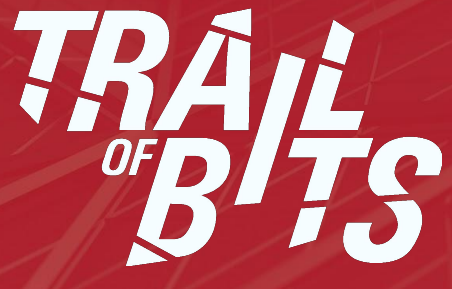


Binary symbolic execution with KLEE-Native

Sai Vegasena

The Team

Sai Vegasena

Security Engineering Intern

sai.vegasena@trailofbits.com

@svegas18

Peter Goodman

Senior Security Engineer

peter@trailofbits.com

@peter_a_goodman

During my internship I ...

- Developed a fork of KLEE that operates on raw binaries
- Translated machine code to LLVM bitcode with Remill and ran it in KLEE
- Wrote a custom allocator to accurately model heap memory in KLEE's emulator
- Implemented virtualized system calls that productively handled symbolic data
- Developed a new forking model for KLEE's symbolic executor
- Reproduced a CVE used in an old ChromeOS exploit chain

KLEE is a symbolic virtual machine that executes LLVM bitcode

- Applied in software testing and verification
- Dynamically generates high-coverage producing inputs
- Leverages a custom runtime environment

Bugs Galore :)

KLEE's greatest strength is also its greatest weakness

Pros of running LLVM bitcode

- Allows for custom runtime definitions and intrinsics
- Executes anything clang can compile
  - i.e C, C++, Rust, Swift, Go,, etc

Cons of running LLVM bitcode

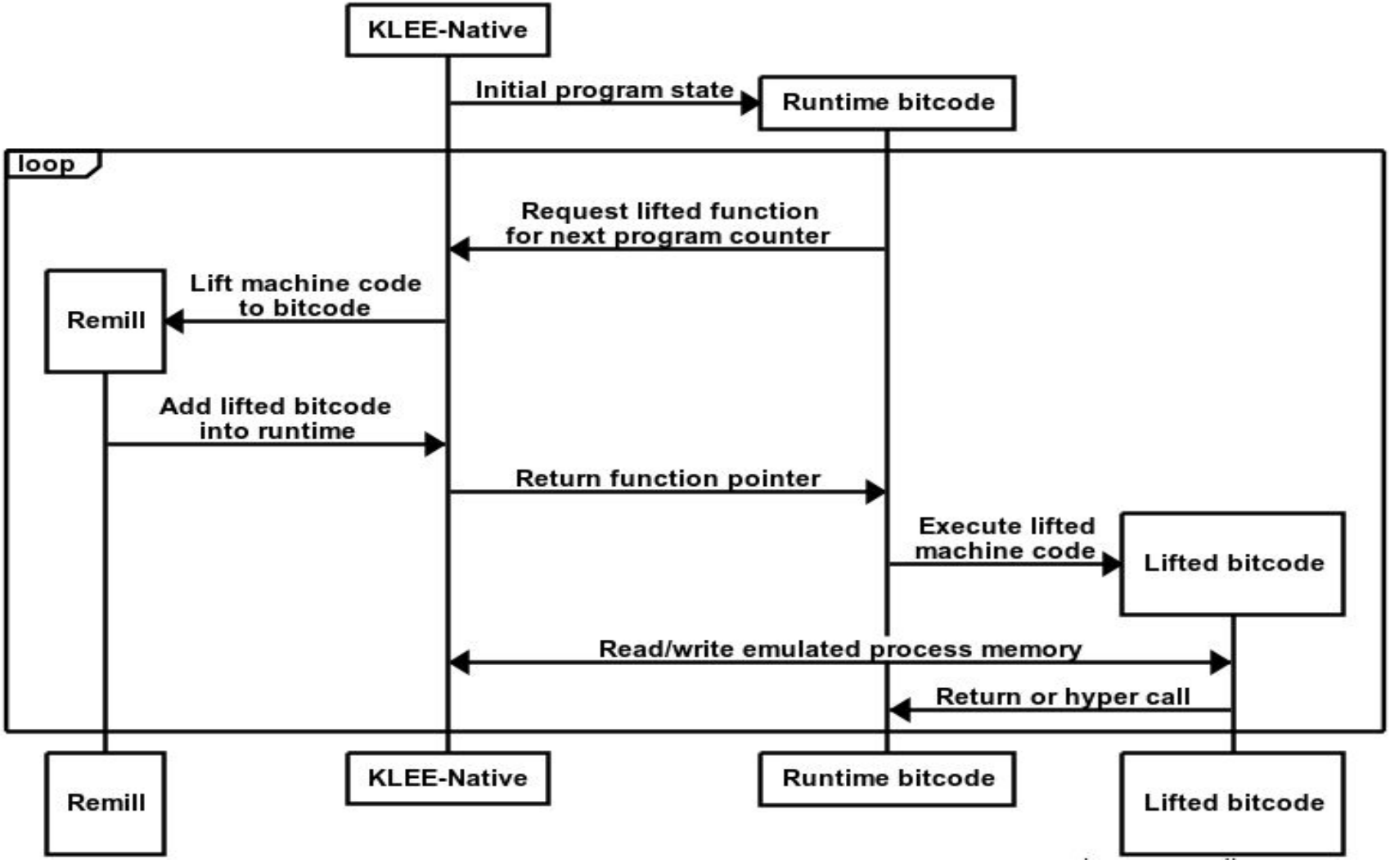
- Sometimes need source
- Build systems, configs, and dependencies
- Manually Injecting KLEE-API calls into the source
- McSema is an option but CFG recovery is limiting and there are occasional inaccuracies

KLEE-Native operates on snapshotted program binaries

- Binaries are snapshotted with user defined breakpoints
  - Static breakpoints
  - Dynamic breakpoints for ASLR
  - `./klee-snapshot-7.0 --workspace_dir ws --dynamic --breakpoint 0x1337 --arch amd64_avx -- ./a.out`
- 

lifts machine code instructions to LLVM bitcode
- KLEE-Native executes the runtime and the lifted LLVM

```
./klee-exec-7.0 --workspace_dir ws
```


KLEE-Native
Initial program state
Runtime bitcode
loop
Request lifted function
for next program counter
Remill
Lift machine code
to bitcode
Add lifted bitcode
into runtime
Return function pointer
Execute lifted
machine code
Lifted bitcode
Read/write emulated process memory
Return or hyper call
Remill
KLEE-Native
Runtime bitcode
Lifted bitcode

Runtime is the kernel and the machine

- Remill async hyper call is defined in runtime
  - “Implements” OS functionality
  - Execution is passed to a linux system call wrapper in runtime

```
template <typename ABI>
static Memory *SysOpen(Memory *memory, State *state,
                       const ABI &syscall);
```

- Custom ABI extracts arch info from state
  - Store return value
  - Extract args
  - Find syscall number

```
if (path_len >= PATH_MAX) {
  STRACE_ERROR(open, "Path name too long: %s", gPath);
  return syscall.SetReturn(memory, state, -ENAMETOOLONG);
// The string read does not end in a NUL-terminator; i.e. we read less
// than `PATH_MAX`, but as much as we could without faulting, and we didn't
// read the NUL char.
} else if ('\0' != gPath[path_len]) {
  STRACE_ERROR(open, "Non-NUL-terminated path");
  return syscall.SetReturn(memory, state, -EFAULT);
}
```

- Wrappers do error checking done by OS
  - Gives a kernel-level “insight”

Simplifying lifted libc functions with accuracy

- Problem
  - Lifting libc functions is slow
  - Unnecessary state forking on symbolic data

- LD_PRELOAD-based library into snapshotted programs
  - Lets us interpose and hook to simple libc variant in the runtime

- Variants handle symbolic data in a simple way with no lifting

Heap memory is accurately modeled with libc intercepts

- Lifted mallocs call brk or mmap
  - Technically accurate but bad for bug-finding
  - No clarity for bounds checks on allocations
  - Hard to oversee UAFS, double frees, and access violations
- Utilize the intercept hook to organize allocations in a uniform structure
- Basically implemented a custom allocator
- Custom address encoding helps “locate” allocations in alloc lists on mallocs and frees

Encoding

```
union Address {
  uint64_t flat;
  struct {
      uint64_t offset :16;
      uint64_t must_be_0x1 :4;
      uint64_t size :16;
      uint64_t alloc_index :24;
      uint64_t must_be_0xa :4;
  } __attribute__((packed));
};
```

It is possible to fall back to real libc functions in KLEE-Native

Before Snapshot

```
malloc:                                 ; CODE XREF: .text:
                                        ; .text:__GI___libc
        jmp     cs:ptr_to_malloc_handler
```

Emulated in KLEE-Native

```
handle_malloc_hypercall:
        int     81h
        retn
```

Fallback to real libc malloc just in case

```
handle_malloc_passthrough:
        push    r10
        mov     r10, cs:ptr_to_addr_of_real_malloc
        mov     r10, [r10]
        xchg    r10, [rsp]
        retn
```

It is possible to fall back to real libc functions in KLEE-Native

Before Snapshot

```
malloc:                                                ; CODE XREF: .text:
                                                       ; .text:  GT __libc
```

NOP when we load state

Emulated in KLEE-Native

```
handle_malloc_hypercall:
                int     81h
                retn
```

Fallback to real libc malloc just in case

```
handle_malloc_passthrough:
                push    r10
                mov     r10, cs:ptr_to_addr_of_real_malloc
                mov     r10, [r10]
                xchg    r10, [rsp]
                retn
```

It is possible to fall back to real libc functions in KLEE-Native

Before Snapshot

```
malloc:                                          ; CODE XREF: .text:
                                                 ; .text:  GT __libc
```

NOP when we load state

Emulated in KLEE-Native

```
handle_malloc_hypercall:
        int     81h
        retn
```

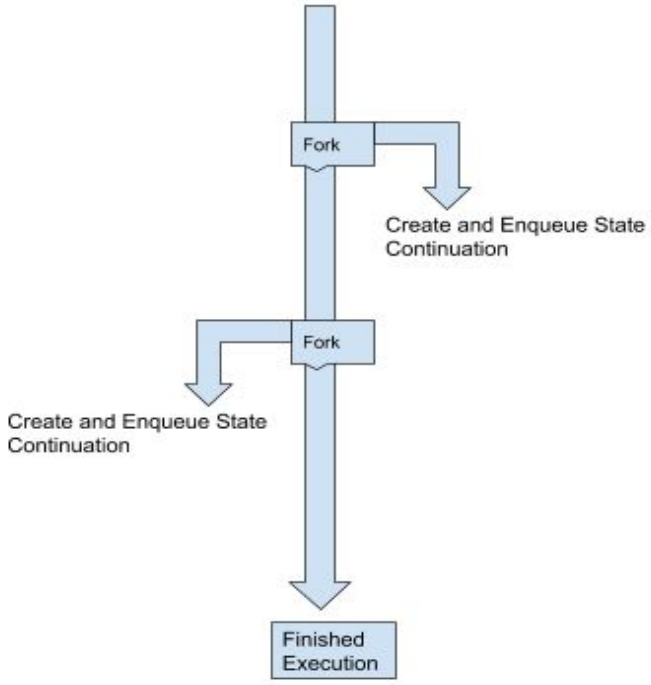
Skip the ret in emulator

Fallback to real libc malloc just in case

```
handle_malloc_passthrough:
        push    r10
        mov     r10, cs:ptr_to_addr_of_real_malloc
        mov     r10, [r10]
        xchg    r10, [rsp]
        retn
```

Eager concretization is better than eager forking

- Closer in spirit to SAGE, a static symbolic symbolic executor
- Akin to lazy evaluation
- “State continuations” are moral equivalent of a Python generator, and may be invoked to give “the next viable fork at this point”
- Continuations are enqueued and scheduled later
- Handle branches and symbolic addresses



KLEE-Native can identify real bugs

- Make a snapshot in vulnerable function

```
klee-snapshot-7.0 --workspace_dir ws_CVE --dynamic --breakpoint
0xb33 --arch amd64_avx -- ./c_ares_repro
```

- Run klee-exec

```
klee-exec-7.0 --workspace_dir ws_CVE
```

- Policy handler provides an interesting feature that Valgrind and Asan don't

```
KLEE: WARNING ONCE: Alignment of memory from call "_Znwm" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling external: vfprintf(140017979533152, 94410329517216, 94410333418800) at [no debug info]
libc_memcpy:dest=7ffcee613f70, src=400d94, len=8, ret=7ffcee613f70
libc_malloc:size=19, ptr=a000000001310000
E0729 16:48:17.126665  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9dfd9d 3 (BYTES 0f 50 dc) MO
VMSKPS_GPR32_XMMps (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM4)))
E0729 16:48:17.128990  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9e01c2 4 (BYTES 66 0f 50 d8)
 MOVMSKPD_GPR32_XMMpd (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM0)))
E0729 16:48:17.130098  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9e0434 3 (BYTES 0f 50 dd) MO
VMSKPS_GPR32_XMMps (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM5)))
E0729 16:48:17.140801  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9e1a1a 4 (BYTES 66 0f 50 d8)
 MOVMSKPD_GPR32_XMMpd (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM0)))
E0729 16:48:17.174049  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9dfd9d 3 (BYTES 0f 50 dc) MO
VMSKPS_GPR32_XMMps (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM4)))
E0729 16:48:17.176441  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9e01c2 4 (BYTES 66 0f 50 d8)
 MOVMSKPD_GPR32_XMMpd (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM0)))
E0729 16:48:17.177579  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9e0434 3 (BYTES 0f 50 dd) MO
VMSKPS_GPR32_XMMps (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM5)))
E0729 16:48:17.188436  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9e1a1a 4 (BYTES 66 0f 50 d8)
 MOVMSKPD_GPR32_XMMpd (WRITE_OP (REG_64 RBX)) (READ_OP (REG_128 XMM0)))
E0729 16:48:17.305361  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcb9f6250 8 (BYTES 66 0f 57 05
b8 6b 14 00) XORPD_XMMxuq_MEMxuq (WRITE_OP (REG_512 ZMM0)) (READ_OP (REG_512 ZMM0)) (READ_OP (DWORD_PTR (ADD (REG_64 PC
) (SIGNED_IMM_64 0x146bc0)))))
E0729 16:48:17.633155  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcbb1abca 4 (BYTES d9 74 24 d8)
 FNSTENV_MEMmem28 (WRITE_OP (DWORD_PTR (ADD (REG_64 RSP) (SIGNED_IMM_64 -0x28)))))
E0729 16:48:17.633316  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcbb1abfb 4 (BYTES d9 74 24 d8)
 FNSTENV_MEMmem28 (WRITE_OP (DWORD_PTR (ADD (REG_64 RSP) (SIGNED_IMM_64 -0x28)))))
E0729 16:48:17.633388  6124 Lifter.cpp:123] Missing semantics for instruction (AMD64 7fbfcbb1ac10 4 (BYTES d9 74 24 d8)
 FNSTENV_MEMmem28 (WRITE_OP (DWORD_PTR (ADD (REG_64 RSP) (SIGNED_IMM_64 -0x28)))))
KLEE: WARNING ONCE: Alignment of memory from call "_Znam" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling external: write(2, 94410361759936, 12) at [no debug info]
HIT STRCMP!
  1:write:fd=2, size=12/12
E0729 16:49:12.509608  6124 AllocList.cpp:157] Heap address overflow on memory write address a000000001310013
KLEE: ERROR: (location information missing) Failed 1-byte write of 1 to address 0xa000000001310013 in address space 1
KLEE: NOTE: now ignoring this error at this location
I0729 16:49:12.509789  6124 Executor.cpp:3078] Finished state, continuation stack size is 1
```

Sai Vegasena: sai.vegasena@trailofbits.com
Peter Goodman: peter@trailofbits.com

Symex on Binary Snapshots with KLEE-Native