LibAFL QEMU: A Library for Fuzzing-oriented Emulation

Romain Malmain, Andrea Fioraldi, Aurélien Francillon

LibAFL QEMU: A Library for Fuzzing-oriented Emulation

Romain Malmain, Andrea Fioraldi, Aurélien Francillon
EURECOM
{firstname.lastname}@eurecom.fr

***Abstract*—Despite QEMU's popularity for binary-only fuzzing, the fuzzing community faces challenges like the proliferation of hard-to-maintain QEMU forks and the lack of an up-to-date, flexible framework well-integrated with advanced fuzzing engines. This leads to a gap in emulation-based fuzzing tools that are both maintainable and fuzzing-oriented.**

To cope with that, we present LIBAFL QEMU, a library written in Rust that provides an interface for fuzzing-based emulation by wrapping around QEMU, in both system and user mode. We focus on addressing the limitations of existing QEMU forks used in fuzzing by offering a well-integrated, maintainable and up-to-date solution. In this paper, we detail the design, implementation, and practical challenges of LIBAFL QEMU, including its APIs and fuzzing capabilities and we showcase the library's use in two case studies: fuzzing an Android library and a Windows kernel driver.

We compare the fuzzers written for these 2 targets with the state-of-the-art, AFL++ qemu_mode for the Android library, and KAFL for the Windows driver. For the former, we show that LIBAFL QEMU outperforms AFL++ qemu_mode both in terms of speed and coverage. For the latter, despite KAFL being built above hardware-based virtualization instead of emulation, we show we can run complex targets such as Windows and still reach comparable performance, with an overhead expected by a software emulator.

I. INTRODUCTION

Fuzzers are popular tools for software security assessment both in a software testing perspective, to catch defects before production, and in an adversarial context, in which a security researcher targets third-party codebases, in some cases without having access to the source code.

Fuzzing binary code is an important problem for the security community, with many challenges to address [9], [18], [37], [53]. For a long time, QEMU [7] has been one of the most popular choices for binary-only fuzzing, with the pioneering work of AFL qemu_mode [64] to fuzz Linux userspace programs and TriforceAFL [30] to fuzz operating systems with coverage feedback. QEMU, the Quick Emulator, is a popular choice for various reasons in comparison to emulators written from scratch. In particular, QEMU supports a wide number of CPU architectures, emulates extensive difficult-to-develop and maintain hardware, and regularly supports new features, often before they are available on real hardware [26].

Workshop on Binary Analysis Research (BAR) 2024
1 March 2024, San Diego, CA, USA
ISBN 979-8-9894372-0-7
https://dx.doi.org/10.14722/bar.2024.23xxx
www.ndss-symposium.org

Over the years, many frameworks offered fuzzing capabilities on top of a modified version of QEMU, such as Qiling [3] or PANDA [19], introducing a growing number of forks of QEMU that are hard to maintain up to date with upstream. The set of APIs to control the emulator is often, like in the case of Qiling, based on low-performance languages, such as Python, unsuited for fuzzers development, or the framework does not offer a tight integration with fuzzing engines.

As a result, new improvements in the field of emulation-based fuzzing with QEMU do not have an up-to-date framework that bridges the latest versions of QEMU to the world of fuzzing. So, researchers are often stuck to old versions, or they have to spend a considerable engineering effort to build custom forks of QEMU for their specific need [28], [49].

What is missing is a flexible solution that does not require the user to patch QEMU, a minimal variation to the upstream codebase so that it can be maintained up to date easily. It should also be well integrated with a state-of-the-art fuzzing framework that offers performance and scalability as first-class design choices.

To address these needs, in this paper, we proposed LIBAFL QEMU, a library written in Rust that wraps QEMU as a static library with most of the codebase for instrumentation and fuzzing written out of the main source tree of QEMU for ease of maintenance. It is integrated with LIBAFL, the modular framework for fuzzers development and it is not a standalone tool. It offers a wide range of instrumentation options and fuzzing capabilities for both userspace and system emulation such as a powerful hooking system, a ready-to-use collection of instrumentation like AddressSanitizer for binaries and support for snapshot fuzzing.

Our approach puts genericity, modularity, and maintainability in the foreground. Treating QEMU as a library makes it possible to interact with the emulator at various moments of the execution of the target with minimal overhead. Having an API to control QEMU, it is no longer necessary to write C code into QEMU's codebase for instrumentation. Unlike PANDA [19], we created a thin wrapper around QEMU's codebase to maximize performance and minimize maintenance hassles: we only needed to modify around 2K lines of code in the core QEMU code. Although we modified QEMU with LIBAFL [23] applications in mind, our API could be used for other purposes.

When compared to the most used solution for userspace fuzzing, AFL++ [22] QEMU, it not only offers an advantage in terms of fuzzer development with a customizable interface, but it can also outperform it in terms of speed, uncovered coverage

and bugs. In this paper, we described a use case on how to fuzz an Android library with both tools and compared the outcome of the fuzzing campaigns.

In addition to userspace applications, we make use of QEMU's features to the fullest by handling full system targets. Our implementation not only runs most x86 binaries, just like with KAFL [51], but it can also execute embedded applications or full virtual machines that require the most recent versions of QEMU. We make multiple tools available, greatly facilitating harness development without compromising performance. We evaluate this LIBAFL QEMU system emulation feature with fuzzing Windows 10's NTFS kernel driver and compare it to the state-of-the-art - KAFL [51] / Nyx [50].

In short, the contributions of this paper are:

- We developed a library for fuzzing-oriented emulation with a powerful interface and many different built-in capabilities;
- We present two use cases on how to fuzz an Android library and a Windows kernel driver and compare our library with the state-of-the-art;
- We release the framework and the examples as open-source software;

II. BACKGROUND

In this section, we introduce binary-only fuzzing and its usage. Then, we discuss the available solutions for harnessing, instrumenting and fuzzing binary code.

A. Binary-only Fuzzing

Multiple purposes of fuzzing binary code justify the adversarial setup in testing software without all the information that a source-level engine may have.

Firstly, the straightforward reason is that security researchers need to test third-party software to report vulnerabilities without working for the company developing the software. But, not limited to that, companies themselves use binary-only fuzzers on third-party software to test unmaintained dependencies – e.g., a library in the supply chain is buggy but the developing company is now closed – and eventually patch them at the binary level, or even to test the code produced by the company itself but that can only compile under complex build systems or non-conventional compilers. As the last reason, binary-only fuzzing is also a complement to source-level fuzzing as the latter is used to test the code in a debug environment, while the former can be used to test directly the production binaries that are going to be released and uncover bugs that may stay silent in the debug environment due to, for instance, different compiler optimizations.

B. Static or Dynamic Emulation?

Emulation The main purpose of an emulator is to reproduce, to some extent, the behavior of a given execution environment. While some emulators can only execute simple snippets of assembly code, others can accurately reproduce entire instruction sets for a wide range of architectures. In addition, more complex emulators must take into account some kind of environment: the operating system for executables, the hardware and devices running underneath for full-system emulation, etc. Many different emulators are still being actively maintained as of today [7], [8], [11], [29], [33], [41]. One of the most important features of an emulator, alongside correctness, is speed: we often expect an emulator to run with reasonable overhead. Naive approaches are unable to achieve acceptable execution time for real-world programs. Different approaches have been developed over the years to address this issue. When the host and the target architecture are similar, a hardware-assisted approach like virtualization [56] has shown to be particularly efficient in getting near-native performances (when the ISA supports it). However, this technique cannot be used to emulate foreign architectures. It also restricts the control we have over the target at runtime, since the hypervisor must wait for the virtual machine to stop before performing introspection. On the other hand, software-based emulation can fully control the target at any point in time. For those reasons, it remains a relevant option to run and test various programs. We mainly find two major software-powered emulator families in the wild: Dynamic Binary Translation and Static Binary Translation.

Dynamic Binary Translation Dynamic Binary Translation (DBT) is an emulation technique that translates a target binary's code into host instructions at runtime. The input code is either directly translated into native instructions or goes through an intermediate representation (IR) before the final translation. Various DBT-based emulators actively maintained as of today are available [7], [11], [36]. A vast range of techniques have been developed over the past decades to improve their accuracy and performance [46]. A widely used method [7], [36] is JIT compilation: code is translated per chunk at runtime. It is quite common for many emulators to include some kind of linking between these chunks to avoid going out of the target whenever possible. Once the translation is over, the resulting native code is often cached to allow reuse if the corresponding code block must be executed again. Ideally, frequently executed code should be kept in the cache and reused often. This approach presents multiple advantages compared to interpretation: execution can be greatly sped up (use of host hardware features, translation-time optimizations), it is very flexible (code can be easily analyzed or injected), and it can make use of runtime information to improve performances further.

Static Binary Rewriting Contrary to Dynamic Binary Translation (DBT), Static Binary Translation (SBT) transforms the target binary before it starts running. A static binary rewriter takes a binary as input, rewrites it entirely (while applying any desired transformations), and outputs another executable binary, potentially for a foreign ISA. The final binary can be run as-is without the need for any extra work. This technique may be used to modify a binary whose source code is potentially unavailable for optimization [42], [57] and hardening [43], [45], [58], [59] purposes or to port it from one ISA to another [48], [62], [63]. It also makes it possible to add custom instrumentation to any existing binary. Nowadays, a common and widely used method is to lift the input binary into the intermediate representation of a modern compiler like LLVM [16], [17] or GCC [25]. An SMT solver can thus use many features already implemented by the compiler (optimizations, instrumentation, cross-architecture compilation, etc). It is only natural for many fuzzers [16]–[18], [38], [44], [65] to make use of static binary rewriting.

C. Pros and Cons of both Approaches for Fuzzing

SBT is a promising technique for fuzzing: since it instruments the target statically, most of the extra work is done ahead of time. It can run natively on the target hardware, the only overhead being the (lightweight) fuzzer's feedback code (contrarily to DBT, which must also run its engine at runtime). As stated in the previous section, SBT can make use of the full power of modern compilers, further improving the quality of the final instrumentation.

However, DBT presents multiple advantages that have not been, to our knowledge, overcome by existing static rewriters. First, we are unaware of any tool being able to rewrite full-system binaries for any existing architecture so far. It is a huge issue, excluding a lot of security-sensitive code. Static rewriters also tend to focus on a small number of architectures (mostly x86 and ARM). Meanwhile, many emulators can completely emulate a lot of architectures, including x86, ARM, MIPS, PowerPC, RISC-V, etc. As a result, emulator-based fuzzers can target every supported architecture without any additional effort. On the other hand, static rewriting often requires writing complex extra code to interface with a given ISA. In addition, a lot of static rewriters often presume target binaries enforce some kind of structure when emulators do not make any assumption. Emulators also have full access to precious runtime information, which static rewriters cannot get easily. SBT also faces a theoretical problem: disassembly is undecidable [27], [60]. Thus, SBT will never be able to rewrite any binary. DBT is not concerned by this issue since it translates the binary along the execution flow. Static Binary Translation not only requires rewriting the target software but also every runtime dependency (e.g. libraries) it may have. This is especially true when instrumentation must propagate information like for tainting [13]. Indeed, not instrumenting some parts of the code would most likely result in a loss of symbols that would degrade the quality of the final result. Previous work like MetaEmu [14], UnicoreFuzz [35], SAFIREFUZZ [53], and Icicle [15] have already shown that DBT can be used for fuzzing purposes. However, we believe working with an easy-to-merge QEMU fork directly, in addition to high speed, would highly benefit from the community's efforts: making a performant yet complete emulator is no easy task.

D. QEMU and the Tiny Code Generator

QEMU [7] (Quick Emulator) is an emulator capable of both user-mode and full-system emulation. While QEMU user mode runs Linux and BSD binaries written for any supported architectures by performing syscall translation, QEMU system mode emulates an entire machine, including many devices. QEMU is also able to run multiple threads in parallel. Although QEMU can make use of hardware-assisted virtualization by leveraging KVM, we will solely focus on emulation. Emulation is roughly done in two steps. First, the target code is translated into an intermediate representation called TCG (Tiny Code Generator). After passing through the optimization pipeline, the intermediate code is translated into the host architecture's machine language. The compilation unit of TCG is a translation block (TB), the equivalent of a basic block in most modern compilers. QEMU has multiple layers of TB caches used extensively to improve performance drastically. As a result, a TB is generated on the fly if it is not already cached. QEMU also links TB together whenever possible to speed up the execution and links back to the emulator when further translation must be performed. QEMU also allows the execution of any C code directly during the target emulation, called helper functions. They are often used to perform complex operations or change QEMU's internal state. QEMU embeds a large number of device implementations, including various display, audio, and input drivers. They enable running complex full-system targets in an environment close to bare-metal hosts. QEMU has a large and active community, meaning it should continue to be maintained in the coming years.

III. Design

As discussed in the previous section, the fuzzing community is still using emulators that are either not at the state of the art or with limited instrumentation capabilities. It is also common to find many specialized QEMU forks for a specific fuzzer that cannot be easily reused or that are quickly abandoned and not kept up-to-date with the QEMU project.

In this section, we present the design of LibAFL QEMU, the library we developed to write emulation-based harnesses to fuzz Linux user space programs as well as whole systems.

A. General Overview

Writing a library using QEMU as a binary translation engine for fuzzing was the first practical challenge that we had to overcome: to transform QEMU, a standalone tool, into a library and expose its inner workings to Rust.

Thus, `qemu-libafl-bridge`, our forked version of QEMU, which we maintain up to date with the master branch upstream, can produce a shared object instead of a binary. However, due to the nature of Rust crates in which the libraries are included statically, we opted to avoid dynamic linking and so we engineered two helper libraries to build QEMU as a Rust crate and expose its API to Rust via C FFI bindings (a way to call native C functions from Rust) [1], for a total of 3 Rust libraries:

- `libafl_qemu_build` is the library that offers an API to build QEMU and generate automatically bindings with `bindgen`.
- `libafl_qemu_sys` is the library exposing the C FFI bindings to Rust. It invokes the `libafl_qemu_build` API during the build process, notably responsible for generating the code available in `libafl_qemu_sys`.
- `libafl_qemu` is the main library which has `libafl_qemu_sys` as a dependency and wraps it in a Rust-friendly API with the capabilities that will be discussed later in the section.

Including `libafl_qemu` as a dependency in a fuzzer crate will result in the automatic building and linking of QEMU as a library using only the `cargo build` command as any other Rust crate.

We divided the functionalities of the main library into 3 levels of abstractions in terms of design:

1) The low-level API, exposed by the `Emulator` singleton. The API is close to `libafl_qemu_sys` but wrapped in safe Rust.

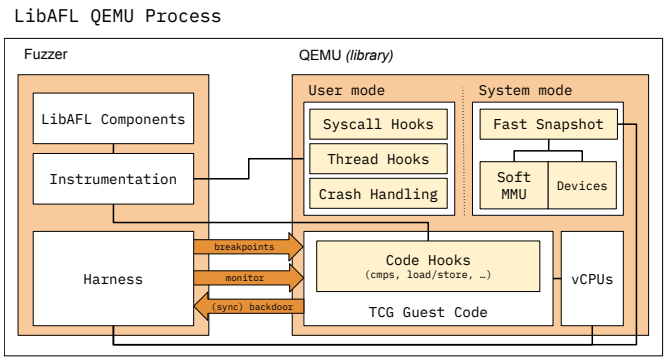

Fig. 1: LIBAFL QEMU interactions between Fuzzer and Guest

2) The high-level API is an abstraction built on top of the low-level ones. This API offers hooking capabilities that can interact with the LIBAFL fuzzer's state and with the other hooks to implement systems like `AddressSanitizer`.
3) The fuzzing-oriented capabilities, built on top of the low and high-level APIs, implement a set of pre-made instrumentations such as `AddressSanitizer` or coverage tracking called *instrumentation helpers* that are useful to build a fuzzer without writing any hook from scratch.

LIBAFL QEMU supports any architecture supported by QEMU for at least the basic functionalities, while many additional fuzzing-oriented capabilities are available only for x86, x86_64, ARM, AArch64, MIPS, PPC and Hexagon. Each architecture has a module defining the registers, the disassembler and the *cdecl* calling convention. The target architecture can be specified either using a feature flag or an environment variable. Adding an architecture to LIBAFL QEMU requires around 100 lines of code in C and 300 lines of code in Rust, and only requires moderate knowledge of the target architecture.

Besides the architecture, the user can choose to use user mode or system mode QEMU. Enabling one or the other will result in some shared APIs being exposed that are specific to the chosen emulation mode. For instance, listing the memory mapping is available only in user mode and the TLB (QEMU's emulated virtual memory translations from virtual to physical addresses) query only in system mode.

The different layers of abstraction offer the possibility to set up a working fuzzer with a few lines of Rust code using the helpers and also to write an optimized tracer using the low-level API defining C FFI functions from Rust that are directly called from the JITted code. The goal of the library is to provide a DBI-like API tailored for fuzzing using QEMU and its wide support for different hardware.

B. Low-Level API

The low-level APIs are mainly exposed as part of `Emulator`, the singleton instance in Rust that interacts with QEMU. The main functionalities exposed are the management of the state of all the running CPUs – that have their corresponding objects wrapped from C that can be used to access registers and memory, save and restore the CPU state, verify the validity of an address and query the TLB in full system mode – the various API to manage breakpoints and hooks and the custom GDB commands and hypercalls. Of course, this component is used also to run the emulator until the next breakpoint.

In user mode, it is extended with memory mapping retrieval and modification with an API similar to `mmap`, guest to host and backward address translation routines and ELF path and base address information. An ELF parser is available in the library to locate symbols and thus set breakpoints by name.

In system mode, the additional facilities allow the user to query the MMU for physical addresses, and snapshot the emulator state with the QEMU built-in snapshots and the list of available devices. QEMU built-in snapshots are too slow to be used in fuzzing but useful to restore a VM without running the very slow boot phase every time.

A summary of the interactions between a LIBAFL QEMU fuzzer and the guest running in QEMU is displayed in Fig. 1.

Hooking System

It is important for an instrumentation engine to allow the definition of custom hooks for events such as memory operations or execution of specific instructions. In LIBAFL QEMU, hooks are inserted as QEMU helpers (introduced in II-D) with some patched code in the QEMU fork to produce helpers on the fly, and thus these hooks are functions with a C ABI. There are several types of hooks in place:

- Instruction hooks: as the name suggests, to hook a specific instruction given its address;
- Blocks hooks: to run code before the execution of each basic block in the target;
- Edges hooks: to run code between two basic blocks, for instance, to log the execution of an edge in the CFG. In detail, it is implemented by emitting an intermediate block when chaining [1] two blocks with more than one exit;
- Read and write hooks: executed every memory read or write;
- Comparisons hooks: executed before every comparison instruction, carrying information about the operands;
- Thread creation hook: triggered when a new thread is spawned in user mode;
- Syscalls and post-syscalls hooks: they are triggered before or after syscalls in user mode and can be used as filters;
- Crash hooks: to hook crashes in the virtual CPU in user mode;

Specific events happening in the guest trigger hooks that are executed as callbacks without changing the execution flow of the program. Thus, they can trigger breakpoints and suspend the guest execution to give control back to the host-side harness.

To communicate with the host to inform the fuzzer of some actions to perform, such as the memory ranges that must be considered for instrumentation, we created a custom instruction: the "**backdoor**". We crafted an instruction that is invalid for all the architectures supported by QEMU. QEMU interprets it as a hypercall and uses it to trigger the backdoor hook in the host.

In addition to the backdoor instruction, we also developed a similar custom instruction, the **sync backdoor**. While the

[1] https://www.qemu.org/docs/master/devel/tcg.html#direct-block-chaining

backdoor instruction makes it possible to run Rust code during the execution of the VM, the sync backdoor stops the VM and makes the emulator return to the host-side harness. This difference is necessary because not every action can be performed while the VM is running, e.g., saving and restoring a snapshot.

Execution Control

The wide variety of targets in both user mode and system mode makes harnessing a non-trivial problem to tackle: the general memory layout can significantly change from one architecture to another, some systems work with complex virtual memory systems, the final memory layout can be randomized at runtime, etc.

When designing how the execution of the guest should be controlled by the fuzzer, we considered 3 different scenarios and developed 3 execution control systems for LIBAFL QEMU:

- The user knows in advance at which instruction address fuzzing should start and stop and thus the fuzzer can tell the emulator where to give back the control. In this case, we propose the use of **breakpoints** to harness the target;
- The addressing model is more complex, and the instruction address at which fuzzing should start and stop cannot be easily determined before running the target. This is commonly the case in systems that randomize the virtual address space of both the kernel and processes at runtime (Linux, Windows, MacOS, etc.). Kernel fuzzing perfectly illustrates this situation where it is usually easy to integrate custom (userland) code but difficult to determine precisely at which address this code will be located. In this case, the guest must inform the emulator to stop and give back control to the fuzzer. This is the use case of the **sync backdoor**, a version of the backdoor designed to stop the target, while backdoors execute code without going out of the VM. It makes it possible to perform operations that can only be done when QEMU does not run, like snapshot and restore;
- The user wants to dynamically explore the target with a debugger, and control the fuzzer within the debugging context as a usability feature. To allow this scenario, LIBAFL QEMU offers an extensible interface to add **custom monitor** commands[2] that can be called from GDB. This allows the user to directly issue commands to the fuzzer such as snapshot or start fuzzing from the debugger;

All of these are different ways to do the same thing: drive the fuzzer's target. We developed a generic operation handling mechanism that makes it possible to create commands handled similarly, regardless of the method used. We also make available a helper C header file reusable in different harnesses, making it possible to call a sync backdoor with few lines of code. For now, x86, x86_64 and ARM architectures are supported by this helper header file, but we plan to port it to the remaining target architectures supported by QEMU. However, the rest of LIBAFL QEMU features fully work with the 7 aforementioned architectures.

The design of the breakpoint system is different between user mode and system mode as the design of QEMU differs a lot in how the virtual CPUs are executed.

In user mode, we exploit the exception system in TCG adding a custom exception for our breakpoints. When a breakpoint is inserted, the block of the target instruction is retranslated with a callback on the instruction that jumps outside the CPU loop. Then, for every architecture-specific CPU loop, we return to the caller – `Emulator::run` in most cases – if the exception is our custom breakpoint exception. While being extremely fast, in user mode, breakpoints are thread-local. For system mode, as there are at least two threads, one for the CPUs and one for the events, we reuse the system debug events in QEMU to append a breakpoint event to the event thread that will be processed as soon as possible and issue a VM stop command to freeze all the CPUs of the virtual machine.

C. High-Level API

The high-level API is an abstraction integrated with LIBAFL built on top of the low-level ones. Using the `QemuHooks` object hooks can be registered as Rust functions or closures, not just C ABI functions, and have the fuzzer state as a parameter.

These hooks can alter the LIBAFL State, which is persistent across crashes and timeouts, to store relevant information for the instrumentation. For instance, an edge coverage hook can store the mapping between the addresses describing an uncovered edge in the CFG to the unique ID used to log the execution of such edge in the coverage map.

These hooks also make it possible to interact with the instrumentation helpers, which we will introduce in the following section.

Instrumentation Helpers

The purpose of the instrumentation helpers, represented with the `QemuHelper` trait in the library, is to group hooks and data related to the same implementation of specific instrumentation, for instance, call stack logging or an emulated filesystem hooking the relevant system calls.

These helpers live in the `QemuHooks` object in a compile-time tuple list that enables queries by type with a compile-time cost only, like the observers in the core library of LIBAFL [23]. High-level hooks must query the correct helper they are made for and use it like any other Rust method using the `self` parameter.

D. Fuzzing oriented capabilities

On top of the previously discussed API, LIBAFL QEMU offers some ready-to-use components and instrumentations with fuzzing capabilities. Here we introduce them starting from the executor to the snapshot capabilities in both user mode and system mode.

The QemuExecutor

The `Executor` component responsible for running the test harness is an extension of the LIBAFL's `InProcessExecutor` [3]. This executor keeps the original

[2]https://qemu-project.gitlab.io/qemu/system/monitor.html

[3]https://aflplus.plus/libafl-book/core_concepts/executor.html#inprocessexecutor

timeout handler to catch `SIGALRM` and restart the fuzzing process on timeouts as LIBAFL does with source-based instrumentation executors. The difference is that this executor notifies the helpers when the first execution is about to be performed – so that helpers can do pre-harness initialization things such as taking a memory snapshot of the guest process – and the crash handler, in case of user mode, is kept the original QEMU handler as QEMU uses `SIGSEGV` to execute RWX guest code and so not every crashing signal is a real crash for the fuzzer. In this case, the executor installs a crash hook that is called from the emulator context and not from a signal handler allowing both the execution of RWX code and getting only true positive crashes.

Another executor offered by the library is the `QemuForkExecutor` that forks the process before each fuzz case, making it resistant to destructive changes such as memory exhaustion in a simple but inefficient way, superseded by the snapshot helper.

Coverage Tracking

Coverage tracking is one of the important features in coverage-based fuzzing and requires dedicated instrumentation. Several helpers in LIBAFL QEMU implement different strategies.

One helper provides the classic edge coverage using AFL-like instrumentation with a fixed-size map. It instruments each block and keeps track of the previously executed block to create a hash of the edge (the two addresses used) to index a fixed-size coverage map.

To avoid collisions [10], another helper exists that makes use of edge hooks and assigns at each new translated edge a unique ID. This allows not only collision-free edge coverage but also to have a dynamically growing coverage map, with a large performance gain in the case of small targets.

This helper has, however, some limitations: (1) like normal source-level collision-free edge coverage (with *SanitizerCoverage* [31]) it cannot track indirect branches; (2) it cannot track cross-pages branches in system mode because pages can be remapped by the emulated MMU. Therefore, the AFL-style edge coverage is preferred for system mode.

CmpLog for Instructions and Routines

CmpLog is an instrumentation introduced by WEIZZ [20] to log comparison instructions and routines and later used in AFL++ to implement REDQUEEN [6], which originally instrumented the target via breakpoints. LIBAFL offers several observers to collect the values used inside comparison instructions and as parameters in comparison functions and some of them are employed for source-based fuzzing.

LIBAFL QEMU implements two helpers that bridge the LIBAFL observers with an instrumentation of the guest code to log these values. The helper that logs the instructions registers a comparison hook and fills the CmpLog map in a very straightforward way calling the same routines offered by LIBAFL for source-level fuzzing. The other one, made to log the parameters of the routines, is quite more complex.

Our library does not offer a direct hooking system for function calls as they are architecture-specific constructs. To do so, this helper installs a block generation hook executed before the translation of each basic block and scans the instructions in the block, using the *capstone* [5] disassembler, to find call instructions and then installs an instruction hook on such addresses. The instruction hooks are executed before the call instruction and they are responsible for logging the parameters, following the calling convention described in each architecture-specific module.

Callstack Tracing

The very same method used for comparison routines logging is used by another fuzzing-oriented helper in LIBAFL QEMU, the call stack tracer. In addition to call instructions, it scans also for return instructions and it adds additional instruction hooks to log the return address.

This tracer supports several subcomponents that define how to use the collected call stack, for instance for context-sensitive coverage [10], [12] or for debugging purposes.

User mode: Binary-only AddressSanitizer

AddressSanitizer [54] is a compiler-based instrumentation to catch silent memory corruptions introduced by LLVM. It instruments the creation of memory objects (on heap, stack, globals and more) and checks the validity of every memory access using a shadow memory.

To achieve a similar goal on binaries, AFL++ employs QASan [21] to catch heap-based silent bugs. QASan is composed of a runtime library that is injected in the target that hooks the allocator to report the heap objects and various common functions to insert validity checks. The other part of QASan lives in the QEMU code, keeping track of the validity of the objects thanks to the shadow memory while getting updates from the guest via a custom syscall.

In user mode LIBAFL QEMU, we implemented a QASan runtime in Rust and an instrumentation helper that inserts validity checks at each memory load and store. We utilize the very same runtime library of AFL++, while reimplementing the host code in Rust instead of a patch to QEMU itself.

Support for system-wide sanitization is planned for the future, alongside other sanitizers such as MSan [55].

User mode: Memory Snapshots

In-process fuzzing targets, such as the LIBFUZZER [32] harnesses, must be stateless so that each execution of the target function is independent of the previous. In binary-only targets, this is often impossible as the stateful code cannot be removed or changed, and so it is common practice to use a snapshot primitive in this case instead of simply iterating over the harness.

As previously introduced, LIBAFL QEMU offers the fork executor that, like AFL, uses fork to snapshot the state of the host and guest at the same time as a simple snapshot primitive. This is, as shown by Xu et al. [61], a roadblock for scalability and cannot work with multiple threads.

To cope with this issue, we created a memory snapshot helper for user space targets. It works by instrumenting every memory store operation to track dirty pages with an internal tree data structure and every memory mapping-related syscall

to maintain an overview of what changed during execution. To roll back the memory snapshot, newly created mappings are deleted and unmapped and dirty pages are restored. In addition, the helper can avoid unmapping pages and cache memory mappings served by `mmap` in the next execution to speed up the process and avoid kernel interaction.

System mode: Fast VM Snapshots

As for user mode targets, system mode targets must completely reset their state to keep each run deterministic. However, as identified by Schumilo et al. [50], three different parts must be taken care of to restore the initial state of the target entirely. We decided to opt for a more modular and architecture-agnostic approach. In addition, our implementation is stack-based, allowing for fast saving and restoration of intermediate states of the target.

System Memory QEMU adopts a hierarchical model to represent the system's memory. A priority system allows QEMU to decide which subsystem should handle a read or a write at a given address. As a result, a virtual machine may possess different blocks of memory called RAM Blocks (RB), depending on the devices of the system. When fuzzing starts, each RB is completely saved and stored in a hashmap. Leveraging the SoftMMU, we can, in an architecture-independent manner, track writes to memory. A push operation saves every dirty page in a dedicated hashmap (linked to the precedent increment in a linked list) and flushes the dirty page tracker. Thus, restoring a target to a previous state consists of restoring every dirty page by going through the linked list.

Devices Full-system targets heavily rely on multiple machine-agnostic devices, implementing their internal logic. Thus, they embed a state that must be restored alongside the system memory. We leverage existing QEMU's device snapshotting in our implementation. At the core of the save & restore system resides a complex recursive function, serializing basic types, pointers, and nested structures. It also runs device-specific code before and after operations. It makes it difficult to perform a device-independent fast snapshotting system. Contrary to Nyx [50], we keep it as-is (within a lightweight wrapper), since it was never a performance bottleneck in our experiments.

Block Devices QEMU handles Block Devices (BDs) differently than other more standard devices. BDs mainly refer to hard memory devices, which are usually much bigger than the RAM available on a host machine: QCOW2 disks, raw images, etc. Therefore, treating BD snapshotting like System Memory snapshotting is not realistic. Instead, we solved the problem differently: nothing should ever be saved to BDs directly; instead, it should be written in RAM. The approach is quite similar to Nyx [50] approach for the basic idea: a hashmap stores each write to a BD to ensure data integrity. During reading, QEMU first tries to find the corresponding data in the hashmap and falls back to the BD if nothing is found. Besides the performance gains, it conveniently allows opening the BD in read-only, allowing the reuse of the same BD file for different QEMU instances. We also started implementing the push and pop features, to remain consistent with the RAM snapshot.

Once assembled, those basic blocks result in a target-independent, host-independent, lightweight full-system incremental snapshot mechanism compatible with the latest version of QEMU. We plan to make use of incremental snapshotting for potential future work [34], [52]. We developed our fast VM snapshotting system with modularity as a priority. We, therefore, consider potentially unifying Nyx's snapshotting with our implementation to also support hardware-based virtualization accelerators like KVM in the future.

IV. Practical Challenges

During the development of LibAFL QEMU we faced several practical challenges while trying to design an architecture that allows writing fuzzers in Rust while easily maintaining the QEMU fork up to date.

The first challenge was to transform QEMU, a standalone tool, into a library and use it from Rust. To do this, we modified the build system to produce a shared object instead of an ELF executable. In `libafl_qemu_build` when we invoke the configure command we replace the linker with a logger keeping track of the linker invocation and then its output is used by the library to produce a single object file containing all the QEMU code with partial linking. The object file is then included in the Rust linker command statically and the logged command line is parsed to output the correct Rust `build.rs` directives to include the needed libraries.

The other important challenge was to avoid changing too much the original QEMU codebase to keep the number of conflicts minimal when merging from a more recent commit upstream. To do so, we placed all the self-contained code such as the system snapshot and the hooks in a separate folder in the QEMU fork and marked the minimal changes to the codebase with comments encapsulating the changes that we performed to the original code. By design, we also opted to develop in Rust as much as possible, for instance, the QASan runtime is entirely located in the LibAFL QEMU crate while in AFL++ it is hardcoded as a patch to its QEMU fork.

V. Use Cases

We now show how to write fuzzers with LibAFL QEMU on two examples and compare them with AFL++ and kAFL. We only show here the code relevant to LibAFL QEMU, complete code can be found in the artifact, and the baby fuzzer with LibAFL tutorial [4] can be used as a reference.

A. Android Library Fuzzing

We first show the usage of LibAFL QEMU in user mode for the ARM64 architecture to fuzz the `libimagecodec.quram.so`, a well-known closed-source image parsing library present on Samsung devices. This library was previously fuzzed by Project Zero [47] with some custom hacks to QEMU. Later @flankerhqd [24] fuzzed it with a custom fuzzer based on Unicorn [39] and a different harness, we base our fuzzer on those ideas.

Harnessing

Using Ghidra, and information from Flanker's slides [24], we created two harnesses to fuzz the library. The first harness reads the image from a file, then gets the basic information such as image width and height using the `QuramGetImageInfoFromFile2` API and then reads the

metadata using `QrParseMetadata`. The second harness is more complex and invokes image decoding using the `QrDecodeDNGFile` routine. We compiled and linked the harnesses with the Android toolchain version r21d.

Building a Simple Fuzzer

Fuzzing with a LIBAFL QEMU-based fuzzer starts with setting up QEMU and executing until our harness executes. We load the binary, get the harness function address by parsing its symbols, set a breakpoint to the harness and, execute until execution reaches the harness. `MAGIC_FILENAME` is the name of the file we will use to write our fuzzer-generated input.

```
const MAGIC_FILENAME: &'static str = "SLASTI_MORMANTI";
const HARNESS_NAME: &str = "harnessSimple";

pub fn fuzzer() -> Result<(), Error> {
    // ...

    let mut args = vec!["qemu".into(), "./harness".into(),
                        MAGIC_FILENAME.into()];
    let mut env: Vec<(String, String)>
                 = env::vars().collect();

    let emu = Emulator::new(&mut args, &mut env)?;

    let mut elf_buffer = Vec::new();
    let elf = EasyElf::from_file(emu.binary_path(),
                                 &mut elf_buffer)?;

    let harness_ptr = elf
     .resolve_symbol(HARNESS_NAME, emu.load_addr())
     .expect(&format!("Symbol {} not found", HARNESS_NAME));
    println!("{} @ {:#x}", HARNESS_NAME, harness_ptr);

    emu.set_breakpoint(harness_ptr);
    unsafe { emu.run() };
```

After reaching the target function, using the low-level API, we read the return address and snapshot the registers' state with `CPU::save_state`. We set a breakpoint at the return address, which will be hit when the target function completes its execution.

```
    let ret_addr: u64 = emu.read_reg(Regs::Lr)?;
    println!("Return address = {:#x}", ret_addr);

    emu.remove_breakpoint(harness_ptr);
    emu.set_breakpoint(ret_addr);

    let saved_cpu_states: Vec<_> = (0..emu.num_cpus())
        .map(|i| emu.cpu_from_index(i).save_state())
        .collect();
```

We can now write the Rust harness function which: (1) writes the input to the file and, (2) restores the state of the registers. This part of the harness calls the part of the harness in the binary, using the emulator API.

```
    let mut harness = |input: &BytesInput| {
        input.to_file(MAGIC_FILENAME).unwrap();

        unsafe { let _ = emu.run() };

        for (i, s) in saved_cpu_states.iter().enumerate() {
            emu.cpu_from_index(i).restore_state(s);
        }

        ExitKind::Ok
    };
```

In order to bypass coverage roadblocks we choose to build a CmpLog-based fuzzer, we follow LIBAFL examples[4].

To execute our harness in QEMU, in the context of a LIBAFL fuzzer, we need to set up an executor with the right instrumentation options. So we create the QemuHooks object with one helper to trace collision-free edge coverage and another one to trace cmp instructions for CmpLog.

```
    let mut hooks = QemuHooks::new(
        emu.clone(),
        tuple_list!(
            QemuEdgeCoverageHelper::default(),
            QemuCmpLogHelper::default(),
        ),
    );

    let executor = QemuExecutor::new(
        &mut hooks,
        &mut harness,
        tuple_list!(edges_observer, time_observer),
        &mut fuzzer,
        &mut state,
        &mut mgr,
    )
    .expect("Failed to create QemuExecutor");
```

All the components of a LIBAFL-based fuzzer, as the reader can see in the arguments of `QemuExecutor::new`, must be created according to the techniques that we want to use, as described in the LIBAFL book [4].

By tracing syscalls (using QEMU's `-strace` option) we observed calls to `rt_sigprocmask` and a lot of unnecessary stdout output. Therefore, as a small optimization, we eliminated both syscalls using a hook and telling LIBAFL QEMU to skip those syscalls and return a provided value.

```
    hooks.syscalls(Hook::Closure(Box::new(
        |_, _, sys_num, arg0, _, arg2, _, _, _, _, _| {
            match sys_num as i64 {
                SYS_write => {
                    if arg0 == 1 || arg0 == 2 {
                        // arg2 is the `count` parameter
                        return SyscallHookResult::new(
                            Some(arg2 as u64));
                    }
                }
                SYS_rt_sigprocmask => {
                    return SyscallHookResult::new(Some(0));
                }
                _ => (),
            };
            SyscallHookResult::new(None)
    })));
```

We now have a working minimal fuzzer for this Android library that can be run setting the `QEMU_LD_PREFIX` environment variable to the device root file system from which the target library comes. This filesystem was obtained from the vendor's website. Several further optimizations are possible to increase the coverage, bug-finding performance and scalability over multiple cores.

A Better Fuzzer

For an optimized fuzzer, we want to hook the filesystem syscalls involved in the application (using `strace`) and provide the input from memory instead of using a file to avoid any expensive kernel and device interaction. To increase the ability to uncover silent memory corruptions, we add binary-only ASan, and to avoid potential memory leaks and persistent states of the application across multiple executions, we also add the memory snapshot helper. To increase the performance, we also add a filter to locate only the target library and harness memory regions instrument for coverage and comparisons only the interesting code.

[4]https://github.com/AFLplusplus/LibAFL/tree/main/fuzzers

```rust
// This calls Emulator::new with the QASan runtime preloaded
let (emu, asan) = init_with_asan(&mut args, &mut env)?;

// ...

let mut allow_list = vec![];
for region in emu.mappings() {
    if let Some(path) = region.path() {
        if path.contains("imagecodec") ||
           path.contains("harness") {
            allow_list.push(region.start()..region.end());
        }
    }
}
let filter =
     QemuInstrumentationFilter::AllowList(allow_list);

let mut hooks = QemuHooks::new(
    emu.clone(),
    tuple_list!(
        QemuEdgeCoverageHelper::new(filter.clone()),
        QemuFilesystemBytesHelper::default(),
        QemuCmpLogHelper::new(filter.clone()),
        QemuCmpLogRoutinesHelper::new(filter.clone()),
        QemuSnapshotHelper::new(),
        QemuAsanHelper::new(asan, filter.clone(),
                            QemuAsanOptions::Snapshot),
    ),
);
```

Note that the `QemuFilesystemBytesHelper` is a user-written helper for this fuzzer. It hooks the various filesystem syscalls used by the target and emulates them in memory to provide the fuzzer test case without involving the disk thus enhancing scalability. Only a few syscalls are implemented, mostly returning constants gathered from `strace`, but a generic helper emulating an entire in-memory filesystem is possible and planned.

Comparison with AFL++ QEMU

The very same harness can be used with AFL++ and its QEMU mode. In this section, we compare two similar setups built with AFL++ with our two fuzzers testing the `QrDecodeDNGFile` function with a simple DNG image as seed. Firstly, we set the AFL++'s environment variables to achieve persistent mode looping on the `harnessDecode` harness function we wrote, then in the second fuzzer we add QEMU AddressSanitizer [21] and fork-mode to replicate the user space memory snapshot capabilities of our second fuzzer.

We let the simple fuzzers in persistent mode run for 48h, the results are reported in Table I. No fuzzer found any crashing test case, but the LIBAFL QEMU fuzzer outperforms the other in terms of speed (2.86x) and block coverage over the harness and the target libraries of the saved corpus (1.28x) while having saved much fewer test cases as the collision-free coverage is less sensitive compared to the AFL++ one.

TABLE I: First user space fuzzing campaign over 48h.

Fuzzer	Block Coverage	Corpus Size	Execs/s
LIBAFL QEMU	8071	4374	6.6k
AFL++	6277	7462	2.3k

In a second round, we took the corpus produced by both fuzzers to continue the campaign, but we added the sanitization and snapshot capabilities to both. The results of another 48h campaign are reported in Table II. This time LIBAFL QEMU is 1.92x faster than AFL++, a higher but similar block coverage (1.02x) and 2 bugs were found while AFL++ found none. The corpus size is reduced when compared to the first run, even if the previous corpus was used as initial input. This is due to the downgrade in terms of speed, causing many inputs in the initial corpus to be classified as timeouts and thus not included in the fuzzer corpus.

TABLE II: Second user space fuzzing campaign over 48h.

Fuzzer	Block Coverage	Corpus Size	Execs/s	Bugs
LIBAFL QEMU	8272	4063	248	2
AFL++	8121	7855	72	0

The bugs found are one read and one write heap overflow on heap chunks respectively allocated in the DNG decoder parser and in the harness.

From this small experiment, we can devise that LIBAFL QEMU is not only on par with the state-of-the-art in user space emulation but it can be used to have an improved fuzzing performance thanks to its speed and the advanced fuzzing algorithms available in LIBAFL.

B. Windows Kernel Fuzzing

In our second test case, we fuzz the NTFS partition parser in the Windows 10 kernel. NTFS has already been successfully fuzzed in the past [51]. We tried to reproduce the same experimental setup. Thus, the harnessing methodology is close to the one used in the KAFL [51] experiment. Every experiment from this section was run on a desktop machine with an Intel® Core™ i5-1140 @ 4.40 GHz and 32GB of RAM.

Windows 10 in QEMU

We set up Windows 10 on the x86-64 architecture for fuzzing using the latest available version online (Windows 10 22H2 build 19045.3636 at the time of writing) using QEMU. To fully boot, the VM requires around 10 minutes on 1 vCPU with 8Gio for memory on our virtual machine. The harness was developed on another Windows VM and exported to the target VM. The disks used to store Windows files are used in snapshot mode to avoid polluting the disk across runs.

Fuzzing methodology

The methodology we follow to fuzz the NTFS driver used in Windows 10 is conceptually rather simple: the harness uses the fuzzing input to open a VHD (Virtual Hard Disk) that will be handled like a normal disk. Since VHDs must be opened from the filesystem to be used, we use a RAM Disk to write the fuzzing input on it to speed up the process. The newly created VHD is then opened and mounted using the win32 API. By using legitimate VHD images in the corpus, we can quickly make the fuzzer test interesting parts of the driver.

Getting a working Harness in practice

The LIBAFL harness has been developed using the sync backdoor feature. Setting breakpoints at the right location for a fully emulated Windows image is troublesome: it is necessary to retrieve the page corresponding to the harness process, resolve the physical address, and finally set the breakpoint. These steps would be hard to avoid if we could not use our harness in the VM. Since we can inject our own code, we can make use of backdoors instead.

The harnessing part of the fuzzer is simple to develop with the default exit handler: after setting up the emulator with

the desired snapshot method and exit manager, we run the *run_handle* method and let it do all the low-level work for us.

```
let emu_snapshot_manager = FastSnapshotBuilder::new(false);
let emu_exit_handler:
     StdEmuExitHandler<FastSnapshotBuilder> =
     StdEmuExitHandler::new(emu_snapshot_manager); //
     Create an exit handler. We choose the default one.
let emu = Emulator::new(&args, &env, emu_exit_handler)?;

match emu.run_handle(input) { // Run and get the result
     given by the exit handler.
    Ok(handler_result) => match handler_result {
        HandlerResult::EndOfRun(exit_kind) => exit_kind, //
     The current run finished
        HandlerResult::Interrupted => { // Interrupted by
     some signal.
            std::process::exit(0);
        },
        _ => panic!()
    },
    Err(handler_error) => panic!() // The exit handler
     returned an error
}
```

Most of the time, returning from an emulation run handled by an exit handler will terminate the current run and return an exit kind to the fuzzer (e.g., normal exit, crash, timeout). The sync backdoor provides a higher-level and rusty way to interact with the VM and removes the breakpoint's boilerplate code.

The default exit handler (replaceable with a custom one) will take care of every command sent by the target-side harness (including snapshot management, paging filtering).

From the target harness point of view, the use of the exit is rather simple: import the LIBAFL header file containing all the necessary code to execute a sync backdoor and use the API as desired.

The target side of the harness communicating with the fuzzer adds a lightweight code overhead.

```
int main() {
  // Target-specific initialization code.
  target_init();

  // Signal to LibAFL the harness will start from now on.
  // Snapshot also happens here.
  uint64_t len = _libafl_exit_call2(LIBAFL_EXIT_START_VIRT,
     (UINT64) input, INPUT_MAX_SIZE);

  // Run the target and report the outcome (either ok or
  // crash).
  if (target_run(input, len)) {
    _libafl_exit_call1(LIBAFL_EXIT_END, LIBAFL_EXIT_END_OK);
  } else {
    _libafl_exit_call1(LIBAFL_EXIT_END,
     LIBAFL_EXIT_END_CRASH);
  }
}
```

This harnessing method makes things simpler whenever it is possible to run custom code in the VM. If the target cannot be modified easily or the start address can be easily identified, the breakpoint method may be a better option.

The same exit handler and the same commands can be used both for synchronous backdoors and breakpoints. Thus, we decorrelate handling logic from the harnessing method, making most of the code reusable.

The *target_init* function mainly sets things up to prepare the RAM Disk. It also sends the address range used for feedback to the fuzzer: we provided a set of Windows drivers from which we want to collect the feedback (e.g., *Ntfs.sys*, *partmgr.sys*, *mountmgr.sys*), merged the address ranges at which these modules virtually reside into one bigger range and added it in the allowed address range.

The target code running in the VM creates a file on the RAM Disk, writes the fuzzer's payload on it and tries to mount the VHD and assign it a drive letter.

```
/// Target code. Simplified for brevity.
void target_run(uint8_t* buf, uint64_t len) {
  DWORD lenWritten;
  HANDLE vhdHandle;
  VIRTUAL_STORAGE_TYPE storageType = ...;
  OPEN_VIRTUAL_DISK_PARAMETERS openParameters = ...;
  WCHAR physicalPath[MAX_PATH];
  DWORD pathSize = MAX_PATH;

  // Create Virtual Disk file.
  HANDLE hVhdFile = CreateFile(TARGET_VHD_PATH_W,
     GENERIC_READ | GENERIC_WRITE, ...);

  // Write payload in newly created file
  WriteFile(hVhdFile, buf, (DWORD) len, &lenWritten, NULL);

  // Open the file as Virtual Disk.
  OpenVirtualDisk(&storageType, TARGET_VHD_PATH_W,
     VIRTUAL_DISK_ACCESS_ATTACH_RW |
     VIRTUAL_DISK_ACCESS_GET_INFO,OPEN_VIRTUAL_DISK_FLAG_NONE,
     ..., &vhdHandle);

  // Attach to the opened Virtual Disk.
  AttachVirtualDisk(vhdHandle, ...);

  // Get the path of the new disk
  GetVirtualDiskPhysicalPath(vhdHandle, &pathSize,
     physicalPath);

  // Define the service
  DefineDosDevice(DDD_RAW_TARGET_PATH, L"R:", physicalPath);

  // Detach the disk
  DetachVirtualDisk(vhdHandle,
     DETACH_VIRTUAL_DISK_FLAG_NONE, 0);

  // Cleanup
  CloseHandle(vhdHandle);
}
```

Listing 1: Harness code for the Windows Driver.

Fuzzer

The LIBAFL [23] framework makes it easy to assemble our QEMU harnessing techniques with existing well-tested fuzzing logic. The feedback, as discussed above, can consist of either block or edge coverage. It is also possible to instrument the TB and the edges with the hook interface in Rust or by injecting it in the TCG IR directly.

The rest of the fuzzer's configuration and code are provided by default in LIBAFL.

Challenges of Windows 10 Fuzzing

Fuzzing the Windows NTFS Kernel Module, and more generally, Windows Kernel Fuzzing, comes with various challenges.

- **Toolchain** Although a Linux-compatible toolchain for Windows compilation exists [2], it is not an ideal drop-in replacement for Visual Studio. Indeed, some parts are outdated (like the *virtdisk.h* header file) and thus cannot compile any possible target without changing the toolchain itself (with the additional bugs it could introduce). We thus decided to keep Visual Studio to develop the harness. It makes developing the target incrementally complicated and requires extra work.

- **Windows low-level API** Windows is infamous for its difficult-to-use low-level API [40], but provides instead a very stable DLL API. Indeed, finding which system calls perform the desired action was not obvious and required multiple attempts.
- **Setup with QEMU-TCG** Setting up Windows for TCG is also non-trivial. Unfortunately, using a Virtual Machine set up with QEMU-KVM did not work on our side, forcing us to install everything with TCG (and its inherent slowness). It did not work on the first attempt, and multiple tests were necessary until we could use Windows correctly.
- **Getting Kernel feedback address range** Determining which address range should be used as feedback was also challenging. Since KAFL [51] does not support many ranges, we compressed everything into one big address range for both experiments (for fairness). We also had to make some guesses on which kernel modules were involved in installing a VHD. It is not a problem for our experiment objective (showing it works as intended and is not too slow), but it becomes critical for more ambitious fuzzing campaigns.

Comparison with KAFL

We ran the previously presented fuzzer for 10 hours on 1 core and compared it with the state-of-the-art for this kind of fuzzing, KAFL [51] / Nyx [50] with the latest version available as of this writing. Our main objective was to showcase we could make a complex full-system target run correctly. We compared the performances in terms of executions per second of our fuzzer in various cases with KAFL. We finally made a comparison in terms of performance between the already implemented QEMU snapshot implementation and our fast VM snapshot method. The results are reported in Table III.

TABLE III: Windows NTFS Driver campaign over 10h.

Fuzzer	Execs / Sec	Fast VM Snapshot
KAFL	0.76	/
LIBAFL	0.016	✗
LIBAFL	0.017	✓

We first note that LIBAFL QEMU can perform fewer executions per second compared to KAFL. This result was expected since KAFL makes full use of hardware virtualization thanks to QEMU-KVM, while LIBAFL QEMU runs QEMU in TCG (emulation) mode. We believe the difference remains within an acceptable margin considering this (∼50x slowdown).

Finally, we compare the concrete (average) difference between QEMU Snapshot and Fast VM Snapshot in this experiment both when the snapshot is created (Save) and when the snapshot is restored (Restoration).

TABLE IV: Windows NTFS Driver Snapshots performance.

Snapshot	Save	Restoration
QEMU SNAPSHOT	137.52s	1.877
FAST SNAPSHOT	1.36s	0.163s

The results, reported in Table IV, show that our Fast VM Snapshot implementation outperforms QEMU's snapshot mechanism by at least an order of magnitude in this experiment in both cases. Although snapshot creation performance is not so important here (since it is performed only once), snapshot restoration must be as fast as possible because it will happen at each run. We explain the small difference between QEMU snapshot and LIBAFL QEMU snapshot in Table III by the huge complexity of the target, which is most likely the performance bottleneck here. Thus, even a few seconds to restore the target does not weigh much compared with the target's runtime (∼60s). As a result, we believe our snapshot implementation makes a bigger difference for less extreme targets.

VI. CONCLUSION

In this paper, we presented LIBAFL QEMU, a new library for fuzzing with emulation based on QEMU. It is designed to allow a flexible usage of the underlying emulator to run a wide variety of targets, with ready-to-use instrumentations available for your fuzzing needs. Fuzzers written with LIBAFL QEMU are state-of-the-art and perform similarly or even better than the available tools.

We share LIBAFL QEMU as free and open-source software with the hope that it will be useful to advance the state of binary-only fuzzing with new tools built on top of this library. It can be downloaded as part of the LIBAFL repository at https://github.com/AFLplusplus/LibAFL.

Artifacts are provided to improve replicability and to be used as examples by practitioners. The associated repository is available at https://github.com/AFLplusplus/libafl_qemu_artifacts.

ACKNOWLEDGMENT

Above all, we want to thank the community around the AFL++ organization. A special thanks to the friends involved in the development of the other parts of LIBAFL and AFL++, Dominik Maier, Dongjia Zhang, Addison Crump, s1341, Marc Heuse and all the contributors of these amazing OSS projects. We want also to thank the anonymous reviewers and Slasti Mormanti for their suggestions to improve the final version of this paper. This work has benefited from a government grant managed by the National Research Agency under France 2030 with reference "ANR-22-PECY-0009". This work was supported in part by Beyond5G project. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA8655-20-1-7048. Finally, part of this work was performed while one of the authors was an intern at Intel Labs.

REFERENCES

[1] "Foreign function interface," https://doc.rust-lang.org/nomicon/ffi.html, [Online; accessed March 6, 2024].

[2] "MinGW-w64 - a complete runtime environment for gcc & llvm for 32 and 64 bits windows," https://www.mingw-w64.org/, [Online; accessed March 6, 2024].

[3] "Qiling Framework," [Online; accessed March 6, 2024]. [Online]. Available: https://qiling.io/

[4] Andrea Fioraldi and Dominik Maier and s1341 and Dongjia Zhang and Addison Crump, "A Simple LibAFL Fuzzer," https://aflplus.plus/libafl-book/baby_fuzzer.html?highlight=Fuzzer#a-simple-libafl-fuzzer, 2020, [Online; accessed March 6, 2024].

[5] Q. N. Anh, "Capstone: Next generation disassembly framework," https://www.blackhat.com/us-14/briefings.html#capstone-next-generation-disassembly-framework, [Online; accessed March 6, 2024].

[6] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[7] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[9] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 03, pp. 79–86, may 2021.

[10] P. Borrello, A. Fioraldi, D. Cono D'Elia, D. Balzarotti, L. Querzoni, and C. Giuffrida, "Predictive context-sensitive fuzzing," in *NDSS 2024, Network and Distributed System Security (NDSS) Symposium, 26 February-1 March 2024, San Diego, CA, USA*, San Diego, 2024.

[11] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, nov 2012. [Online]. Available: https://doi.org/10.1145/2382553.2382554

[12] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[13] S. Chen, Z. Lin, and Y. Zhang, "SelectiveTaint: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1665–1682. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-sanchuan

[14] Z. Chen, S. L. Thomas, and F. D. Garcia, "Metaemu: An architecture agnostic rehosting framework for automotive firmware," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 515–529. [Online]. Available: https://doi.org/10.1145/3548606.3559338

[15] M. Chesser, S. Nepal, and D. C. Ranasinghe, "Icicle: A re-designed emulator for grey-box firmware fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 76–88. [Online]. Available: https://doi.org/10.1145/3597926.3598039

[16] A. Di Federico, M. Payer, and G. Agosta, "Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 131–141. [Online]. Available: https://doi.org/10.1145/3033019.3033028

[17] A. Dinaburg and A. Ruef, "Mcsema: Static translation of x86 instructions to llvm," in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[18] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1497–1511.

[19] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 839–850. [Online]. Available: https://doi.org/10.1145/2508859.2516697

[20] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[21] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[22] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[23] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[24] Flanker, "3rd Real World CTF: Blowing the cover of android binary fuzzing," https://www.youtube.com/watch?v=y05uja2o6GE, 2021, [Online; accessed March 6, 2024].

[25] N. Hasabnis and R. Sekar, "Lifting assembly to intermediate representation: A novel approach leveraging compilers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 311–324. [Online]. Available: https://doi.org/10.1145/2872362.2872380

[26] R. Henderson, "[Qemu-arm] [PATCH 00/17] target/arm: Implement ARMv8.5-MemTag," https://lists.gnu.org/archive/html/qemu-arm/2019-01/msg00182.html, [Online; accessed March 6, 2024].

[27] E. F. (https://www.labri.fr/perso/fleury/), "Why is disassembly not an exact science?" Reverse Engineering Stack Exchange, uRL:https://reverseengineering.stackexchange.com/a/15616 (version: 2024-01-05). [Online]. Available: https://reverseengineering.stackexchange.com/a/15616

[28] S. Huster, M. Hollick, and J. Classen, "To boldly go where no fuzzer has gone before: Finding bugs in linux' wireless stacks through virtio devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 24–24. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00024

[29] Imperas, "Ovpsim," https://www.ovpworld.org/technology_ovpsim, [Online; accessed March 6, 2024].

[30] T. N. Jesse Hertz, "TriforceAFL," https://github.com/nccgroup/TriforceAFL, [Online; accessed March 6, 2024].

[31] LLVM, "SanitizerCoverage," https://clang.llvm.org/docs/SanitizerCoverage.html, [Online; accessed March 6, 2024].

[32] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, Sep. 2018, [Online; accessed March 6, 2024].

[33] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[34] D. Maier, O. Bittner, M. Munier, and J. Beier, "Fitm: Binary-only coverage-guided fuzzing for stateful network protocols," in *Workshop on Binary Analysis Research (BAR), 2022*, 2022.

[35] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/maier

[36] D. Mihocka, S. Shwartsman, and I. Corp, "Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure," 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:15838836

[37] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, San Diego, United States, 02 2018. [Online]. Available: http://www.eurecom.fr/publication/5417

[38] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[39] A. Q. Ngyuen and H. V. Dang. (2020) Unicorn: Next generation cpu emulator framework. [Online]. Available: http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf

[40] D. Nisi, "Unveiling and mitigating common pitfalls in malware analysis," Ph.D. dissertation, 2021, thèse de doctorat dirigée par Dacier, Marc Informatique, télécommunications et électronique Sorbonne université 2021. [Online]. Available: http://www.theses.fr/2021SORUS528

[41] Oracle, "Oracle vm virtualbox," https://www.virtualbox.org/, [Online; accessed March 6, 2024].

[42] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: A practical binary optimizer for data centers and beyond," 2018.

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[44] E. Pauley, G. Tan, D. Zhang, and P. McDaniel, "Performant binary fuzzing without source code using static instrumentation," in *2022 IEEE Conference on Communications and Network Security (CNS)*, 2022, pp. 226–235.

[45] M. Prasad, "A binary rewriting defense against stack-based buffer overflow attacks," in *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. San Antonio, TX: USENIX Association, Jun. 2003. [Online]. Available: https://www.usenix.org/conference/2003-usenix-annual-technical-conference/binary-rewriting-defense-against-stack-based

[46] M. Probst, "Dynamic binary translation," in *UKUUG Linux Developer's Conference*, vol. 2002, 2002.

[47] Project Zero, "MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface," https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmage.html, 2020, [Online; accessed March 6, 2024].

[48] R. C. O. Rocha, D. Sprokholt, M. Fink, R. Gouicem, T. Spink, S. Chakraborty, and P. Bhatotia, "Lasagne: A static binary translator for weak memory model architectures," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 888–902. [Online]. Available: https://doi.org/10.1145/3519939.3523719

[49] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[50] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[51] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, pp. 167–182.

[52] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22, 2022.

[53] L. Seidel, D. Maier, and M. Muench, "Forming faster firmware fuzzers," in *USENIX 2023*, 2023.

[54] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USENIX Association, 2012, p. 28.

[55] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. USA: IEEE Computer Society, 2015, p. 46–55.

[56] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, p. 48–56, may 2005. [Online]. Available: https://doi.org/10.1109/MC.2005.163

[57] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere, "Diablo: a reliable, retargetable and extensible link-time rewriting framework," in *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, 2005, pp. 7–12.

[58] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 331–340. [Online]. Available: https://doi.org/10.1145/2818000.2818017

[59] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 157–168. [Online]. Available: https://doi.org/10.1145/2382196.2382216

[60] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Machine Learning and Knowledge Discovery in Databases*, D. Gunopulos, T. Hofmann, D. Malerba, and M. Vazirgiannis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 522–536.

[61] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[62] S. B. Yadavalli and A. Smith, "Raising binaries to llvm ir with mctoll (wip paper)," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 213–218. [Online]. Available: https://doi.org/10.1145/3316482.3326354

[63] Y.-P. You, T.-C. Lin, and W. Yang, "Translating aarch64 floating-point instruction set to the x86-64 platform," in *Workshop Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP Workshops '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3339186.3339192

[64] M. Zalewski, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016, [Online; accessed March 6, 2024].

[65] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 659–676.