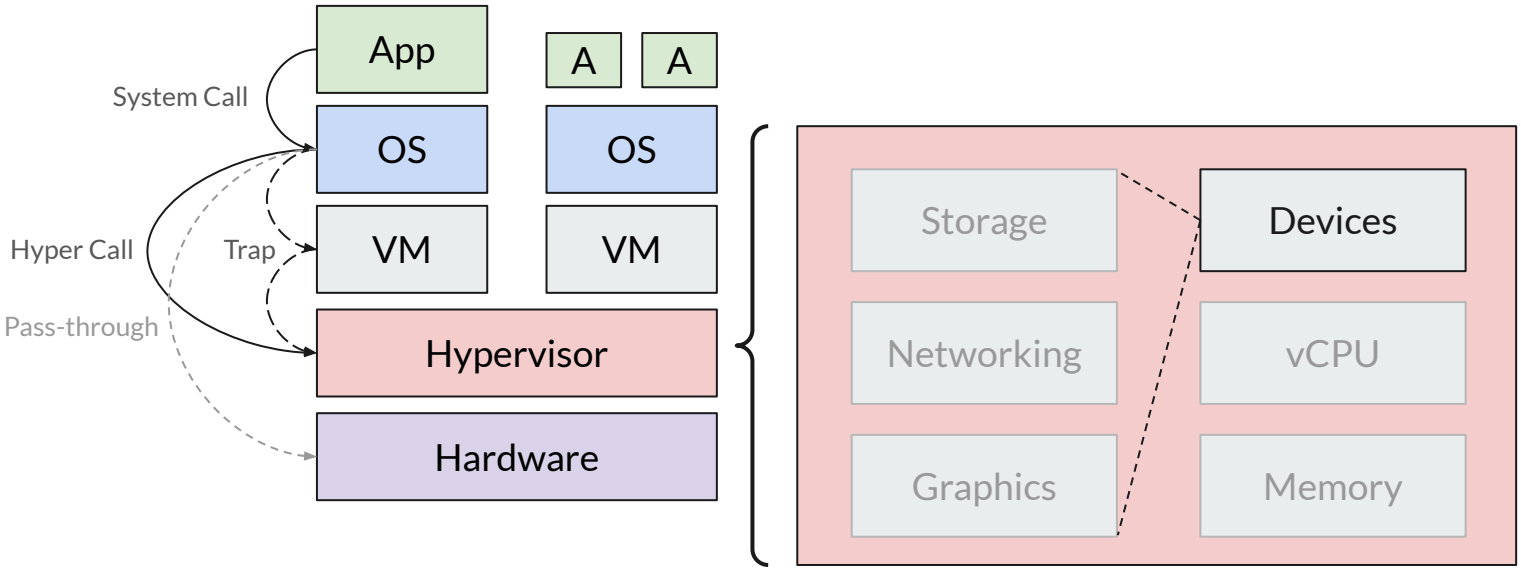
QLEE:
Symbolic Execution of QEMU Devices with KLEE

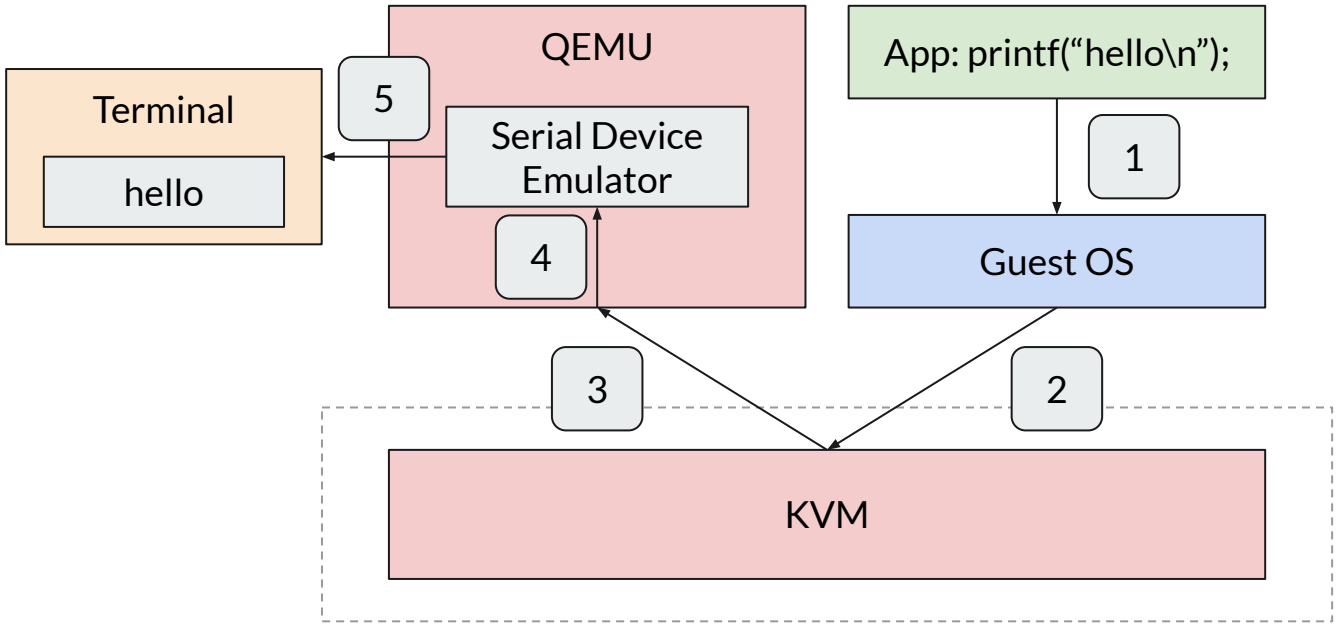
Fairy tale

- [Context] Fuzzing is a popular approach for finding bugs in hypervisors
- [Problem] Existing approaches require a running instance of QEMU to perform the fuzzing, resulting in high overhead and complexity.
- [Solution] QLEE use the KLEE symbolic execution engine to automatically test the hypervisors' emulated devices.
- [Evaluation] QLEE allowed for fine-grained testing of these emulators and found one bug/issue in QEMU (yet to be reported).

Background: Hypervisors



Background: QEMU Device Emulation



Correctness is Critical

Hypervisor Security

- Breaking Isolation Between VMs
- Denial-of-Service
- Data Corruption

Accurate Emulation

- Discrepancies with HW
- Incorrect Device Setup in SW
- Coupling To Implementation Details

Related Work: VDF

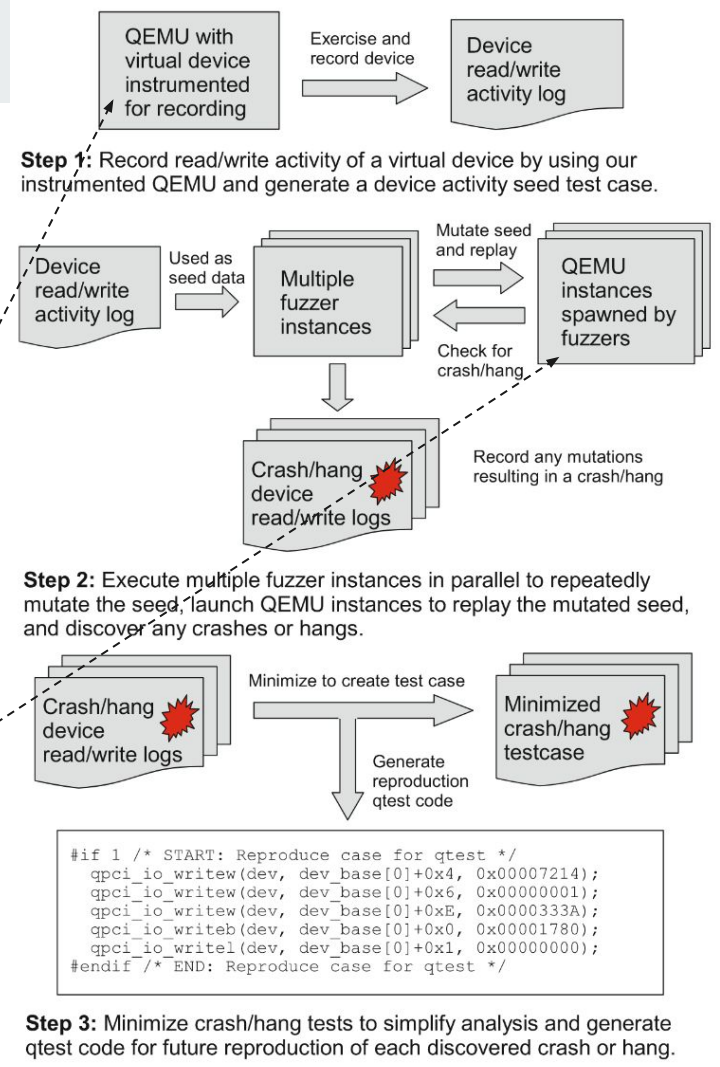
Steps:

1. Record Device Accesses
2. Run AFL:
   a. Mutate the Recordings
   b. Replay on a QEMU Instance
3. Minimize Test Case

Limitations:

- Instrumentation code is added **manually**
- Replay requires **running QEMU** instances

Henderson, A. et al. 2017. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. *Research in Attacks, Intrusions, and Defenses* (Cham, 2017), 3–25.



Related Work: HYPER-CUBE

Steps:

1. Boot the OS & enumerate devices
2. Interpret bytecode to interact w/ devices
3. Minimize test-cases

Limitations:

- Requires **Running** QEMU instance
- OS needs drivers and device enumeration logic
- Complex setup

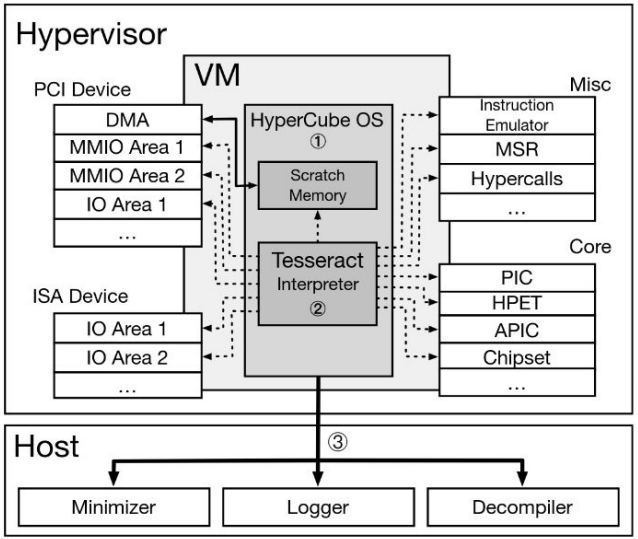
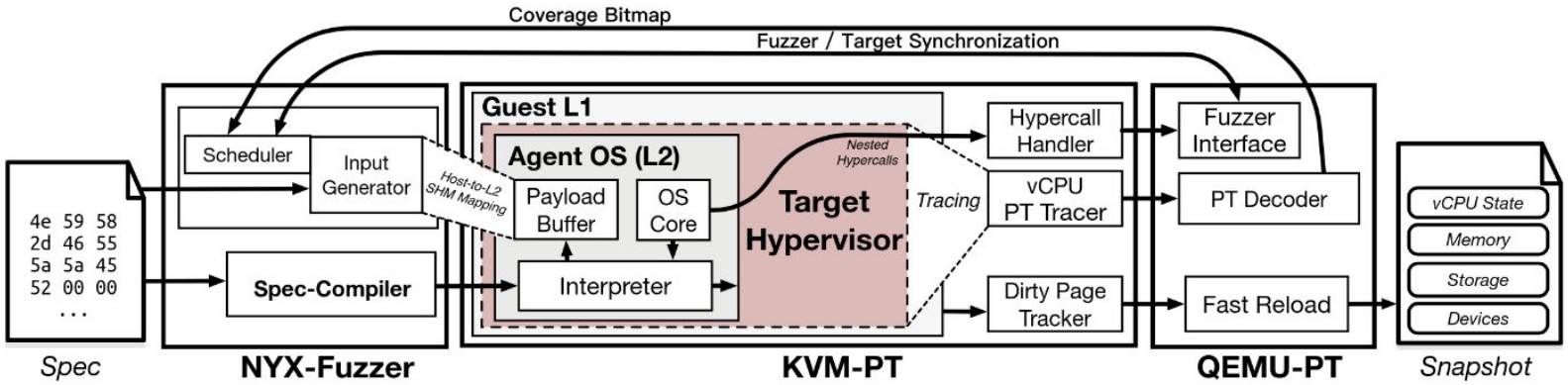


Fig. 2: High-level overview of the system architecture of HYPER-CUBE

Schumilo, S. et al. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. *Proceedings 2020 Network and Distributed System Security Symposium* (San Diego, CA, 2020).

Related Work: NYX (is Complex)



Schumilo, S. et al. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types.

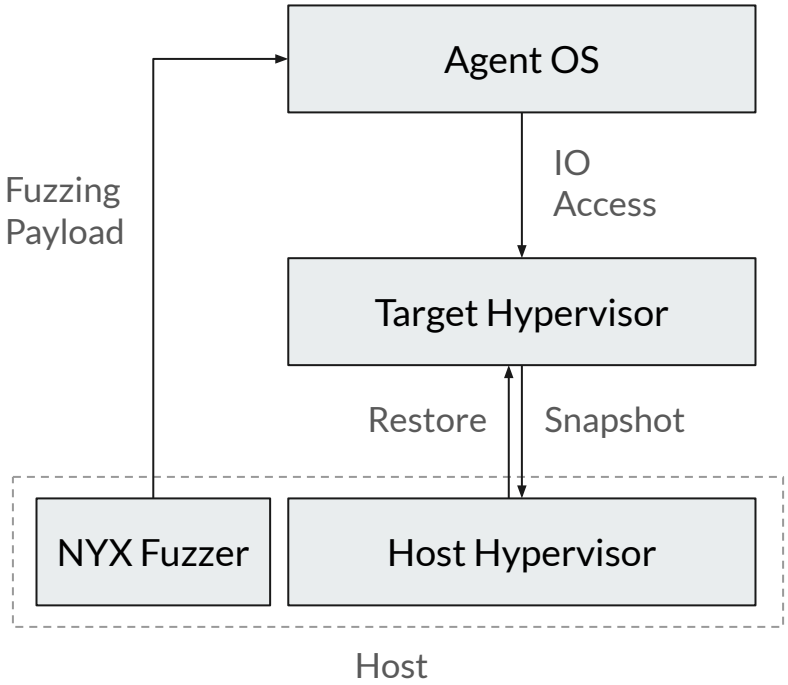
Related Work: NYX

Steps:

1. Initial Snapshot
2. Generate/Mutate Payload
3. Execute & Record coverage
4. Restore snapshot if needed

Limitations:

- Nested Hypervisor Setup
- Requires a **running** hypervisor



Related Work: Common Pain Point

They are all dynamic, i.e., require executing the target

Observations:

- **Running a hypervisor**
  - Overhead
  - Not Needed to Fuzz a Single Device
- Inputs are not constrained to the device operations (VDF & HYPER-CUBE)
  - The recorded initial seed might have bugs (thanks device driver!)
  - The generated ops might not make any sense for the device
  - NYX, the successor of HYPER-CUBE, has a spec.

QLEE:
Symbolic Execution of QEMU Devices with KLEE

KLEE: Symbolic Execution of LLVM IR

Read IR Instruction	Execute (Update State & Constraints)	Check For Errors (at “dangerous ops”)

define i32 @foo(i32 %a, i32 %b, i32 %c) {
entry:
; (1) a + b
%sum = add i32 %a, %b
 	
; (2) c / (a + b)
%result = sdiv i32 %c, %sum
 	
; (3) return result
ret i32 %result
}
	PC
	1. sum = a + b
	2. result = c / sum
	3. return result
	
	Constraints
	1. []
	2. [sum != 0]
	3. [sum != 0]
		(set-logic QF_BV)
		
		(declare-const a (_ BitVec 32))
		(declare-const b (_ BitVec 32))
		
		(assert (= (bvadd a b) #x00000000))
		
		⇒ a = 0, b = 0
		⇒ Division by Zero

Cadar, C. et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.

QLEE Overview

- Specification for Target Device
  - Target device source path
  - Entry-points w/ their arguments
  - Constraints on the arguments
- Harness Generation
- Stub Generation
  - Global Variables
  - External Functions
- Run KLEE on each Entry-point

QLEE: Specification

Includes:

- Source path
- Variable declarations
- Constraints & Assignments
- Entry-points

Why YAML?

- Anchor (&) and Alias (*) to deduplicate
- Order-independent

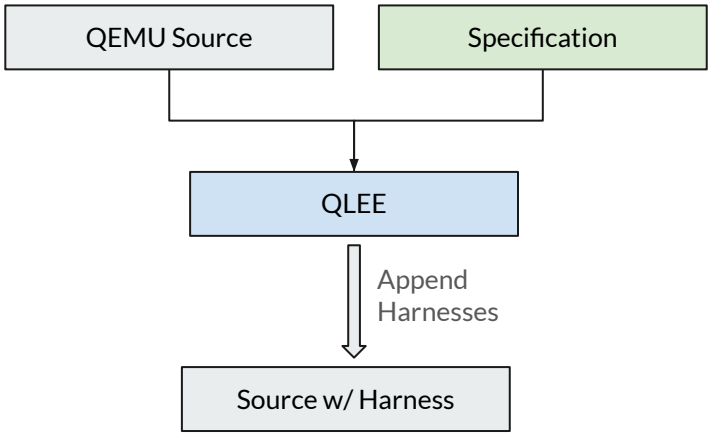
Why Explicit Arguments?

- Function signature has state as void*

```yaml
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"
```

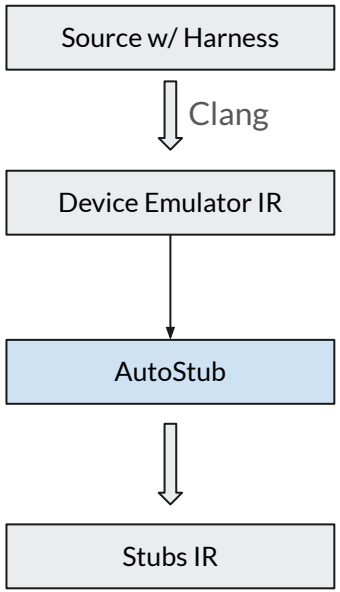
```yaml
assignments:
  - name: "state.id"
    value: "pl011_id_arm"

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*)"
```

QLEE: Harness Generation




```
__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;
}
```

QLEE: Stubs Generation



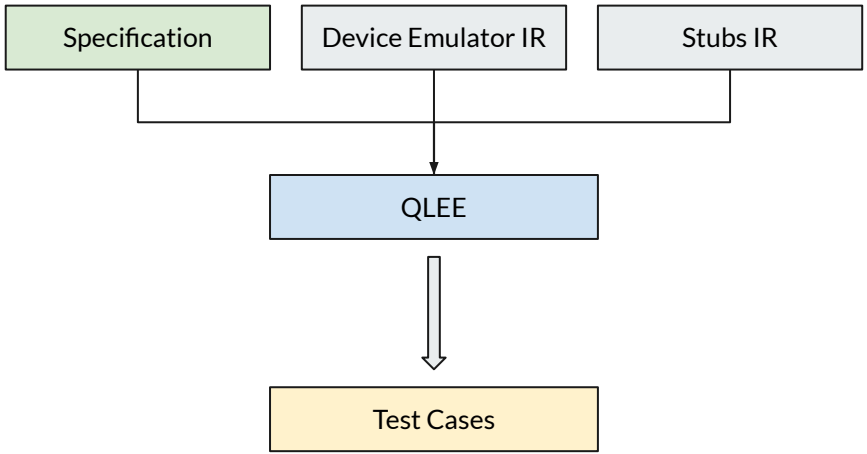
AutoStub:

- C++ and LLVM IR Library
- Stubs
  - Global variables ⇒ initialized to 0
  - External Functions ⇒ return 0
- Types are preserved

Why use IR as input?

- Do not have to deal with #include<header>

QLEE: Run KLEE



- Run KLEE per Entry-point
- Output saved to **<outputdir>/<timestamp>**/<entrypoint>

Evaluation

QEMU: commit 72b88908d1

UART Devices

- Simple & Similar
- Serial Communication
  (e.g., Write character to console)

Target	Entry-points
ARM's PL011	• pl011_read • pl011_write • pl011_get_baudrate • Pl011_can_receive • pl011_loopback_mdmctrl • pl011_reset
"Serial.c" (Intel's 16550A)	• serial_ioport_read • serial_ioport_write
AVR USART (ATmega, e.g., Arduino)	• avr_usart_read • avr_usart_write

Evaluation

	Without Constraints		With† Constraints	
	True Positives	False Positives	True Positives	False Positives
PL011	1	2	1	1
“Serial.c” 16550A	0	3*	0	1*
AVR USART	0	3	0	0

† Not all device constraints were added
* Manually terminated potentially infinite loop (undecidable)

Discussion

PL011 Bug (?)

- Not reported yet
- Array out-of-bounds access

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

- Case range is a gcc extension
- Case x..y is **inclusive** of both x and y

offset == 0x1000
⇒ (0x1000 - 0xfe0 >> 2) = 8
But s->id is 8 bytes, i.e. $0 \leq i \leq 7$

False Positives:

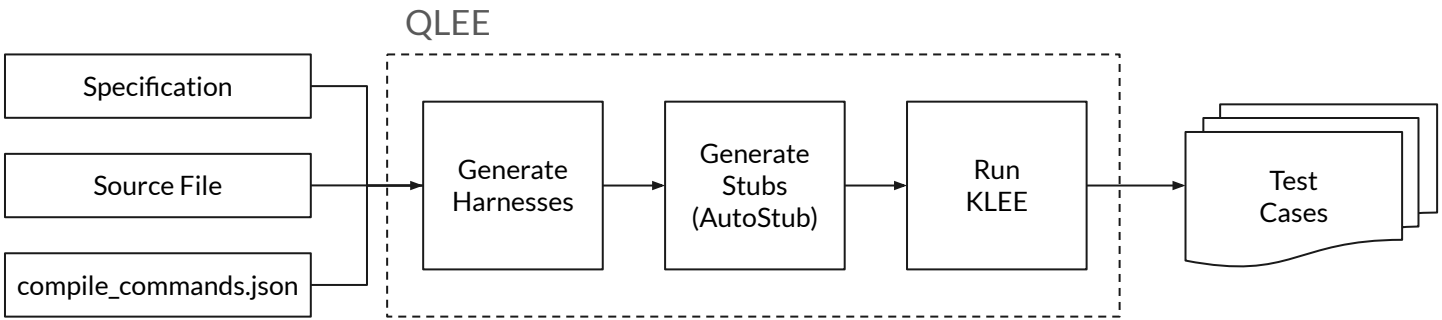
- Trivial Cases
  - Access size == 1
  - Value < 0xff (one byte access)
  - Struct field is null due to missing setup
- Suppressed with constraints

Limitation:

- Proper struct initialization
- Stub functions are not “accurate”

Bibliography

- Schumilo, S. et al. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types.
- Henderson, A. et al. 2017. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. Research in Attacks, Intrusions, and Defenses (Cham, 2017), 3–25.
- Schumilo, S. et al. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. Proceedings 2020 Network and Distributed System Security Symposium (San Diego, CA, 2020).
- LLVM Project. 2003–present. The LLVM Compiler Infrastructure. Available at: https://llvm.org.


QLEE
Specification
Source File
compile_commands.json
Generate Harnesses
Generate Stubs (AutoStub)
Run KLEE
Test Cases