Cuttlefish and CVD.

4.5. SSH

The idea was to have users connect through SSH to their container and port forward through SSH whatever was needed to their workstations.

The problem with that is that Android Cuttlefish uses WebRTC, and SSH port forward does not play well with UDP, and it proved unreliable. In the past it used to work over VNC, but it seems they dropped that in favor of a web interface which works only on Chromium and derivatives! Implementing a custom proxy to forward WebRTC packets over SSH didn't seem like a good idea. We'll still use SSH but not for forwarding traffic.

4.6. WIREGUARD

Wireguard VPN was chosen because it forwards all traffic, it's easy to setup and works over UDP, avoiding the TCP-in-TCP ACK amplification problem. It's also included in the linux kernel, is very performant, low overhead and has fewer lines of code, compared to other VPNs like OpenVPN, which is favorable for auditability [18], [3].

5. SECURITY

5.1. CIA TRIAD

To guarantee confidentiality, we'll encrypt data at rest and in transit. At rest by encrypting the disks with LUKS, and in-transit by using a Wireguard VPN.

Unfortunately we do not encrypt RAM as hardware for such is expensive, and doing homomorphic encryption is computationally expensive.

We'll rely on GNU/Linux's usermode separation and EXT4 filesystem permissions to guarantee a degree of confidentiality and integrity. While virtualization could of considerably lowered the attack surface, the lack of nested virtualization on the RPi 5 forces our hand. While we could of had a system where we create two VMs for each user, one for the initial connection and running of tools, the second for Android, it would mean that users would not have full control over their Android VM, which would needlessly limit them.

We create a user on the host machine for each user, from which they launch a podman container, to which they connect over Wireguard, which launches the Android VM.

In-transit checksumming is usually provided by TCP, and Wireguard goes beyond that in that it encrypts, authenticates and authorizes packets using public key cryptography.

A cold boot attack can be made less likely by configuring the BIOS to not boot from untrusted sources and putting glue in the ports. Although that would be outside the scope of this project.

To make an evil maid attack less likely we encrypted the root partition using LUKS. We also have a detachable boot partition that is attached only during boot and detached afterwards and kept separate of the hardware in a secure location. This way if an attacker has access to the physical hardware, they can't tamper with the filesystem without it going unnoticed and it would likely fail to decrypt on boot.

We find it unlikely that someone would go the extent of replacing our hardware with their backdoored hardware, or trying to insert a hardware backdoor in our hardware, given the complexity and low reward for doing so.

Ensuring the supply chain had not been tampered with, and that the manufacturer had not backdoored the hardware is outside the scope of this project.

By hardware it is also meant the firmware that is shipped with the hardware, like the BIOS/UEFI, or the GPU's BIOS, which interestingly, on the Pi the GPU has to be initialized before the CPU with proprietary closed source binary blobs, which perhaps it was imposed to them by the HDMI Forum for HDMI certification.

We don't have measures against bit rot. The ZFS or BTRFS filesystems would provide it, but they add a non-negligible level of complexity to the system, and the data stored in it is not worth the effort to prevent a few random bit flips. Drives like SSDs and HDDs have a measure of error correction that should be adequate.

Same goes for memory. We consider that at this time, the cluster is not doing any operations for which a random bit flip could cause prejudice, therefore non-ECC RAM and non-ECC CPU caches should be fine.

We'll have a policy of not storing anything of importance on the fuzzing cluster and not doing any important computing on the fuzzing cluster.

We manage availability at the software level, rather than at the virtualization solution's level. Given AFL++ has a distributed fuzzing mode with master-slave architecture, we just need to guarantee that a master is present at all times. This is done by a custom election system built on top of GlusterFS, which is also used for sharing findings and coordinating coverage between AFL++ instances.

Our custom election system works by constantly writing timestamps to a shared medium. Once others detect that the current master is no longer writing to the shared medium, the lowest numbered host elects itself to master, one of its AFL++ instances is stopped and restarted in master mode.

Should the ancient master come back online, they'll find that their hostname is not present in the master file and will demote themselves to slave, restarting its AFL++ master instance in slave mode.

5.2. NON-REPUDIATION

Non-repudiation is rather difficult to achieve, especially if we consider the systems administrator as a threat, where they pose as a malicious actor impersonating users and fabricating logs, or mis-attributing them.

A solution for that would include having the system be accessible only if two system administrators are present, assuming they don't fraternize. But what if one is ill? We need redundancy, therefore we need at least three sysadmins, where the system is accessible if at least two are present. This could be achieved by encoding a master secret using Shamir's Secret Sharing Scheme, a cryptographic technique where a secret can be split into n parts and reconstructed by k parts, where k is less than n . In this case a 2-of-3 scheme. To not have to perform SSSS every time, only the root password could be encoded this way. While the sudo user is limited using AppArmor or SELinux to not be able to alter or delete logs.

A limitation of that is that anyone who has physical access to a system, will most likely be able to get through any software security measure. Therefore we'd need to implement a physical vault for the hosts, where the master secret is shared using SSSS... That is outside the scope of this project.

Another problem is guaranteeing that there are no security vulnerabilities, with which one can impersonate another user. This seems impossible at the time of writing. As long as it remains impossible to guarantee the lack of bugs and security vulnerabilities in a program, it remains impossible to attribute an action to someone with 100% certainty, using the program's own logs.

To make matters simple we'll put all the responsibility on a single systems administrator, they'll have complete control of the system, and we'll have a policy where users can not access other systems through our system. This is achieved by setting the firewall to disallow non-root users from initiating connections.

We'll consider the fuzzing cluster as non-critical, where its destruction has no prejudice to anything or anyone. This way we don't need to log actions and deal with the extremely difficult task of ensuring non-repudiation. We could even reward users for finding bugs in the system, to make our reward more appealing than whatever satisfaction they may get in destroying it.

5.3. STATE MANAGEMENT

If the entire infrastructure state is described in code, then backups are not necessary, as we can restore it to its condition simply by running a script. While setting up IaC (infrastructure-as-code) is time consuming initially, it pays off big dividends later by not having to manage backups, by making migration easier, and by making every machine the same, rather than an irreplaceable unicorn.

There seem to be a tendency to anthropomorphize computers, giving them unique names and treating them like babies. Each one unique, critical and irreplaceable. Unfortunately this does not bode well with the requirement for availability and the needs of business, which demand easily replaceable resources, whether human or machine.

A) PODMAN

For IaC we opted to use Podman containers, one because they offer easier rootless mode versus Docker containers, two because it's easy to have versioning of builds, three because they are less time consuming to prepare, four because they are easily moved between machines, and five because it offers a ready made network stack.

Podman or Docker containers have the advantage of being completely described in code at time of creation and are easier to create because of the checkpointing at every instruction. If one does a mistake, (which is extremely common during software development), one can correct that mistake and restart building from the last successful instruction. Not so with traditional VMs. For instance, I can add a step with `"rm -rf /*"` instead of `"rm -rf ./*"`, and I don't have to recreate the container from zero to correct that mistake, which is a huge time saver.

Podman comes with `slirp4netns`, a user-mode network emulator for unprivileged network namespaces. An unsuccessful attempt was made to create linux user-mode network namespaces manually and was quickly abandoned in favor of `slirp4netns`, which allows users to have their own network interfaces and bridges, and their custom routing. All without any root privileges.

`Slirp4netns` allowed us to easily implement the Android Cuttlefish networking configuration, isolated from the host system. As well as routing through Wireguard that sets the user as the gateway for all the traffic, reducing the need for logging and monitoring.

B) ANSIBLE

Ansible is a tool from Redhat for managing remote computers, orchestration and mass deployment. It allows, in theory, idempotent configuration of a multitude of machines, not applying a modification unless needed. It was chosen simply because of familiarity and experience with the tool.

We found idempotency hard to achieve, requiring lots of coding to achieve it. And we haven't achieved it yet, especially for containers. Applying changes to a container requires restarting the container, defeating the purpose of using an idempotent tool. It struggles with assigning permissions to directories and we found that issuing bash commands is orders of magnitude quicker for certain operations, like recursive `chmod` and `chown`.

Other than that, we found it to be an excellent tool for our purposes.

5.4. USER MANAGEMENT AND ACCESS CONTROL

Users are defined in an Ansible configuration file, in this case `vars/main.yml`. Their wireguard ports are defined in the `inventory.yml` file. At this time there is no centralized management server like a LDAP or a CMDB.

To delete an user, they are simply removed from the `vars/main.yml` and `inventory.yml` files, and the Ansible playbook will delete excess users. In this case a task looks for users with a UID higher than 1000 that have a home directory and are not present in the `vars/main.yml` file, then another task deletes them.

For authentication, we use Linux PAM along with SSH PKI, where both hosts and users trust the certificate authority, where host keys and user keys are signed by the CA.

When a user connects, the server sends their signed public key and the user accepts it without confirmation. Then the user sends their signed public certificate and the server accepts it for the given principal. This guarantees no MITM on the first connection and no need to manage user's public keys. Users are not given passwords. They log in with their SSH key signed by the CA.

Without this system, we'd have to permanently store user's public keys somewhere and they'd run the risk of a MITM on first connection, although mitigated considerably by Wireguard.

There is no intermediate CA at this stage, although one could be created to allow a third party to manage users.

5.5. NETWORK ACCESS

The host firewall is set up to only allow established and related outgoing connections for users. They cannot initiate new outgoing connections, except to their own containers on other hosts on their Wireguard UDP port. Users can access their containers only through a Wireguard VPN.

Using a VPN for each user also means that we can treat the user as a gateway and forward all traffic through the user, avoiding the need for network monitoring and logging. While doing so doubles the bandwidth usage, fuzzing is generally not bandwidth heavy, and it's generally not allowed to fuzz someone else's network. For network protocol fuzzing, users can setup a cluster-local solution, given they are all in a Wireguard full-mesh.

5.6. MONITORING AND ALERTING

We collect data such as CPU usage, memory usage and disk usage every 10 minutes through the package sysstat.

If disk space or inode usage is above 85%, an alert email is sent. This is managed by the `check-disk-space.sh` script.

6. VIRTUALIZATION TOPOLOGY

From a control perspective, the Android VM is launched from inside rootless Podman by the user. There they control every aspect of their Android VM. Please see the Figure 1.

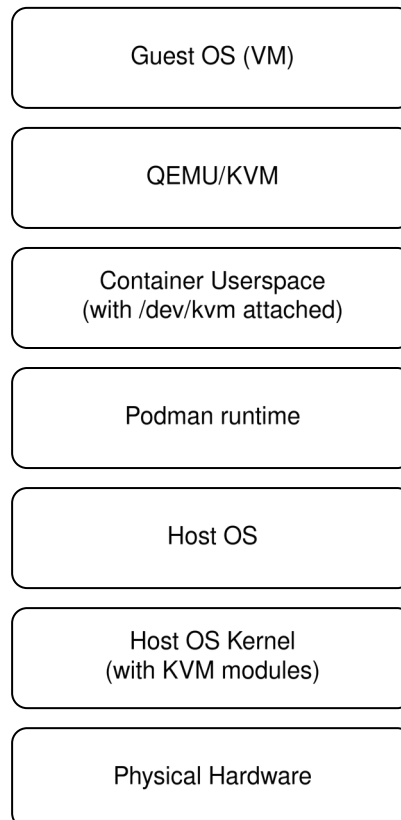


Figure 1: Virtualization

Topology - Control

From an execution flow and priority perspective, the Android VM is at the same level as the Host OS. Please see Figure 2.

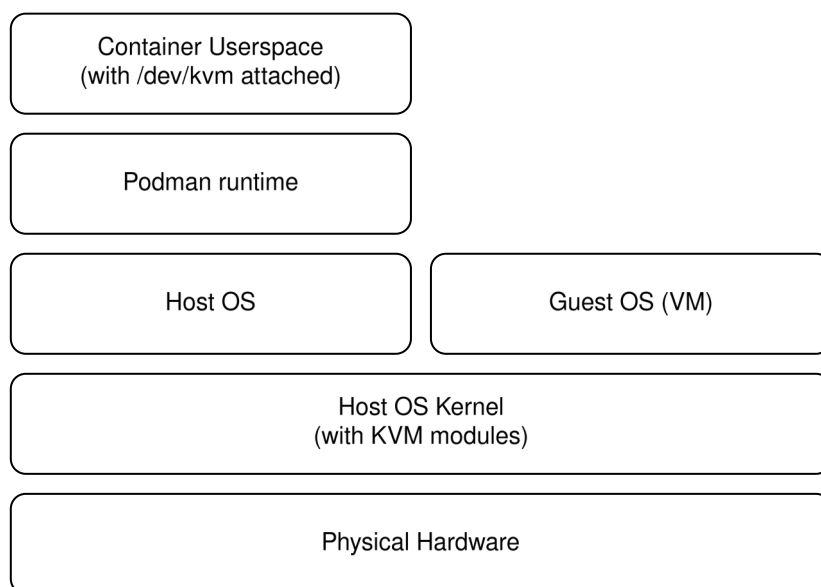


Figure 2: Virtualization Topology - Execution Flow

7. NETWORK TOPOLOGY

Figure 3 details the network stack in terms of control. Each host OS has their Wireguard VPN and each user has their own Wireguard VPN.

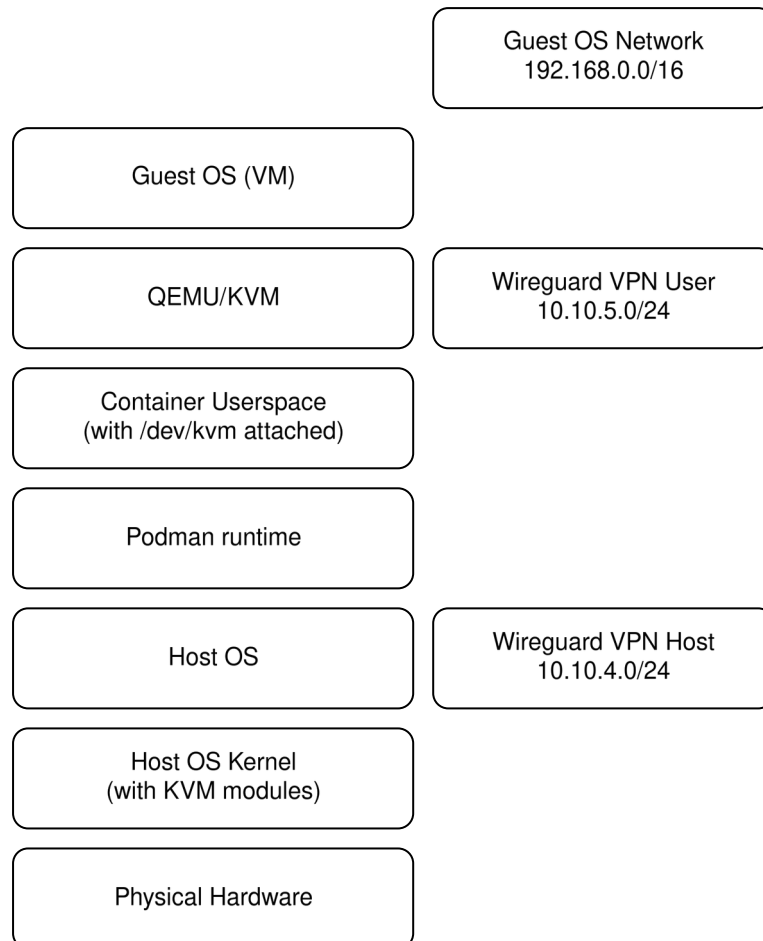


Figure 3: Network Topology - Control

Users are not allowed to initiate outbound WAN connections. Example rules of the nftables firewall on a host :

```

root@rpi01:~# nft list table inet filter
table inet filter {
  chain output {
    type filter hook output priority filter; policy accept;
    ct state established,related counter packets 401871 bytes 123602237 accept
    meta skuid 1001 udp dport 50201 ip daddr 10.222.1.22 counter packets 1 bytes 176 accept
    meta skuid 1002 udp dport 50202 ip daddr 10.222.1.22 counter packets 1 bytes 176 accept
  }
}

```

```

meta skuid 1001 udp dport 50301 ip daddr 10.222.1.23 counter packets 1 bytes 176 accept
meta skuid 1002 udp dport 50302 ip daddr 10.222.1.23 counter packets 5 bytes 880 accept
meta skuid 1001 udp dport 50401 ip daddr 10.222.1.24 counter packets 1 bytes 176 accept
meta skuid 1002 udp dport 50402 ip daddr 10.222.1.24 counter packets 1 bytes 176 accept
ct state new meta skuid 1001 counter packets 0 bytes 0 drop
ct state new meta skuid 1002 counter packets 0 bytes 0 drop
}
}

```

Outbound packets are allowed if the connection is already established. A connection is established when a user connects from the exterior. Only then can packets flow outbound and to the same source.

We can see that users with UID 1001 and 1002 are allowed to communicate to other nodes on port their Wireguard port.

A final drop rule drops any new packet for user 1001 or 1002 that does not respect these rules.

GlusterFS instances communicate only through the host's Wireguard network. Please see Figure 4.

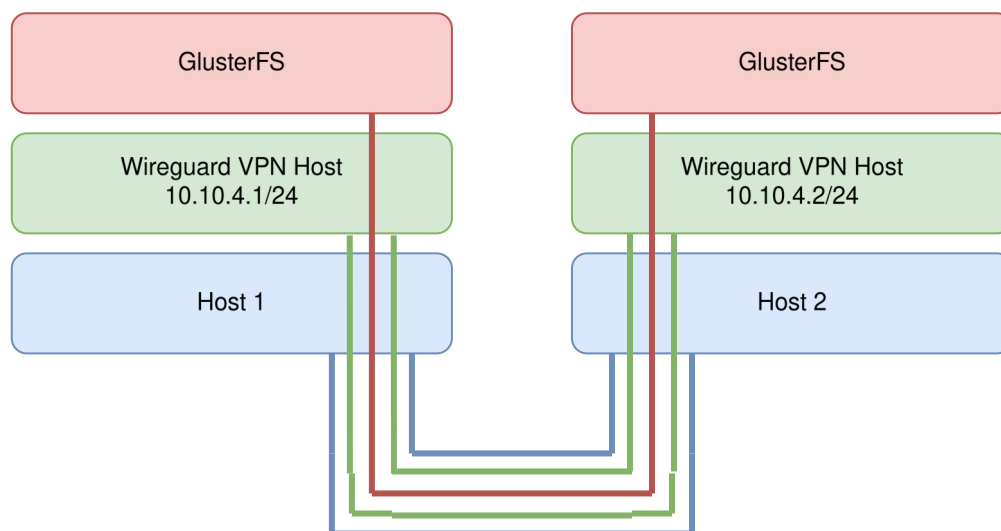


Figure 4: Network Topology - GlusterFS

The advantage is that we don't need to setup authentication and encryption at the GlusterFS level, which uses SSL and is no walk in the park.

Figure 5 shows the client’s Wireguard VPN network. Path a packet takes from container to WAN :

App → Wireguard Client → Container network → Host network → Various switches and routers (if not same network, maybe even WAN) → Client’s workstation (Wireguard) → Client’s workstation (masqueraded) → WAN

The return packet’s path :

WAN → Client’s workstation (reverse masquerade) → Client’s workstation (Wireguard) → Various switches and routers (if not same network, maybe even WAN) → Host network → Container network → Wireguard Client → App

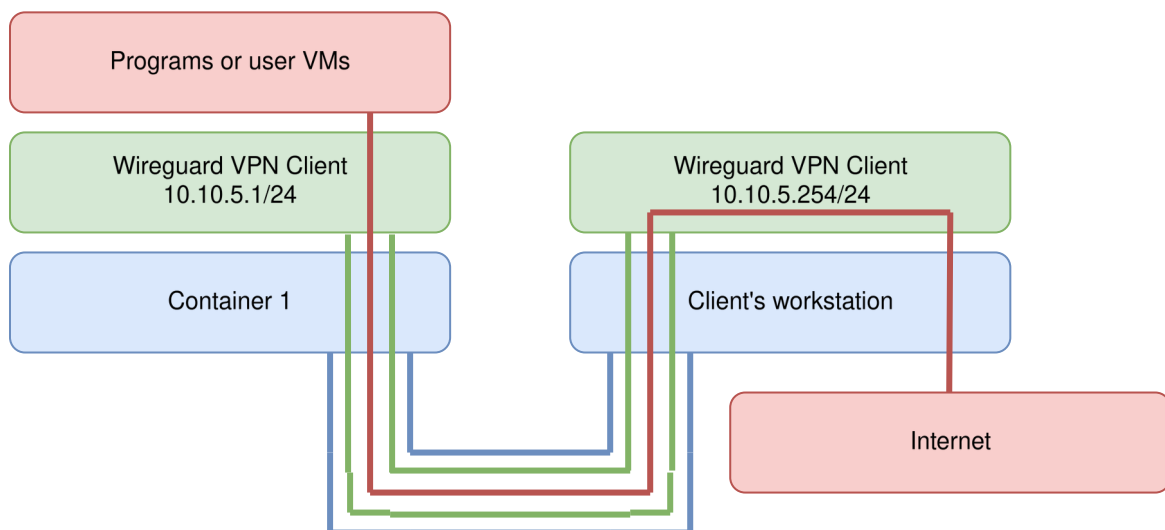


Figure 5: Network Topology - Container to WAN

This approach allows us to not log any packets as the user acts as gateway for all their traffic.

8. DEPLOYMENT WORKFLOW

8.1. SYSADMIN

The workflow for the sysadmin is detailed in `sysadmin/README.md` and here's an overview :

- The sysadmin starts by creating a SSH certificate authority.
- They then define for each host the hostname and IP address.
- They prepare a list of usernames.
- They then use this information to modify `inventory.yml` and `vars/main.yml` to reflect their reality.
- If testing things out, they run `generate-all-secrets.sh` and it will automatically create all SSH keys, SSH certificates, wireguard keys and wireguard configurations for all users. If they don't want to have the user's private key, they should ask the user to generate a key pair, give their public key, have it signed by the CA and have the certificate provided to them. They can use this certificate to log into any computer with the given principal.
- They prepare the RPi images by running the script `rpi-create-encrypted-images.sh`, which produces two images, one for the USB key, one for the main drive. The latter is already encrypted.
- They plug in the main drive followed by the usb drive and retrieve their paths from `/dev`.
- Using the script `rpi-write-and-customize.sh`, the images are customized and written to both drives. Each host's key is signed automatically by the CA and embedded in the main drive.
- The sysadmin inserts the drives into the RPis and boots them.
- They install ansible and execute the playbook. At this point all the ansible tasks will be executed. First the system will be configured with packages, services, groups, users, firewall rules and cpu governor. Consequently, a full Wireguard mesh is configured

between the hosts. Next, GlusterFS is deployed over the Wireguard mesh, where it accepts connections only through Wireguard. Monitoring and master election scripts are deployed. Finally, the podman container is deployed. This can take four hours for the initial container build. Once the container is built, it's launched for every user, in their own context. The scripts `adb-master-afl-monitor.sh` and `adb-sync.sh` run continuously inside the container, making sure to synchronize AFL++'s directories and making sure the right master is running.

A) TIME TO DEPLOY

The following deployment was performed on a workstation with the following specs: Asrock E3C236D4M-4L, Xeon E3-1270 V6, 4x16GB DDR4 2400 ECC UDIMM, 2x1TB WD Red SA500 SSDs in RAID1. The target SSD was connected over USB3.

Image preparation:

```
time ./rpi-create-encrypted-images.sh rpi_images 4G
```

Timings :

```
real    6m6.420s
user    5m38.360s
sys     0m35.058s
```

An SSD and USB drive were connected, attached as `/dev/sdc` and `/dev/sdd` respectively, while the images were being created.

Writing to drives :

```
time ./rpi-write-and-customize.sh /dev/sdc /dev/sdd ./rpi_images/usb.img
./rpi_images/main.img ./rpi_images/rpi_luks.key "rpi01" "10.222.1.21/24" "10.222.1.1"
~/.ssh/ca ~/.ssh/ca.pub supersecret

real    2m21.008s
user    0m15.458s
sys     0m4.506s
```

Another pair of SSD and USB drives were connected, taking ~10 seconds.

```
time ./rpi-write-and-customize.sh /dev/sdc /dev/sdd ./rpi_images/usb.img
./rpi_images/main.img ./rpi_images/rpi_luks.key "rpi02" "10.222.1.22/24" "10.222.1.1"
~/.ssh/ca ~/.ssh/ca.pub supersecret

real    1m57.305s
```

```
user 0m15.326s
```

```
sys 0m4.512s
```

Another pair of SSD and USB drives were connected, taking ~10 seconds.

```
time ./rpi-write-and-customize.sh /dev/sdc /dev/sdd ./rpi_images/usb.img  
./rpi_images/main.img ./rpi_images/rpi_luks.key "rpi03" "10.222.1.23/24" "10.222.1.1"  
~/.ssh/ca ~/.ssh/ca.pub supersecret
```

```
real 1m3.052s
```

```
user 0m15.416s
```

```
sys 0m4.395s
```

Another pair of SSD and USB drives were connected, taking ~10 seconds.

```
time ./rpi-write-and-customize.sh /dev/sdc /dev/sdd ./rpi_images/usb.img  
./rpi_images/main.img ./rpi_images/rpi_luks.key "rpi04" "10.222.1.24/24" "10.222.1.1"  
~/.ssh/ca ~/.ssh/ca.pub supersecret
```

```
real 1m10.450s
```

```
user 0m15.470s
```

```
sys 0m4.656s
```

Attaching the SSD and USB drives to the PIs took another 30 seconds.

The booting of RPis took about 1 minute.

Generating secrets and deploying with Ansible:

```
time ./scripts/generate-all-secrets.sh --mode generate inventory.yml  
vars/main.yml ~/.ssh/ca
```

```
real 0m0.989s
```

```
user 0m0.548s
```

```
sys 0m0.511s
```

```
time ansible-playbook -i inventory.yml playbook.yml --tags all_tasks
```

```
real 233m22.255s
```

```
user 8m54.545s
```

```
sys 3m40.289s
```

In total this took ~ 4,2 hours.

8.2. STUDENT

- The student creates an ed25519 key pair if not provided, sends the public key to the professor/sysadmin for signing, and places the signed key and the certificate authority public key (ca.pub) in the appropriate locations within the .ssh directory.
- The student receives a WireGuard configuration file ([user]_client-wg1.conf) containing the IPs of all containers and edits endpoints and ports if necessary.
- The student installs WireGuard tools on their workstation, places the WireGuard configuration file in the appropriate directory, and starts the tunnel.
- They test connectivity by pinging the container IPs and connect to the containers via SSH using their WireGuard IPs.
- The student checks the created network interfaces using the 'ip a' command to ensure proper setup.
- The student sets up their workstation to act as a gateway for the containers by enabling IP forwarding and configuring masquerading using nftables, iptables, or UFW.
- They verify connectivity and internet access from the containers.
- The student downloads AOSP prebuilt binaries of the Android version of their choosing. Alternatively the student optionally builds AOSP by installing necessary packages, downloading and building the AOSP source code.
- The student installs required packages for Ansible, sets up the Ansible project directory with necessary files, generates the Ansible inventory, tests connectivity, and deploys the Android Cuttlefish VM using Ansible playbooks.
- The student can then use tools such as JADX, Androguard, Ghidra, Frida, and GDB for APK analysis, following an example workflow.
- The student can perform black-box and/or white-box fuzzing using AFL++ and the provided playbooks for orchestration.

A) TIME TO DEPLOY

The preparation time took about 30 minutes.

```
time ansible-playbook -i inventory.yml deploy_cuttlefish.yml
```



```
real    6m38.166s
user    0m41.011s
sys     0m22.442s
```

It took about 6 minutes and 38 seconds for the deployment playbook to run, which does the following :

- Deploys and starts an Android Cuttlefish VM on each host.
- Configures the Android environment by setting up directories, permissions, and networking.
- Copies a test program (fuzzme.c) to the hosts and compiles it for Android.
- Runs a fuzzing tool (AFL++) on the Android device to test the program and find crashes.
- Collects and reports the number of crashes found during fuzzing.

The timing includes two minutes of fuzzing.

8.3. STUDENT WHITE BOX FUZZING WORKFLOW

1. Write a target to be fuzzed in `‘/root/fuzzme.c’`.
2. Launch Ansible : `‘ansible-playbook -i inventory.yml start_fuzzing.yml’`

The Ansible playbook will compile `‘fuzzme.c’`, copy the binary to the Android devices and start fuzzing in distributed fashion on all machines.

8.4. STUDENT BLACK BOX FUZZING WORKFLOW

An example workflow is given in the file `‘student/ansible-project/example-apk-analysis.sh’`.

It consists of setting up a structured project directory, decompiling the target APK with JADX, identifying all the libraries and their dependencies, pulling dependencies from Android using ADB, importing all libraries and their dependencies into Ghidra, decompiling with Ghidra, and generating a call graph using Androguard. The student is further instructed to visit two online guides on how to perform instrumentation and black box fuzzing, as detailed in their README file.

9. FUZZING PERFORMANCE

9.1. OUR ANDROID CLUSTER

The code in Annex 1 was compiled for Android and launched on all four RPi 5 machines using an Ansible playbook. The AFL++ version was 4.33c with a custom shared memory implementation as detailed in Annex 3. The compiler used was clang 18, cross compiling to Android with NDK version r27 and LLVM branch ‘llvm-r522817’ from Google’s repository. The only launch flag was ‘AFL_HARDEN=1’.

After a few minutes we issue the following command on a host to verify progress :

```
adb shell /data/local/tmp/afl-android/bin/afl-whatsup -d
/sdcard/shared_files/output
```

Output :

```
Fuzzers alive : 4
Dead or remote : 12 (included in stats)
Total run time : 1 hours, 4 minutes
Total execs : 16 millions
Cumulative speed : 83101 execs/sec
Total average speed : 20775 execs/sec
Current average speed : 18217 execs/sec
Pending items : 0 faves, 0 total
Pending per fuzzer : 0 faves, 0 total (on average)
Coverage reached : 56.52%
Crashes saved : 19
Hangs saved : 0
Cycles without finds : 339/276/279/295/295/295/295/2401/2265/2228/1588/2
Time without finds : 8 minutes, 49 seconds
```

Figure 6: Screenshot of distributed fuzzing results on our Android cluster

The screenshot shows a cumulative speed across the entire cluster of 83’101 executions per second.

Given there are four machines each with four cores, that’s about 5’194 executions per second per core.

The temperature of the RPi 5s CPUs was at 69 °C on average, as self-reported by the sensor package, with an outside temperature of 26 °C as measured by a thermometer. Each Pi

had a small heat sink. A fan was used to keep them cool. Power use from the outlet was measured at 9W for a single RPi 5 during fuzzing, using a brennenstuhl Primera-Line PM 231 E Wattage and current meter.

During this test, the RPi 5s operated at a fixed 2.4 GHz with the performance governor.

The power of the fan was measured at 28 W with a power factor of 0,99, using the same Wattage meter.

9.2. RPi 5 BUT GNU/LINUX

For comparison, we ran the fuzzing on a single RPi 5 using the same clang 18 compiler, but not crosscompiling for Android. We compiled the same code as in the previous test.

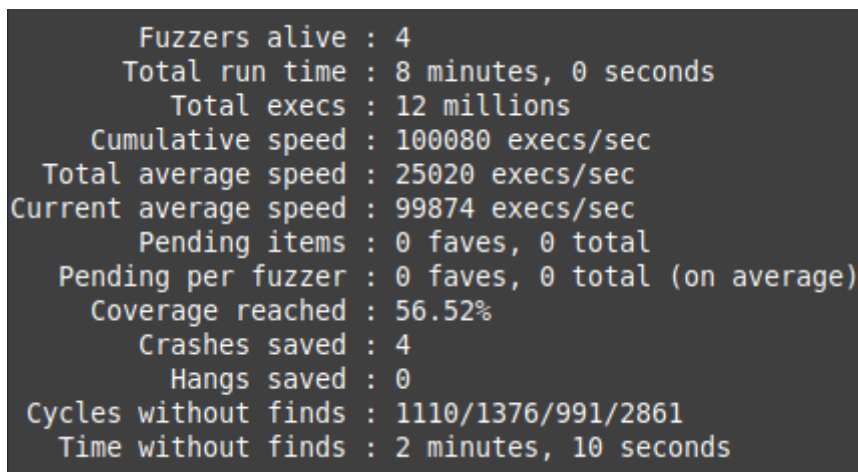
We ran the following command to compile and execute :

```
AFL_HARDEN=1 afl-cc fuzzme.c -o fuzzme
afl-fuzz -M fuzz01 -i input -o output ./fuzzme > /tmp/afl-debug01.log 2>&1 &
afl-fuzz -S fuzz02 -i input -o output ./fuzzme > /tmp/afl-debug02.log 2>&1 &
afl-fuzz -S fuzz03 -i input -o output ./fuzzme > /tmp/afl-debug03.log 2>&1 &
afl-fuzz -S fuzz04 -i input -o output ./fuzzme > /tmp/afl-debug04.log 2>&1 &
```

We let it run for 10 minutes and check results using command :

```
afl-whatsup output
```

Result :



```
Fuzzers alive : 4
Total run time : 8 minutes, 0 seconds
Total execs : 12 millions
Cumulative speed : 100080 execs/sec
Total average speed : 25020 execs/sec
Current average speed : 99874 execs/sec
Pending items : 0 faves, 0 total
Pending per fuzzer : 0 faves, 0 total (on average)
Coverage reached : 56.52%
Crashes saved : 4
Hangs saved : 0
Cycles without finds : 1110/1376/991/2861
Time without finds : 2 minutes, 10 seconds
```

Figure 7: Screenshot of fuzzing results on RPi 5

We obtained 100'080 executions per second, or 25'020 executions per core.

9.3. HETZNER CLOUD

For comparison, we rented a VPS instance on the Hetzner Cloud specified as so : OS Debian 12, 4x vCPU Ampere ARM64 shared, 8 GB RAM. Executing the ‘lscpu’ command we find the CPU is a Neoverse-N1 @ 2.0 GHz.

We compiled AFL++ version 4.33c on it and installed it, although without our patches and using LLVM 14.

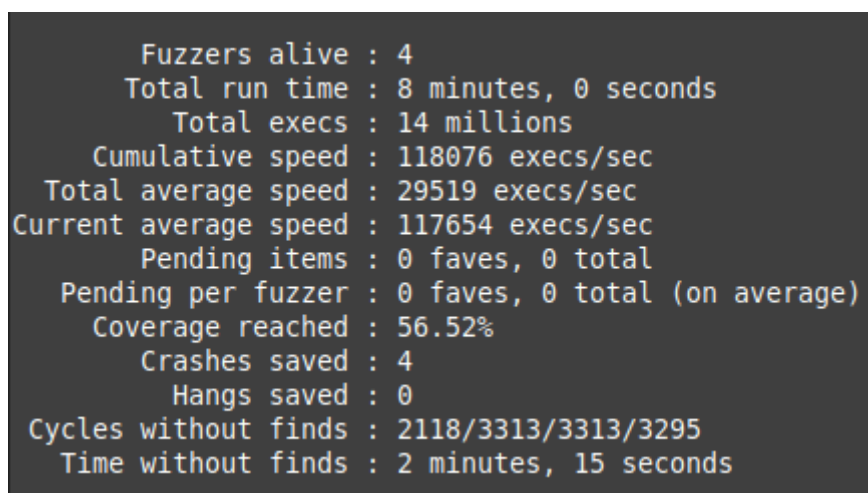
We compiled the same program and executed with the following commands :

```
./afl-fuzz -M fuzz01 -i input -o output ./fuzzme > afl-debug01.log 2>&1 &
./afl-fuzz -S fuzz02 -i input -o output ./fuzzme > afl-debug02.log 2>&1 &
./afl-fuzz -S fuzz03 -i input -o output ./fuzzme > afl-debug03.log 2>&1 &
./afl-fuzz -S fuzz04 -i input -o output ./fuzzme > afl-debug04.log 2>&1 &
```

After a few minutes we check progress with the following command :

```
./afl-whatsup ./output
```

Result:



```

Fuzzers alive : 4
Total run time : 8 minutes, 0 seconds
Total execs : 14 millions
Cumulative speed : 118076 execs/sec
Total average speed : 29519 execs/sec
Current average speed : 117654 execs/sec
Pending items : 0 faves, 0 total
Pending per fuzzer : 0 faves, 0 total (on average)
Coverage reached : 56.52%
Crashes saved : 4
Hangs saved : 0
Cycles without finds : 2118/3313/3313/3295
Time without finds : 2 minutes, 15 seconds

```

Figure 8: Screenshot of fuzzing results on VPS

We see that the VPS instance is fuzzing at 118’076 executions per second combined. Or 29’519 executions per second per core.

9.4. COMMENTARY ON RESULTS

We note that the VPS had 29'519 executions per second per core, while the RPi did 25'020. Given the unequal hardware we can speculate that the VPS had better performance per core compared to the RPi due to more CPU cache and higher IPC, even though operating at a lower frequency, 2.0 GHz for VPS vs 2.4 GHz for RPi.

We note that we produced only 5'194 executions per second per core on the Android VM, or ~79 % less than on GNU/Linux. We weren't expecting such a significant difference, which leads us to believe that there is something very inefficient about our shared memory implementation, as detailed in Annex 3.

Of note is that there is one CPU hungry process by Android Cuttlefish CVD running at all times, 'vhost_device_vsock', which could of tanked our performance. Even though it's only one process on a machine with four cores, which alone does not explain the difference.

We think that it should be possible to obtain very similar results inside the Android VM, given our benchmarking tests showed negligible differences at one layer of virtualization.

Unfortunately due to time constraints, we can't optimize it at this time, nor investigate the 'vhost_device_vsock' process.

We haven't run a benchmark on Android though and the difference could simply be due to

10. DIFFICULTIES

10.1. RPi TROUBLES

No nested virtualization. Major problem that is undocumented.

The RPi imaging utility did not allow encrypting storage, so we had to come up with a custom solution for that, writing an imager from zero.

There are startup scripts that force user creation and are poorly documented. We opted to create a dummy systemuser during image creation. Without that, anyone could plug in a monitor and keyboard and create a user, as the RPi would present them with a dialog for initial setup if no user is created other than root.

The boot partition is mounted at `/boot/firmware`, which is confusing and hard to grasp initially.

The way that initramfs embeds an encryption key is not intuitive. It took a while to get it right. Basically one has to specify a key in `/etc/crypttab`, and where to find it using an initramfs hook with the `KEYFILE_PATTERN` variable. It will then embed the key and unlock the partition on boot, which we chose to identify by its label.

Given the RPi is ARM64, we had to use qemu in static mode to emulate ARM64 to update its initramfs while building the image on x86_64 hardware.

10.2. PODMAN TROUBLES

Not easy to attach a GlusterFS mount. Have to simulate being logged in with `machinectl` and lingering.

Setting up a registry for sharing images in a safe way is rather daunting and would of required setting up a SSL CA. We opted to use a shared directory on each host instead.

Initially the idea was to run as normal user inside the container, but it requires some complex subuid and subgid mappings to be done on the host. The idea was quickly dropped.

Users connect to their container as their username, and then elevate privileges to root inside their container. This way we don't sign a user's SSH keys with root as principal.

10.3. WIREGUARD TROUBLES

Wireguard doesn't seem to allow adding 0.0.0.0/0 allowed IP to a peer without being run as a privileged container. By privileged, it does not mean that it has root access to the host, but that root inside the container can do anything.

10.4. ANDROID TROUBLES

Android Cuttlefish requires a custom network configuration that had to be reverse engineered. It requires at least two bridges and three network interfaces, as well as a DHCP and DNS server. The setup is provided in 'sysadmin/ansible-project/cuttlefish-container/setup-network.sh.' It lacks GNU utilities we are accustomed to.

Setting up fuzzing on Android has been time-consuming and complex, partly because fuzzing itself is a niche activity, and Android fuzzing is even more so. As a result, there is a lack of ready-to-use solutions at the time of writing.

10.5. GLUSTERFS TROUBLES

It seems that a double failure, which can be triggered by simply rebooting two machines at the same time, leaves the GlusterFS frozen, forcing us to nuke it completely from the system and recreate from scratch.

10.6. NDK TROUBLES

Google only provides the NDK for x86_64. Given it requires a specific heavily patched version of LLVM, we have to download it and compile it for ARM64, which takes many hours.

Then use that to build the NDK for ARM64. Getting all the build flags right was no walk in the park. Though the build script looks simple right now.

10.7. AFL TROUBLES

AFL++ uses shared memory for communication between afl-fuzz and the harness. Even though it supposedly has Android support to redirect shm calls to ashmem, it didn't work. Considerable time was spent trying to get it to work, which would of been impossible without a debugger. It was decided to compile GDB given our familiarity with it, to aid in patching AFL++'s shared memory implementation to get it to work on Android.

We found the makefile confusing, with some flags poorly documented.

The makefile for its Frida plugin did not have Android support and had to be patched.

For anyone that wants to debug that part of code, please see Annex 2 for some useful GDB commands.

10.8. GDB TROUBLES

Even though its makefile has over 60 thousand lines, it does not have Android support. We had to compile GMP, MPFR, Libiconv, readline and expat for Android to be able to compile GDB.

GDB had to be patched lightly and we had to get all the build flags [9] right to compile it for Android.

10.9. GHIDRA TROUBLES

The decompiler had to be patched to add ARM64 support, and it had to be compiled.

The headless analyzer does not offer decompilation out of the box. A special script had to be written, 'GhidraDecompiler.java.' Inspiration taken from a youtube video [12] and a guide [15] online.

It also uses Java 21, while JADX uses Java 17, so we had to take care of that incompatibility as well.

11. LICENSE

The Apache License 2.0 was chosen because it is a permissive open-source license that allows for broad use, modification, and distribution of the software. In addition it includes an explicit grant of patent rights, providing legal protection to users of the software. It's compatible with the licenses of modified components, such as AFL++, Ghidra and the Android NDK, which are also licensed under Apache 2.0, and GDB, licensed under the GPL.

Our patches to GDB are documented in the LICENSE file and they are also GPL licensed.

12. AREAS FOR IMPROVEMENT

- We should try to implement Android Cuttlefish snapshotting, to allow resuming of fuzzing sessions. The official documentation [10] seems to imply that the CPU and RAM state are saved as well. Although a note indicates that it's only supported on x86_64 platforms but we could try reverse engineering their code.
- Establish a staging environment, currently blocked by hardware limitations (lack of support for hardware-accelerated nested virtualization). Emulation or unaccelerated virtualization would slow development significantly.
- Deploy LDAP for centralized access management.
- Finalize versioning across all packages to ensure reproducibility and stability (many are already versioned).
- We should implement a LDAP for centralized access control.
- We should finish versioning everything for better reproducibility and stability. Although many packages are already versioned.
- Introduce centralized log collection for better observability.
- Strengthen error management and retry mechanisms for failed tasks.
- Investigate Android performance issues.

13. CONCLUSIONS

This thesis addressed the challenge of automated fuzzing for Android’s native ARM64 components, a critical need for both penetration testing teams and cybersecurity education. Android’s unique environment—featuring Bionic libc, a modified Linux kernel, and ARM64 architecture—complicates traditional fuzzing approaches, especially when source code is unavailable. To bridge this gap, we designed a scalable, distributed fuzzing infrastructure using AFL++, Android Cuttlefish, Podman containers, and Raspberry Pi 5 clusters. The system supports both white-box and black-box testing, with automation scripts and documentation to ensure reproducibility in academic and industrial settings. Performance benchmarks revealed a 79% execution speed drop when fuzzing on Android versus GNU/Linux, highlighting the need for optimizations in shared memory handling and runtime overhead.

Technically, the project delivered a functional infrastructure capable of distributed fuzzing across four ARM64 nodes, achieving 83,101 executions per second cumulatively. Key achievements included integrating AFL++ with custom patches for Android compatibility, deploying rootless Podman containers for isolation, and automating the entire workflow—from disk imaging to user access—using Ansible. The cluster’s design prioritized cost efficiency (CHF 700 for four Pis) and energy consumption (9W per node), while WireGuard and GlusterFS ensured secure, redundant storage and networking. For educators, the system provides a turnkey lab environment; for professionals, it offers a framework for industrializing vulnerability discovery.

Challenges were managed as engineering constraints, not roadblocks. The lack of nested virtualization on Raspberry Pi 5 forced a simplified topology with Android VMs running as first-class containers, avoiding emulation overhead. Compiling tools like GDB and Ghidra for ARM64 required custom builds and dependency resolution, now documented for reuse. Security trade-offs, such as minimal logging for non-repudiation, were justified by the non-critical nature of the cluster, emphasizing usability over auditability.

Critical insights emerged from performance comparisons: Android Cuttlefish’s overhead (e.g., the `vhost_device_vsock` process) significantly impacted fuzzing speed, while cloud-based ARM64 instances (Hetzner) outperformed Pis per core. These findings underscore the importance of hardware-software co-optimization.

Future work should focus on three areas: performance, usability, and integration. Optimizing the custom ashmem implementation and investigating Cuttlefish snapshots could narrow the speed gap with GNU/Linux. Centralized user management (e.g., LDAP) and hybrid cloud-on-premise setups would enhance scalability. Scientifically, exploring AI-driven fuzzing (e.g., LLM-assisted harness generation) or Android-specific vulnerability patterns (e.g., Bionic libc quirks) could yield deeper insights. For industry adoption, integrating with pentesting suites (e.g., Metasploit) and expanding hardware support (e.g., Ampere Altra) are logical next steps.

Ultimately, this work proves that automated Android fuzzing is achievable with modest resources, provided that trade-offs are carefully managed. By open-sourcing the infrastructure, we aim to lower barriers for researchers and educators, fostering collaboration to address the evolving challenges of mobile security. The system’s modularity invites further experimentation—whether in classrooms, bug bounty programs, or large-scale vulnerability research.

APPENDICES

APPENDIX 1 – SIMPLE FUZZING CODE

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <limits.h>
#include <stdint.h>
#pragma clang optimize off
#pragma GCC optimize("O0")
#ifndef __AFL_INIT
#define __AFL_INIT() do {} while (0)
#endif
#ifndef __AFL_LOOP
#define __AFL_LOOP(x) (1)
#endif
static void crash_here(const char *why) {
    fprintf(stderr, "CRASH: %s\n", why);
    fflush(stderr);
    abort();
}
int main(int argc, char **argv) {
    __AFL_INIT();
    setbuf(stdout, NULL);
    setbuf(stderr, NULL);

    uint8_t buf[512];
    while (__AFL_LOOP(UINT_MAX)) {
        memset(buf, 0, sizeof(buf));
        ssize_t len = read(0, buf, sizeof(buf));
        if (len <= 0) continue;
        if (len < 8) continue;
        if (buf[0] == 'a' && buf[1] == 'b' && buf[2] == 'c' &&
            buf[3] == 'd' && buf[4] == 'e' && buf[5] == 'f' &&
            buf[6] == 'g' && buf[7] == '!') {
            crash_here("LOW: abcdefg!");
        }
    }
    return 0;
}

```

APPENDIX 2 – GDB DEBUGGING AID

```
set breakpoint pending on
set environment AFL_DEBUG=1
set environment AFL_DEBUG_CHILD=1
set environment ASAN_OPTIONS=abort_on_error=1:detect_leaks=0:symbolize=0
set environment AFL_FORKSRV_INIT_TMOUT=1000000
set follow-fork-mode child

breakpoint shmat
watch __afl_area_ptr
watch __afl_map_addr
handle SIGBUS nostop noprint pass
run -i input -o output -d -- ./target
```

APPENDIX 3 – CUSTOM SHARED MEMORY IMPLEMENTATION

```

#ifdef __ANDROID__
#ifndef _ANDROID_ASHMEM_H
#define _ANDROID_ASHMEM_H
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#include <sys/syscall.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/ashmem.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#define ASHMEM_DEVICE "/dev/ashmem"
#define DEFAULT_MAP_SIZE 65536
#define MAX_MAPPINGS 128 // Maximum tracked mappings
// #define ENABLE_ASHMEM_DEBUG
#ifdef ENABLE_ASHMEM_DEBUG
#define DBG_PRINT(fmt, ...) fprintf(stderr, "ASHMEM: " fmt, ## __VA_ARGS__)
#else
#define DBG_PRINT(fmt, ...) do { } while (0)
#endif
typedef struct {
    void *addr;
    size_t size;
} mapping_entry;
static mapping_entry mapping_table[MAX_MAPPINGS];
static void register_mapping(void *addr, size_t size) {
    for (int i = 0; i < MAX_MAPPINGS; i++) {
        if (mapping_table[i].addr == NULL) {
            mapping_table[i].addr = addr;
            mapping_table[i].size = size;
            break;
        }
    }
}
static size_t unregister_mapping(const void *addr) {
    size_t size = 0;
    for (int i = 0; i < MAX_MAPPINGS; i++) {

```



```

        if (mapping_table[i].addr == addr) {
            size = mapping_table[i].size;
            mapping_table[i].addr = NULL;
            mapping_table[i].size = 0;
            break;
        }
    }
    return size;
}

static void cleanup_all_mappings(void) {
    for (int i = 0; i < MAX_MAPPINGS; i++) {
        if (mapping_table[i].addr != NULL) {
            DBG_PRINT("Cleanup: unmapping address=%p, size=%zu\n",
                mapping_table[i].addr, mapping_table[i].size);
            munmap(mapping_table[i].addr, mapping_table[i].size);
            mapping_table[i].addr = NULL;
            mapping_table[i].size = 0;
        }
    }
}

static void __attribute__((constructor)) init_ashmem(void) {
    DBG_PRINT("Android ASHMEM implementation initialized\n");
    atexit(cleanup_all_mappings); // auto cleanup on exit
}

__attribute__((used, visibility("default")))
static void *ashmem_shmat(int __shmid, const void *__shmaddr, int __shmflg) {
    (void)__shmflg;
    DBG_PRINT("shmat called with shmid=%d, addr=%p, flags=0x%x\n",
        __shmid, __shmaddr, __shmflg);
    size_t map_size;
    // Anonymous/shared mapping request
    if (__shmid == 0) {
        map_size = DEFAULT_MAP_SIZE;
        DBG_PRINT("Anonymous mapping request, size=%zu\n", map_size);
        void *ptr = mmap(NULL, map_size, PROT_READ | PROT_WRITE,
            MAP_SHARED | MAP_ANONYMOUS, -1, 0);
        if (ptr != MAP_FAILED) {
            register_mapping(ptr, map_size);
        }
        return ptr;
    }
    // Get actual ashmem region size
    int size = ioctl(__shmid, ASHMEM_GET_SIZE, NULL);
    if (size < 0) {
        DBG_PRINT("Invalid ashmem region: %s\n", strerror(errno));
        return (void *)-1;
    }
    map_size = (size_t)size;
    // Map ashmem region

```

```

    void *ptr = mmap((void *)__shmaddr, map_size,
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED,
                    __shmid, 0);

    if (ptr == MAP_FAILED) {
        DBG_PRINT("Mapping failed: %s\n", strerror(errno));
        return (void *)-1;
    }
    register_mapping(ptr, map_size);
    DBG_PRINT("Mapping succeeded at %p, size=%zu\n", ptr, map_size);
    return ptr;
}

__attribute__((used, visibility("default")))
static int ashmem_shmdt(const void *address) {
    DBG_PRINT("shmdt called with address=%p\n", address);
    size_t size = unregister_mapping(address);
    if (size == 0) {
        DBG_PRINT("Address %p not found in mapping table\n", address);
        errno = EINVAL;
        return -1;
    }
    DBG_PRINT("Unmapping address=%p, size=%zu\n", address, size);
    return munmap((void *)address, size);
}

__attribute__((used, visibility("default")))
static int ashmem_shmctl(int __shmid, int __cmd, struct shmctl *__buf) {
    (void)__buf;
    DBG_PRINT("shmctl called with shmid=%d, cmd=%d\n", __shmid, __cmd);
    if (__cmd == IPC_RMID) {
        int length = ioctl(__shmid, ASHMEM_GET_SIZE, NULL);
        unsigned int safe_length = (length >= 0) ? (unsigned int)length : 0;
        struct ashmem_pin pin = {0, safe_length};
        ioctl(__shmid, ASHMEM_UNPIN, &pin);
        close(__shmid);
        DBG_PRINT("IPC_RMID: region unpinned and closed, size=%u\n",
                  safe_length);
    }
    return 0;
}

__attribute__((used, visibility("default")))
static int ashmem_shmget(key_t __key, size_t __size, int __shmflg) {
    (void)__shmflg;
    DBG_PRINT("shmget called with key=%d, size=%zu, flags=0x%x\n",
              __key, __size, __shmflg);
    int fd = open(ASHMEM_DEVICE, O_RDWR);
    if (fd < 0) {
        DBG_PRINT("Failed to open %s: %s\n", ASHMEM_DEVICE, strerror(errno));
        return -1;
    }
}

```

```

char name[32];
snprintf(name, sizeof(name), "afl_%d", __key);
ioctl(fd, ASHMEM_SET_NAME, name);
size_t alloc_size = (__size > 0) ? __size : DEFAULT_MAP_SIZE;
if (ioctl(fd, ASHMEM_SET_SIZE, alloc_size) < 0) {
    DBG_PRINT("Failed to set size: %s\n", strerror(errno));
    close(fd);
    return -1;
}
DBG_PRINT("Created ashmem region fd=%d, size=%zu\n", fd, alloc_size);
return fd;
}
#define shmget    ashmem_shmget
#define shmat     ashmem_shmat
#define shmdt     ashmem_shmdt
#define shmctl    ashmem_shmctl
#endif /* !_ANDROID_ASHMEM_H */
#endif /* !_ANDROID__ */

```


BIBLIOGRAPHY

- [1] “▷ Workstations mit Ampere Altra ARM-CPU zusammenstellen | MIFCOM.” Accessed: Aug. 20, 2025. [Online]. Available: <https://www.mifcom.ch/workstations-ampere-altra-cid774>
- [2] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, AFL++. (Jan. 2022). C. Accessed: Aug. 20, 2025. [Online]. Available: <https://github.com/AFLplusplus/AFLplusplus>
- [3] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, doi: 10.1145/96267.96279.
- [4] “Attestation compatibility guide | Articles | GrapheneOS.” Accessed: Aug. 20, 2025. [Online]. Available: <https://grapheneos.org/articles/attestation-compatibility-guide>
- [5] geerlingguy, “Benchmark Adlink Ampere Altra Dev Kit - 64-core 2.2 GHz · Issue #19 · geerlingguy/top500-benchmark,” GitHub. Accessed: Aug. 20, 2025. [Online]. Available: <https://github.com/geerlingguy/top500-benchmark/issues/19>
- [6] geerlingguy, “Benchmark Pi 5 model B 16 GB (new 2025 version with D0 stepping) · Issue #51 · geerlingguy/top500-benchmark,” GitHub. Accessed: Aug. 20, 2025. [Online]. Available: <https://github.com/geerlingguy/top500-benchmark/issues/51>
- [7] ByteBrigand, ByteBrigand/Bachelor-IT-HEPIA-2025. (Aug. 20, 2025). Accessed: Aug. 20, 2025. [Online]. Available: <https://github.com/ByteBrigand/Bachelor-IT-HEPIA-2025>
- [8] “CAPEC - CAPEC-28: Fuzzing (Version 3.9).” Accessed: Aug. 20, 2025. [Online]. Available: <https://capec.mitre.org/data/definitions/28.html>
- [9] “Configure Options (Debugging with GDB).” Accessed: Aug. 20, 2025. [Online]. Available: <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Configure-Options.html#Configure-Options>
- [10] “Cuttlefish: Snapshot and restore,” Android Open Source Project. Accessed: Aug. 20, 2025. [Online]. Available: <https://source.android.com/docs/devices/cuttlefish/snapshot-restore>

- [11] J. Anyam, R. R. Singh, H. Larijani, and A. Philip, “Empirical Performance Analysis of WireGuard vs. OpenVPN in Cloud and Virtualised Environments Under Simulated Network Conditions,” *Computers*, vol. 14, no. 8, p. 326, Aug. 2025, doi: 10.3390/computers14080326.
- [12] Real Hacker Hours, Episode 35: Ghidra Headless Scripting, (July 02, 2021). Accessed: Aug. 20, 2025. [Online Video]. Available: <https://www.youtube.com/watch?v=qtN7C-KXg1Q>
- [13] “Fuzzing - an overview | ScienceDirect Topics.” Accessed: Aug. 20, 2025. [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/fuzzing>
- [14] “Fuzzing | OWASP Foundation.” Accessed: Aug. 20, 2025. [Online]. Available: <https://owasp.org/www-community/Fuzzing>
- [15] “Headless Analyzer README.” Accessed: Aug. 20, 2025. [Online]. Available: <https://static.grumpycoder.net/pixel/support/analyzeHeadlessREADME.html>
- [16] “Number of Android applications on Google Play (Aug 2025),” AppBrain. Accessed: Aug. 20, 2025. [Online]. Available: <https://www.appbrain.com/stats/number-of-android-apps>
- [17] B. Miller, “Project List,” COMPUTER SCIENCES DEPARTMENT UNIVERSITY OF WISCONSIN-MADISON, 1988, Accessed: Aug. 20, 2025. [Online]. Available: <https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf>
- [18] A. He, B. Xu, and J. Wu, “Security Analysis of WireGuard,” 2018, [Online]. Available: <https://courses.csail.mit.edu/6.857/2018/project/He-Xu-Xu-WireGuard.pdf>
- [19] S. López, slp/start-avm. (July 01, 2025). C. Accessed: Aug. 20, 2025. [Online]. Available: <https://github.com/slp/start-avm>
- [20] “stefan.antun / travailbachelor2025 · GitLab,” GitLab. Accessed: Aug. 20, 2025. [Online]. Available: <https://githepia.hesge.ch/stefan.antun/travailbachelor2025>
- [21] “Virtualization Support | raspberrypi/linux,” DeepWiki. Accessed: Aug. 20, 2025. [Online]. Available: <https://deepwiki.com/raspberrypi/linux/5-virtualization-support>
- [22] “What is fuzz testing?,” about.gitlab.com. Accessed: Aug. 20, 2025. [Online]. Available: <https://about.gitlab.com/topics/devsecops/what-is-fuzz-testing/>
- [23] H. Tan, Q. Luo, J. Li, and Y. Zhang, “LLM4Decompile: Decompiling Binary Code with Large Language Models,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 3473–3487. doi: 10.18653/v1/2024.emnlp-main.203.