QLEE: Symbolic Execution of QEMU Devices with KLEE

Abstract

Hypervisor bugs cause security vulnerabilities and incompatibility issues. While effective, existing hypervisor fuzzing methods are inefficient for targeting a single peripheral due to the overhead of spawning QEMU processes and resetting virtual machine states after crashes. This project introduces QLEE, which leverages the symbolic execution engine KLEE to detect programming errors in peripheral emulators. By utilizing peripherals' operational constraints (e.g., valid value ranges), QLEE reduces false positives reported by KLEE. Running QLEE on ten entry points in three device emulators from QEMU found a (potentially benign) out-of-bounds access in QEMU's PL011 UART emulator.

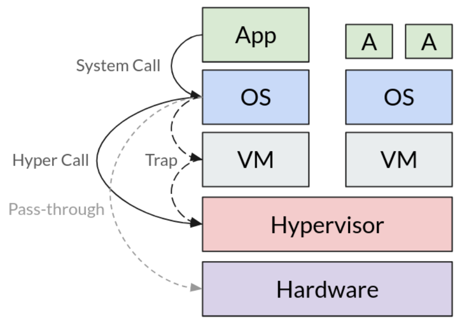

Figure 1: OS-Hypervisor Interaction

1 Introduction

Hypervisors, a.k.a. Virtual Machine Managers (VMMs), allow multiple Virtual Machines (VMs or *guests*) to run on the same physical *host* hardware. Each VM behaves as if it were an independent "physical" machine, and the host's resources, such as memory and peripherals (e.g., storage, network device, etc.), are *virtualized* by the VMM and exposed as isolated resources to each VM. The VMM is, therefore, responsible for multiplexing the underlying host hardware amongst the guest VMs. The VMM either *emulates* peripherals when the device is not physically available and thus the behavior of the device is implemented in software or allows *pass-through* access to the host's hardware. Figure 1 illustrates the different methods of OS-hypervisor interactions.

Hypervisors are prevalent in the cloud, where customers can rent VMs to run their workloads. Since multiple VMs are co-located on the same host, the hypervisor must provide strong isolation between VMs. Bugs in hypervisors could be exploited to break the isolation between VMs, leading to security vulnerabilities.

Emulation platforms such as QEMU are used for software development and prototyping, providing easier debugging and faster development than hardware prototype boards. However, inaccurate emulation of target hardware could result in incompatibility and unexpected behavior when the software is run on actual hardware.

Thus, errors or inaccuracies in the representation and behavior of the emulated hardware could have two-fold effects: they can expose vulnerabilities that break isolation between different VMs or cause incompatibility bugs where software developed on VMs does not run properly on physical hardware.

Prior work in hypervisor fuzzing [3–5] relies on spawning a QEMU VM to perform their fuzzing. This introduces overhead when performing the fuzzing since each access to any MMIO or PIO peripheral will be trapped by QEMU before being passed on to the peripheral emulator. Furthermore, the overhead of resetting the state after each crash takes time out of fuzzing, and most of the state is irrelevant to the peripheral.

Directly targeting the device implementations eliminates this overhead. This project introduces QLEE, enabling fine-grained automated testing of emulated peripherals by leveraging KLEE, a symbolic execution (SE) engine. Hardware vendors define a peripheral device's operational constraints, which can be incorporated to add constraints to the symbolic execution performed by KLEE, reducing false positives reported. QLEE has a multi-step pipeline to prepare and run KLEE on the device emulator source.

QLEE was evaluated on ten target functions in three devices. Though KLEE emitted thousands of test cases for different paths, the errors reported were less than five for each target. Almost all of the errors were false positives identified, with only one error in PL011 being an out-of-bounds access; however, this error is potentially benign.

2 Background

2.1 Quick Emulator (QEMU)

The Quick Emulator (QEMU) [1] is a Virtual Machine Monitor (VMM) and emulator. When paired with the Kernel-based Virtual Machine (KVM) module, which provides virtualization primitives in the Linux kernel, QEMU can function as a VMM to manage virtual machines (VMs) with either virtualized or emulated peripherals, such as serial consoles, storage devices, and networking interfaces.

QEMU provides two modes of emulation:

- *User-space* enables running user-space binaries designed for architectures different from the host hardware.
- *Full-system* emulation provides a complete guest platform with peripherals, supporting both same-architecture and cross-architecture configurations.

This project focuses on QEMU's full-system emulation.

To enable cross-architecture emulation, QEMU employs Just-In-Time (JIT) compilation, translating the guest's basic blocks into the host's architecture for execution. Additionally, QEMU provides the guest platform with emulated peripherals, as these are essential for the guest's functionality but are unavailable on the host system.

Additionally, QEMU provides testing facilities via its *qtest* component, instantiating VMs and directly interacting with them without a guest OS running on the VM.

2.2 Symbolic Execution with KLEE

KLEE [2] is a symbolic execution engine targeting LLVM IR. KLEE interprets the LLVM IR *symbolically* while tracking constraints over the state of the program (e.g., $x > 2$), called the *path condition*.

On reaching a "dangerous" operations (e.g., division, pointer dereference, array accesses), the path constraint and a safety constraint (e.g., the denominator cannot be zero) are passed in to an external solver to check whether the path constraints can be solved such that the safety constraint is violated, i.e., an error is triggered. If so, KLEE will produce a concrete test case for this error.

3 Related Work

The following discusses the previous work in hypervisor fuzzing, all targeting QEMU. Note that all of the following require a running QEMU VM to perform the fuzzing. Furthermore, HYPER-CUBE and NYX only support the *x86* architecture.

3.1 VDF

VDF [3] uses a record-and-replay approach combined with evolutionary fuzzing via AFL. To prepare for fuzzing, instrumentation code is manually added to a device's callback functions. This code is specifically for recording accesses to the device and is not used for coverage tracking.

The recording phase happens as the device is initialized and accessed "normally" by the BIOS and the OS. Recorded accesses are broken into two groups:

- an *init* set used to initialize the device
- a *seed* set used as the initial corpus and mutated.

QEMU's qtest facility consists of a test driver process, which in turn spawns a target QEMU VM process to perform the test. To remove the IPC overhead between the driver and the target VM processes, VDF implements a new accelerator that integrates the driver capabilities into the qemu accelerator used inside the VM and thus only needs a single VM process to perform the testing.

VDF uses AFL to mutate the seed set, and a VM instance runs each input; however, the details of how this is done are not clear[1]. VDF adds a new compile flag to AFL's compiler, so only the files compiled with that flag have branch coverage instrumentation. Each branch is given a unique ID, and a bitmap of the branches is constructed and used for coverage tracking and guidance. Once an error occurs, the test case is logged, trimmed to the shortest possible set of accesses, and finally reported.

3.2 HYPER-CUBE

HYPER-CUBE [4] improves upon VDF by enabling "multidimensional" fuzzing. While VDF can only interact with memory and port-mapped IO, HYPER-CUBE can simultaneously interact with all available interfaces (e.g., DMA).

HYPER-CUBE consists of three components:

- *HYPER-CUBE OS* enumerates the devices, sets up interrupts, and handles memory management
- *Tesseract* is an interpreter which runs in HYPER-CUBE OS and executes arbitrary bytecode input
- Host-side external tools for logging, test-case minimization, and decompiling bytecode to C

HYPER-CUBE boots a regular OS on a VM, which provides complete control over the environment. Furthermore, using a lightweight custom OS instead of a commercial OS such as Linux enables fast reboots after a crash or hang. After booting, Tesseract is fed with input bytecode, which encodes a sequence of hypervisor interactions, such as when accessing a device. The fuzzing input is either provided manually or generated using a pseudo-random number generator and fuzzing is done blindly, i.e., without mutations to previous inputs. When an error is detected, the bytecode input is logged and then minimized. Finally, the decompiler will turn the bytecode to C for further manual inspection.

[1]Section "Playback of Virtual Device Activity" in [3]

3.3 NYX

NYX extends HYPER-CUBE with coverage guidance. NYX is based on nested virtualization, where a guest OS (L2), in this case a modified HYPER-CUBE OS, acts as the fuzzing agent, targeting a guest hypervisor (L1) provisioned on a host hypervisor (L0). To gather coverage data, NYX utilizes Intel Processor Trace (Intel-PT), a hardware feature that captures information about software execution. Prioritizing new coverage based on the gathered traces, NYX's mutation engine incorporates affine types and DAG-based bytecode to avoid API misuse during mutations. Additionally, NYX uses a device specification that describes the semantics of the bytecode and the shape of the generated programs with respect to the memory structure, interactions, and constraints for a given device. To improve throughput, NYX implements a fast snapshot mechanism leveraging dirty page tracking to accelerate VM resets. In KVM-based nested virtualization, hypercalls from the guest OS (L2) are first intercepted by the host hypervisor (L0) before being passed to the guest hypervisor (L1) when appropriate. This allows for implementing special-purpose hypercalls for facilitating fuzzing and also selectively forward hypercalls from the agent to the target hypervisor.

4 Methodology

4.1 Design

The main goal of QLEE is to automatically test the device emulators without running QEMU VMs, reducing the fuzzing overhead and enabling fine-grained testing. To this end, QLEE is based on symbolic execution, specifically the KLEE symbolic execution engine (Section 2.2). The main advantage of symbolic execution is the ability to execute multiple program paths at once without resetting any state or mutating inputs.

Specification File. QLEE relies on a user-provided specification that contains the path to the target source, variable declarations, variable constraints and assignments, and the target entry points (see Listing 6 in Appendix). Symbolic values represent the set of all possible values for a variable, which is an over-approximation of the program state and thus results in false positives. The constraints and assignments in the specification narrow the program state to valid states during constraint solving, reducing the false positive rate. The user can find the proper constraints on the device state from the programmer's manual of a device or by inspecting false positives.

Harness Generation. KLEE requires an entry point from which to start the execution. However, the function arguments must be constructed and marked as symbolic before an entry point can be called. This is particularly important for the `state` structure encapsulating the device state. Since structures are passed to function via pointers, executing the entry point without a harness would cause KLEE to solve for the pointer value and not the values constituting the structure, consistently leading to `null` pointer errors. Additionally, the harness is required for associating the constraints and assignment with the function arguments *before* calling the target function. Therefore, each entry point defined in the specification has a corresponding generated harness. QLEE appends the generated harnesses to the end of the device source file specified in the specification as C code. The device source, along with the harnesses, are then compiled to LLVM.

Stub Generation. Since device emulators in QEMU rely on QEMU's internal facilities to interact with their environment, they need to be linked with the libraries provided by QEMU. However, this requires significant changes to QEMU's build system to build the libraries as LLVM IR. Instead, QLEE has a stub generation component called AutoStub, which, given an LLVM IR file, finds all external functions and global variables and generates a new LLVM IR file with the global variables declared and initialized to their default value (e.g., null or 0), and the external functions defined as stubs returning the default value of their return type. The stub IR file is passed into KLEE (via `--link-llvm-lib`) to be linked against the target device IR file.

Running KLEE. KLEE is called on each harness after the harness and stub generation phases and reports any errors it finds.

4.2 Implementation

The implementation for QLEE can be found at https://github.com/sepehr541/QLEE.

QLEE. QLEE is written in Python and is a set of scripts that:

(1) parse the specification & compile_commands.json
(2) generate the harnesses & append to source
(3) compile the source to LLVM IR
(4) call AutoStub for stub generation
(5) call KLEE on each harness

QLEE takes in a few parameters: the path to the QEMU source root, the path to AutoStub, the path to the specification, and the path to use for KLEE's output directory. Given all the parameters, QLEE will execute all the steps automatically. Figure 2 illustrates an overview of QLEE's pipeline.

Specification. The specification is written in YAML[2]. YAML provides easy definition of key-value maps and lists. Additionally, YAML allows for referencing nodes via its anchor (`&`) and alias (`*`) operators, significantly reducing duplication. An anchor can be declared (e.g., `&state`) for a node and referenced later (e.g., `*state`). The constraints and assignments in the specification are written as strings and then transformed into valid C code using simple string manipulation. The entry points in the specification consist of the name of the target function in the device source and a list of arguments defined as references to variables previously declared using the alias operation.

Harness Generation. This is done using string formatting, which builds a C function that declares the variables locally, calls KLEE's `klee_make_symbolic` function to mark them as symbolic values, calls the `klee_assume` function to associate additional constraints to variables with symbolic values, and calls the target function with the variables passed in as the arguments. The generated C function is defined as `static` and is annotated with a `used` attribute to suppress warnings from the compiler.

AutoStub. AutoStub is implemented as a C++ program which leverages LLVM's IR libraries [3] for C++. Thus, finding the global variables and external functions and generating stubs are greatly simplified by reading and generating LLVM IR directly. This was intentionally not done at the source level to avoid pre-processing `#include` directives and filtering the declarations to those referenced within the source file.

compile_commands.json This file is produced by monitoring a complete build of QEMU and recording the individual compile commands used for each source file. Bear[4] is a utility that can produce this JSON file. QLEE traverses the JSON file to retrieve the appropriate command for a source file and modifies it to compile the source file to LLVM IR.

5 Evaluation

QLEE was evaluated on ten targets across three device emulators listed in Table 1. The emulators are all Universal Asynchronous Receiver Transmitter[5] (UART) devices, which are simple devices that transmit and receive characters over a serial line.

Target	Entry-points
ARM's PL011	• `pl011_read` • `pl011_write` • `pl011_get_baudrate` • `pl011_can_receive` • `pl011_loopback_mdmctrl` • `pl011_reset`
Intel's 16550A ("`Serial.c`")	• `serial_ioport_read` • `serial_ioport_write`
AVR USART (ATmega, e.g., Arduino)	• `avr_usart_read` • `avr_usart_write`

Table 1: Entry-points for different targets.

The results of the evaluation are listed in Table 2. Adding constraints reduced the false positives reported by KLEE. Since QLEE is based on symbolic execution, it would not be a fair comparison to the related work due to a significantly different methodology. However, QLEE is not meant to replace the other tools but to provide an alternative that finds "shallow" programming bugs faster. All but one of the targets finished running in **less than 5 seconds**, and the `serial_ioport_write` seems to be non-terminating. Manual inspection revealed a `do-while` loop in that function, which might be causing an infinite loop in KLEE's execution. Nevertheless, KLEE still reported errors, albeit false positives, before getting stuck in a loop.

As noted, constraints specified for the target devices are not exhaustive, as not every constraint mentioned in the programming manual is specified. Furthermore, the constraints cannot capture *temporal* properties and ensure the correct order of operations.

[2]https://yaml.org/

[3]https://github.com/llvm/llvm-project/tree/main/llvm/include/llvm/IR

[4]https://github.com/rizsotto/Bear

[5]https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

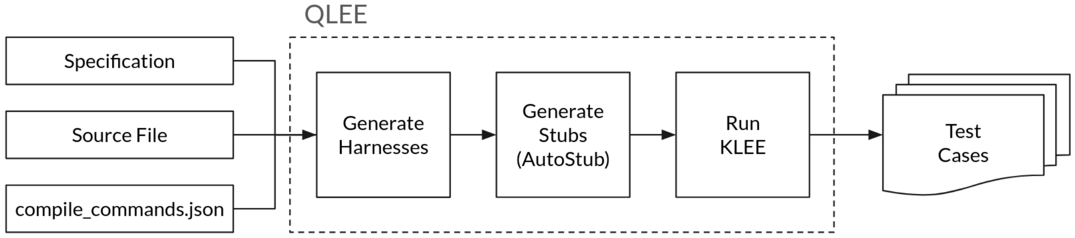

Figure 2: Overview of QLEE

	Without Constraints		With Constraints	
	True Positives	**False Positives**	**True Positives**	**False Positives**
PL011	1	2	1	1
16550A	0	3*	0	1*
AVR USART	0	3	0	0

Table 2: Comparison of true and false positives with and without constraints. * Includes non-terminating case

6 Discussion

QLEE did manage to find a potential bug in PL011's read function[6], which resulted from a subtle misunderstanding of the GCC's case-range extension. Listing 1 contains the code for this case. Given an `offset` value of `0x1000`, the case is triggered with the value of `0x400` since the case-range extension is *inclusive* on both ends[7]. Thus, the resulting index into `s->id` becomes 8 while the array has a length of 8, and therefore, the index must be $0 \leq i \leq 7$.

```
switch (offset >> 2) {
 ...
 case 0x3f8 ... 0x400:
        r = s->id[(offset - 0xfe0) >> 2];
        break;
 ...
}
```

Listing 1: PL011 out-of-bounds Array Access

The other errors were false positives from the solver's exploration of invalid state space. Thus, additional constraints mitigated the majority of false positives. Listing 2 is an example of a false positive error where the solver can cause an assertion violation by assigning `size = 0`.

```
Error: ASSERTION FAIL: size == 1 && addr < 8
File: ../hw/char/serial.c
Line: 474
assembly.ll line: 325
State: 2
Stack: ...
```

Listing 2: False Positive Example from 16550A

Listing 3 shows the constraints added to suppress this error.

```
constraints:
 - name: "offset"
   constraint:
     - "< 8"
 - name: "size"
   constraint:
     - "== 1"
```

Listing 3: Constraints for 16550A

7 Limitations

Though QLEE has shown promising results, the current approach has a few limitations.

Undecidability. KLEE, as with any other symbolic execution engine, suffers from undecidability; thus, there is no termination guarantee. This was noted earlier in Section 5 and could be a significant issue where longer runs could be mistaken as an infinite loop.

Stubs' Inaccuracy. Since the generated stubs merely return the default value of the return type, KLEE could miss some paths if any emulator relies on the return values of those functions.

QEMU Structures. Structure fields and variables that require to be initialized using QEMU's internal facilities (e.g., `qemu_irq` which emulates an interrupt or IRQ line) are not supported.

Complex Devices The evaluation in this report only targeted UART devices, which are simple compared to other devices such as network interfaces or storage devices. Those complex devices extensively use Direct Memory Access (DMA), which could require further initialization to work with KLEE.

8 Conclusion

This project introduced QLEE, a utility built on KLEE, to perform automated testing of device emulators in QEMU. The benefit of using QLEE over prior work is the low overhead of symbolic execution compared to spawning, fuzzing, and restoring QEMU VM processes. QLEE allows for fine-grain testing of emulator functions and supports user-provided constraints to reduce its false positive rate. QLEE shows promising results when running over ten target entry points across ten devices and finds a potential out-of-bounds access bug in PL011. Though QLEE is effective and fast, it is not meant to replace complete hypervisor fuzzers. Spawning a VM has the advantage of a completely dynamic environment that can be controlled and allows for more interaction vectors than QLEE currently supports. Thus, QLEE can complement hypervisor fuzzers and help uncover more bugs.

References

[1] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[6]https://github.com/qemu/qemu/blob/1cf9bc6eba7506ab6d9de635f224259225f63466/hw/char/pl011.c#L316
[7]https://gcc.gnu.org/onlinedocs/gcc/Case-Ranges.html

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[3] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. [n. d.]. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *Research in Attacks, Intrusions, and Defenses* (Cham, 2017), Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer International Publishing, 3–25.

[4] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Proceedings 2020 Network and Distributed System Security Symposium* (San Diego, CA). Internet Society. https://doi.org/10.14722/ndss.2020.23096

[5] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing Using Fast Snapshots and Affine Types. (2021).

A Appendix

```
target:
  source: "hw/char/pl011.c"

variables:
  state: &state
    type: "struct PL011State"
    name: "state"
    pass-by-pointer: true
  offset: &offset
    type: "hwaddr"
    name: "offset"
  size: &size
    type: "unsigned"
    name: "size"
  value: &value
    type: "uint64_t"
    name: "value"

constraints:
  - name: "state.fbrd"
    constraint:
      - "<= 0x3f"
  - name: "state.ibrd"
    constraint:
      - "<= 0xffff"
  - name: "state.read_pos"
    constraint:
      - ">= 0"
      - "<= 15"

assignments:
  - name: "state.id"
    value: "pl011_id_arm"

entrypoint:
  - name: "pl011_read"
    args: [*state, *offset, *size]
  - name: "pl011_write"
    args: [*state, *offset, *value, *size]
  - name: "pl011_get_baudrate"
    args: [*state]
  - name: "pl011_can_receive"
    args: [*state]
  - name: "pl011_loopback_mdmctrl"
    args: [*state]
  - name: "pl011_reset"
    args:
      - << : *state
        cast: "(DeviceState*)"
```

Listing 4: Target configuration for PL011

```
// KLEE HARNESS
void klee_make_symbolic(void *addr,
                        size_t nbytes,
                        const char *name);
void klee_assume(uintptr_t condition);

__attribute__((used))
static int main_pl011_read(void)
{
        struct PL011State state;
        hwaddr offset;
        unsigned size;
        klee_make_symbolic(&state, sizeof(state), "state");
        klee_make_symbolic(&offset, sizeof(offset), "offset");
        klee_make_symbolic(&size, sizeof(size), "size");
        klee_assume(state.fbrd <= 0x3f);
        klee_assume(state.ibrd <= 0xffff);
        klee_assume(state.read_pos >= 0);
        klee_assume(state.read_pos <= 15);
        state.id = pl011_id_arm;
        pl011_read(&state, offset, size);
        return 0;
}

__attribute__((used))
static int main_pl011_write(void)
{
        struct PL011State state;
        hwaddr offset;
        uint64_t value;
        unsigned size;
        klee_make_symbolic(&state, sizeof(state), "state");
        klee_make_symbolic(&offset, sizeof(offset), "offset");
        klee_make_symbolic(&value, sizeof(value), "value");
        klee_make_symbolic(&size, sizeof(size), "size");
        klee_assume(state.fbrd <= 0x3f);
        klee_assume(state.ibrd <= 0xffff);
        klee_assume(state.read_pos >= 0);
        klee_assume(state.read_pos <= 15);
        state.id = pl011_id_arm;
        pl011_write(&state, offset, value, size);
        return 0;
}
```

Listing 5: Sample Harnesses for PL011

```
; ModuleID = 'stubs'
source_filename = "stubs"

%struct.VMStateInfo = type { ptr, ptr, ptr }

@error_fatal = dso_local global ptr null
@_TRACE_PL011_CAN_RECEIVE_DSTATE = dso_local global i16 0

define ptr @qdev_new(ptr %0) {
entry:
  ret ptr null
}

define void @qdev_prop_set_chr(ptr %0, ptr %1, ptr %2) {
entry:
  ret void
}

define i1 @sysbus_realize_and_unref(ptr %0, ptr %1) {
entry:
  ret i1 false
}
```

Listing 6: Sample Stubs for PL011