Fuzzing OP-TEE with AFL

Martijn Bogaard
Senior Security Analyst

martijn@riscure.com
@jmartijnb

About me

- Senior Security Analyst @ Riscure (2015 – now)
  - Code reviews & pentests of security critical sw components of embedded systems, hardware evals (chip design reviews, fault injection testing), sw security trainer
  - Evaluating security of (proprietary) TEEs since late 2015

- Actively researching fuzzing & simulation in last 6 years
  - Distributed fuzzing (Disfuzz AFL, 2014)
  - Fuzzing Windows drivers under Linux using kernel emulation (WKE, 2015)
  - OP-TEE simulator for Linux (2017)
  - Fault Injection attack simulation (FiSim, 2018)
  - Custom simulators for fuzzing & dynamic analysis of bootloaders, RTOS, Android services, …

Today's agenda

- Trusted Execution Environments?!?
- Why is fuzzing TEEs so difficult? (Is it…?)
- Fuzzing OP-TEE syscalls

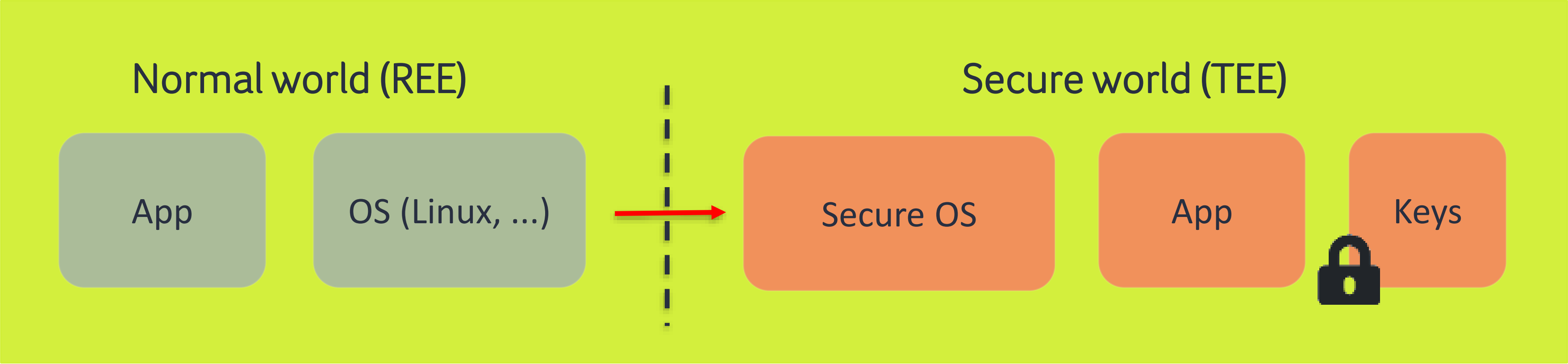
Research motivation

- Trusted Execution Environments become widespread
  - Mandatory in Android 6+ (conditionally)
  - Starting to pick up in other markets e.g. automotive

- Highly privileged component in modern chips
  - Way more than Android/Linux
  - Strictly isolated from the rest

- Often controls access to device's most secret crypto keys
  - KeyMaster, GateKeeper, DRM, Mobile Banking, …

Concept



Normal world (REE)

App

OS (Linux, ...)

Secure world (TEE)

Secure OS

App

Keys

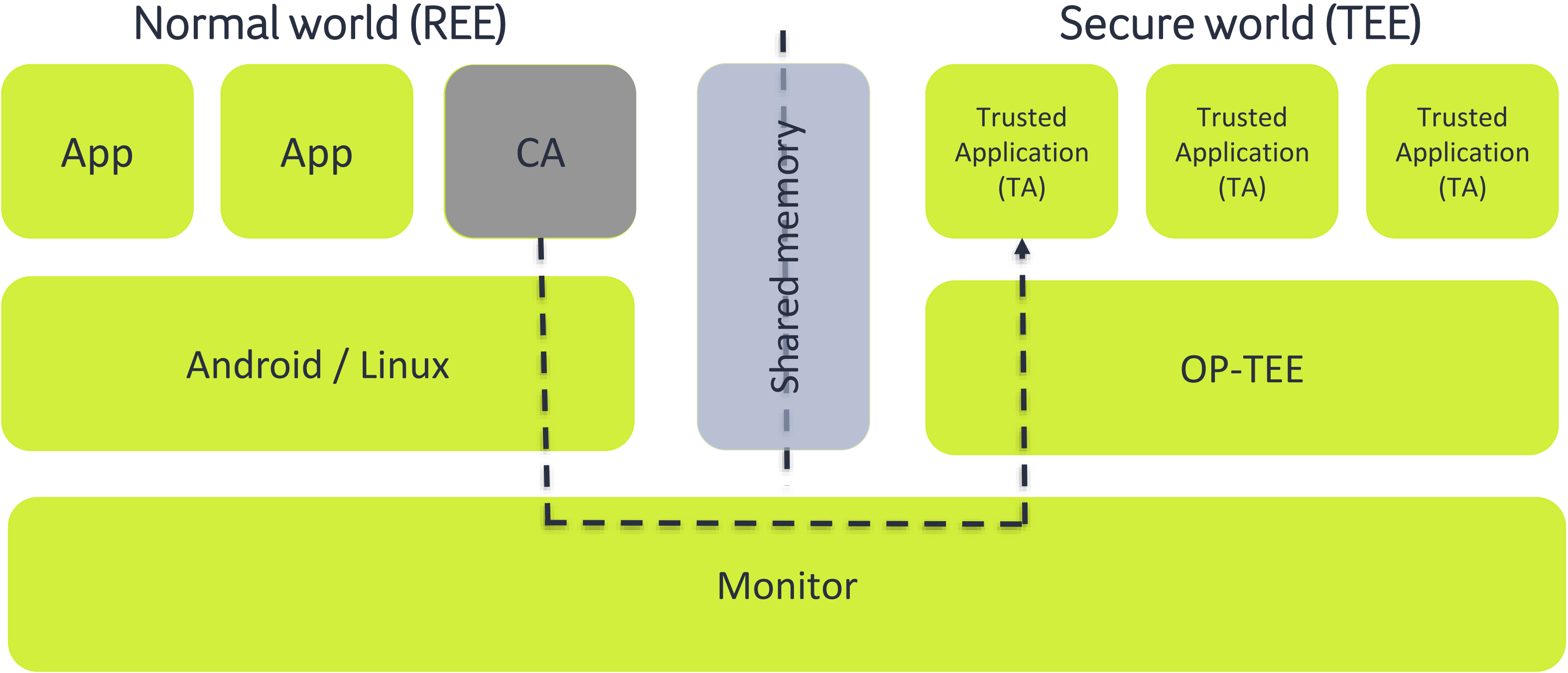
TEE Technology

- **Arm TrustZone**
- Intel SGX
- MIPS Virtualization
- RISC-V Keystone

TEE Technology

- **Arm TrustZone → Linaro's OP-TEE**
- Intel SGX
- MIPS Virtualization
- RISC-V Keystone

Trusted Execution Environment



Normal world (REE)

Secure world (TEE)

App

App

CA

Shared memory

Trusted Application (TA)

Trusted Application (TA)

Trusted Application (TA)

Android / Linux

OP-TEE

Monitor

Trusted Applications

- Often written by chip vendor or OEM in C
- Global Platform API for compatibility between TEEs
  - Specifies exported symbols + TA API

→ Not like any normal world application!

Example: AES in TA

```
TEE_Result TA_InvokeCommandEntryPoint(void* sess_ctx, uint32_t cmd_id,
                                      uint32_t param_types, TEE_Param params[4])
{
    if (cmd_id == CMD_AES_ENCRYPT)  {
        [...]

        TEE_AllocateOperation(&op_handle, TEE_ALG_AES_CBC_NOPAD, TEE_MODE_ENCRYPT, AES128_KEY_SIZE);
        TEE_AllocateTransientObject(TEE_TYPE_AES, AES128_KEY_SIZE, &key_handle);
        TEE_InitRefAttribute(&attr, TEE_ATTR_SECRET_VALUE, key, AES128_KEY_BYTE_SIZE);
        TEE_PopulateTransientObject(key_handle, &attr, 1);
        TEE_SetOperationKey(op_handle, key_handle);
        TEE_CipherInit(op_handle, iv, iv_sz);
        TEE_CipherUpdate(op_handle,  buf_in, buf_in_len, buf_out, &buf_out_len);
    }
    else {
        return TEE_ERROR_BAD_PARAMETERS;
    }
}
```

Let’s go one layer deeper….

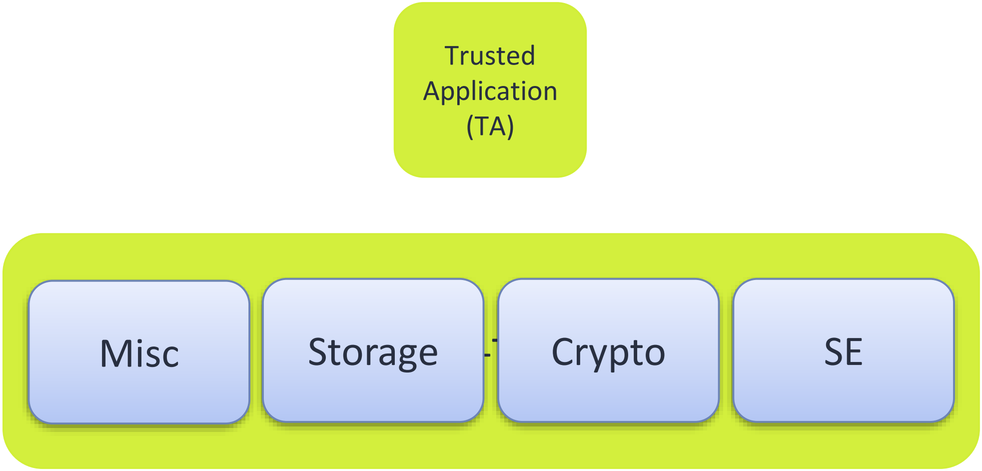
Example: AES in OP-TEE

```
uint32_t obj_handle1;
uint32_t obj_handle2;

uint32_t state_handle;

syscall_cryp_obj_alloc(0xa0000010, 0x80, &obj_handle1);
syscall_cryp_state_alloc(0x10000110, 0x0, obj_handle1, 0x0, &state_handle);
syscall_cryp_obj_alloc(0xa0000010, 0x80, &obj_handle2);
syscall_cryp_obj_populate(obj_handle2, {c0000000, buf_key, 0x10}, 0x1);
syscall_cryp_obj_reset(obj_handle1);
syscall_cryp_obj_copy(obj_handle1, obj_handle2));
syscall_cipher_init(state_handle, iv, 0x10);
syscall_cipher_update(state_handle, "Hello Nullcon!!!", 0x10, buf_out ,&buf_out_len);
```

OP-TEE syscalls



Trusted Application (TA)

Misc

Storage

Crypto

SE

Total: 70 syscalls

However, does *trusted* also mean secure?

Fuzzing

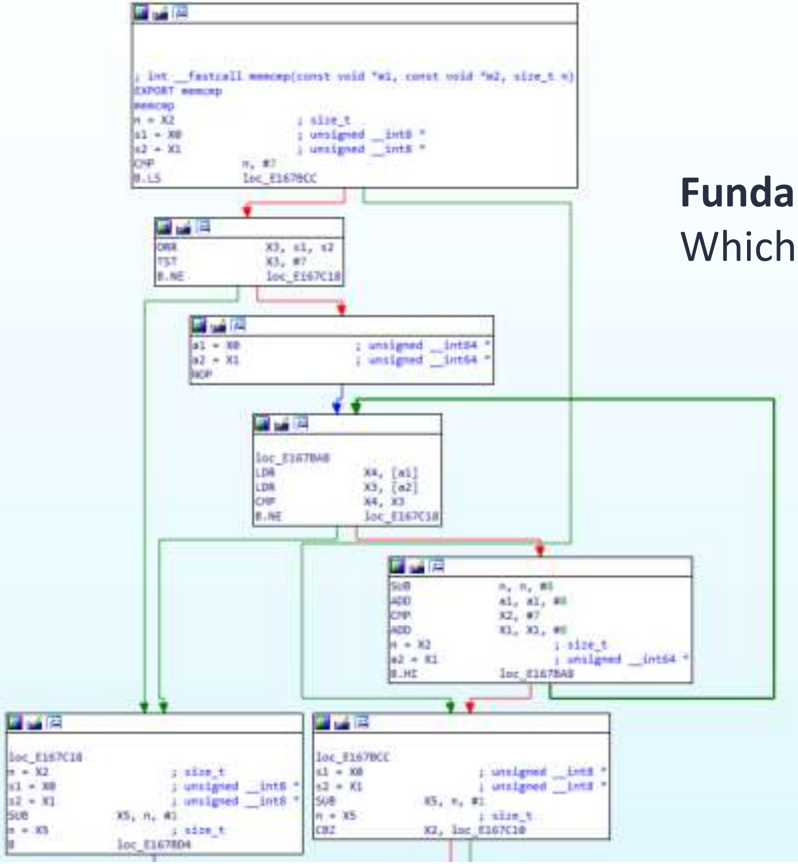
- Random data
  - Prototype by colleagues in 2014
  - cat /dev/urandom > /dev/tee_smc
- Model-based
- Coverage guided evolutionary fuzzing

Coverage guided evolutionary fuzzing

Simple algorithm:

1. Generate new input from collection of corpora
    - By applying 1 or more mutations (e.g. bit flips)
2. Run target with input
3. Collect code coverage information
4. If coverage information shows a previously unseen code path is taken, add to corpus queue

Coverage tracking



```
; int __fastcall memcmp(const void *s1, const void *s2, size_t n)
EXPORT memcmp
memcmp
n = X2                          ; size_t
s1 = X0                         ; unsigned __int8 *
s2 = X1                         ; unsigned __int8 *
CMP             n, #7
B.LS            loc_E167BCC
```

```
ORR             X3, s1, s2
TST             X3, #7
B.NE            loc_E167C18
```

```
a1 = X0                         ; unsigned __int64 *
a2 = X1                         ; unsigned __int64 *
NOP
```

```
loc_E167BA8
LDR             X4, [a1]
LDR             X3, [a2]
CMP             X4, X3
B.NE            loc_E167C18
```

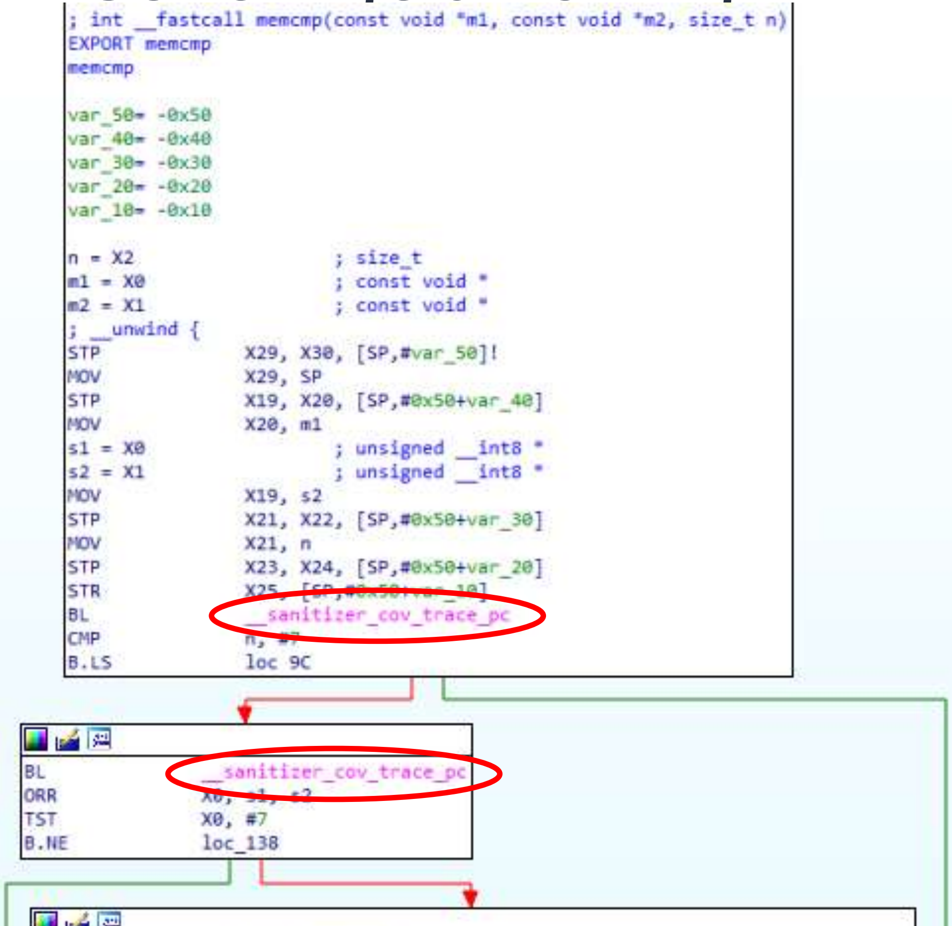
```
SUB             n, n, #8
ADD             a1, a1, #8
CMP             X2, #7
ADD             X1, X1, #8
n = X2                          ; size_t
a2 = X1                         ; unsigned __int64 *
B.HI            loc_E167BA8
```

```
loc_E167C18
n = X2                          ; size_t
s1 = X0                         ; unsigned __int8 *
s2 = X1                         ; unsigned __int8 *
SUB             X5, n, #1
n = X5                          ; size_t
B               loc_E167BD4
```

```
loc_E167BCC
s1 = X0                         ; unsigned __int8 *
s2 = X1                         ; unsigned __int8 *
SUB             X5, n, #1
n = X5                          ; size_t
CBZ             X2, loc_E167C18
```

Fundamental question:
Which branches are taken?

Coverage tracking



```
; int __fastcall memcmp(const void *m1, const void *m2, size_t n)
EXPORT memcmp
memcmp

var_50= -0x50
var_40= -0x40
var_30= -0x30
var_20= -0x20
var_10= -0x10

n = X2                  ; size_t
m1 = X0                 ; const void *
m2 = X1                 ; const void *
; __unwind {
STP             X29, X30, [SP,#var_50]!
MOV             X29, SP
STP             X19, X20, [SP,#0x50+var_40]
MOV             X20, m1
s1 = X0                 ; unsigned __int8 *
s2 = X1                 ; unsigned __int8 *
MOV             X19, s2
STP             X21, X22, [SP,#0x50+var_30]
MOV             X21, n
STP             X23, X24, [SP,#0x50+var_20]
STR             X25, [SP,#0x50+var_10]
BL              __sanitizer_cov_trace_pc
CMP             n, #7
B.LS            loc_9C
```

```
BL              __sanitizer_cov_trace_pc
ORR             X0, s1, s2
TST             X0, #7
B.NE            loc_138
```

Why is fuzzing operating systems difficult?

- Crashes
- Global state
- Coverage tracking
- Seeding
- Trace stability → threading, SMP, interrupts

↓

A lot of progress for Linux and other mainstream Oss
e.g. AFL, Syzkaller, ...

Let’s make use of that!

Goals

- Reuse an existing fuzzer (AFL)
  → Focus on the TEE challenge, not building a fuzzer
- Good, not perfect results (limited time)

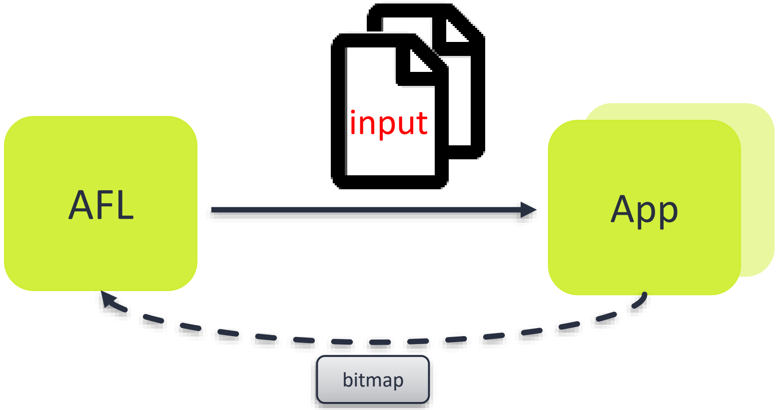
Why is fuzzing *TEEs* difficult?

All before

+

- Isolated environment
- Separate operating system
- Limited API
- **Seeding**

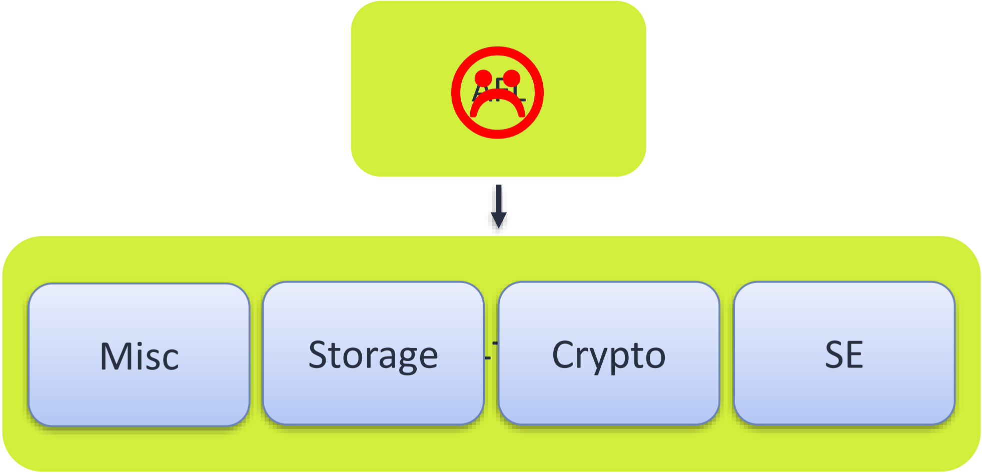
AFL



```
7ae0  00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00  ................
7b00  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7b70  00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00  ................
7b90  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
7fa0  00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
81e0  00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00  ................
8240  00 23 00 00 00 00 00 00 00 00 00 00 00 00 23 00  .#............#.
8250  00 00 00 00 23 00 00 00 23 00 00 00 00 00 00 00  ....#...#.......
8270  00 00 23 00 00 00 39 00 00 00 00 00 00 00 00 00  ..#...9.........
85f0  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Corresponds with $addr somewhere in binary

AFL as Trusted Application?



AFL

Misc

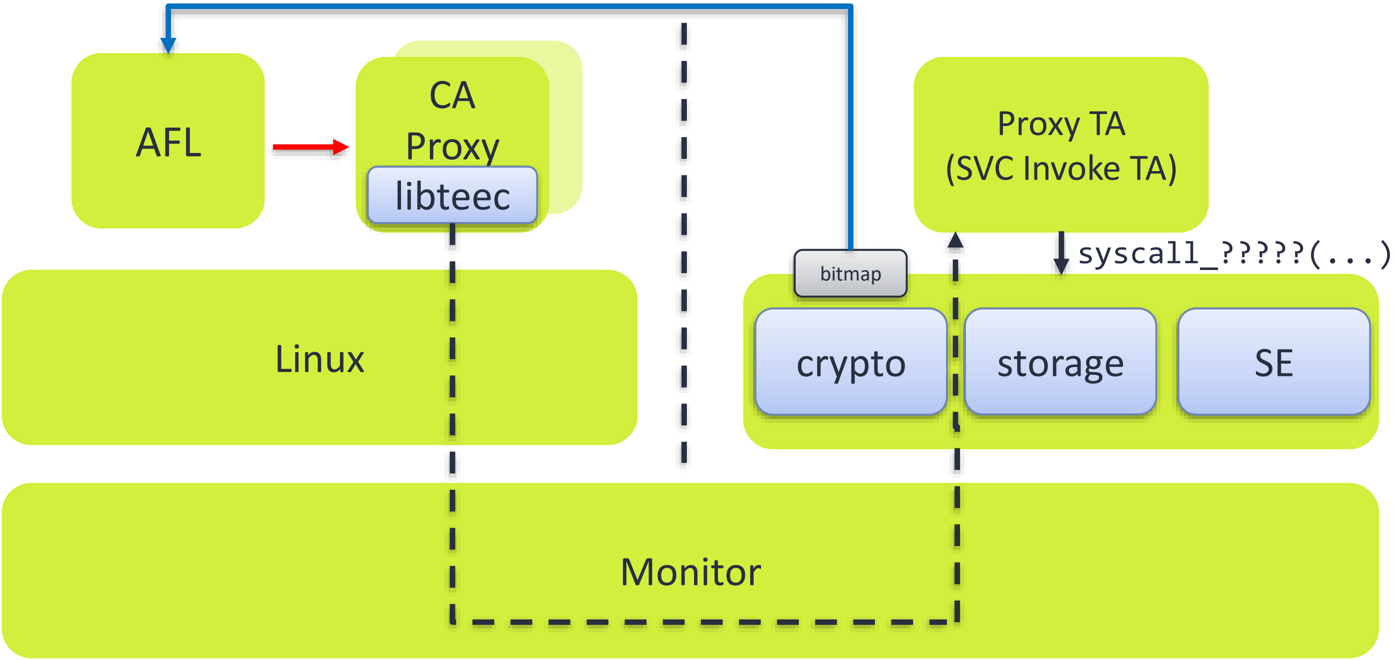
Storage

Crypto

SE

```
fork(…)?    → TEE_ERROR_NOT_IMPLEMENTED
execve(…)?  → TEE_ERROR_NOT_IMPLEMENTED
open(…)?    → TEE_ERROR_NOT_IMPLEMENTED
```

AFL as ~~Trusted~~ Linux Application



How to (randomly) invoke system calls using AFL?

AFL can only mutate a blob of (random) data by flipping bits or bytes...

Hello!

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

```
t[0] = malloc(16);
memcpy(t[0], "\x0aHello Connect!\x00", 16);
r[0] = utee_log(t[0], 0x10);
free(t[0]);
```

Syscall wrapper function

System calls as data

- Syscalls consists of id + up to 8 arguments
  - Values
  - Pointers to data, structures, etc
  - Pointers to structures with pointers, etc.
- Syscall arguments often depend on prev. syscall
  - E.g. returned handles

→ Argument encoding is the hardest part!

System calls as data

- Simple binary format encoding 1 or more syscall invokes
  - Contains arguments inline except buffer content
  - Goal: every bit flip results in a slightly different invoke

- After invoke data follows section with raw data
  - Strings, buffer content, etc.
  - Can be flexible referenced from argument info

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Syscall id

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Argument types

→ 1 nibble per argument

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Argument types

0xa: argument 0 is a buffer with in-line data

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Argument types

0xa: argument 0 is a buffer with in-line data

0x4: argument 1 is a 32-bit integer value

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Argument 0: buffer

Argument 1: value

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Argument 0:

Buffer offset (12-bit) → 0x18

Buffer length (20-bit) → 0x10

System calls as data

```
00000000: 01 00 00 00 4a 00 00 00 18 00 00 01 10 00 00 00  ....J...........
00000010: ff 00 00 00 00 00 00 00 0a 48 65 6c 6c 6f 20 43  .........Hello C
00000020: 6f 6e 6e 65 63 74 21 00                          onnect!.
```

Argument 0:

Buffer offset (12-bit) → 0x18

Buffer length (20-bit) → 0x10

Data

System calls as data

```
00000000: 1b00 0000 4406 0000 1000 00a0 8000 0000
00000010: 0080 0000 0f00 0000 4447 0600 1001 0010
00000020: 0000 0000 0040 0000 0000 0000 0180 0000
00000030: 1e00 0000 c704 0000 0040 0000 0000 00c0
00000040: 8000 0001 0100 0000 1500 0000 a704 0000
00000050: 0140 0000 9000 0001 1000 0000 1600 0000
00000060: a764 0a00 0140 0000 a000 0001 1000 0000
00000070: 0300 0100 b000 8000 ff00 0000 0000 0000
00000080: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f
00000090: 0000 0000 0000 0000 0000 0000 0000 0000
000000a0: 4865 6c6c 6f20 436f 6e6e 6563 7421 2100
000000b0: 1000 0000 0000 0000
```

```
cryp_obj_alloc(0xa0000010, 0x80, &obj_handle);
cryp_state_alloc(0x10000110, 0, obj_handle, 0, &cryp_handle);
cryp_obj_populate (cryp_handle,
                   {c0000000, "\x00\x01[...]\x0e\x0f"}, 1);
cipher_init(cryp_handle, "\x00\x00[...]\x00\x00", 0x10);
cipher_update(cryp_handle, "Hello Connect!!\x00", 0x10,
              buf_out, &buf_out_len);
```

System calls as data

```
SYSCALL_INFO syscalls[] = {
  DEF_CALL(log,              SCN_LOG,              2)
  DEF_CALL(panic,            SCN_PANIC,            1)
  DEF_CALL(get_property,     SCN_GET_PROPERTY,     7)
  DEF_CALL(get_time,         SCN_GET_TIME,         2)
  DEF_CALL(set_ta_time,      SCN_SET_TA_TIME,      1)
  DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5)
  [...]
};
```

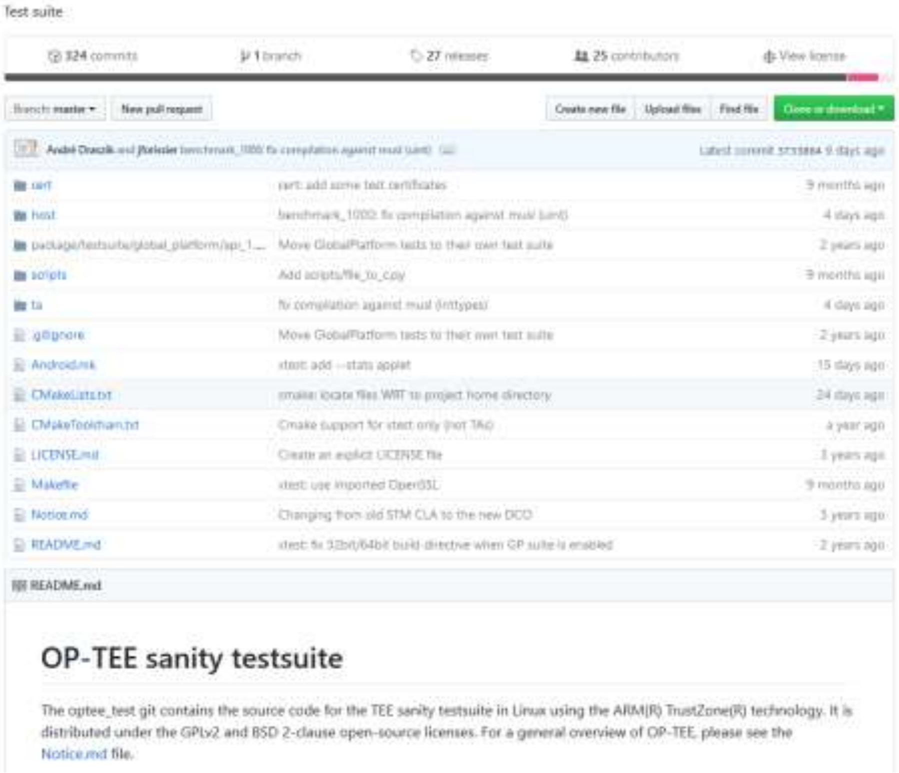
How do we give AFL a good set of inputs to start from?

Creating them by hand is very tedious...

Seeding

- Difficult for the fuzzer to explore paths without good set of inputs (corpora)

- Ideally the start set covers the full interface



André Draszik and **jforissier** benchmark_1000: fix compilation against musl (uint) … Latest commit [illegible] 9 days ago

cert	cert: add some test certificates	9 months ago
host	benchmark_1000: fix compilation against musl (uint)	4 days ago
package/testsuite/global_platform/api_1…	Move GlobalPlatform tests to their own test suite	2 years ago
scripts	Add scripts/file_to_c.py	9 months ago
ta	fix compilation against musl (inttypes)	4 days ago
.gitignore	Move GlobalPlatform tests to their own test suite	2 years ago
Android.mk	xtest: add --stats applet	15 days ago
CMakeLists.txt	cmake: locate files WRT to project home directory	24 days ago
CMakeToolchain.txt	Cmake support for xtest only (not TAs)	a year ago
LICENSE.md	Create an explicit LICENSE file	3 years ago
Makefile	xtest: use imported OpenSSL	9 months ago
Notice.md	Changing from old STM CLA to the new DCO	3 years ago
README.md	xtest: fix 32bit/64bit build directive when GP suite is enabled	2 years ago

README.md

OP-TEE sanity testsuite

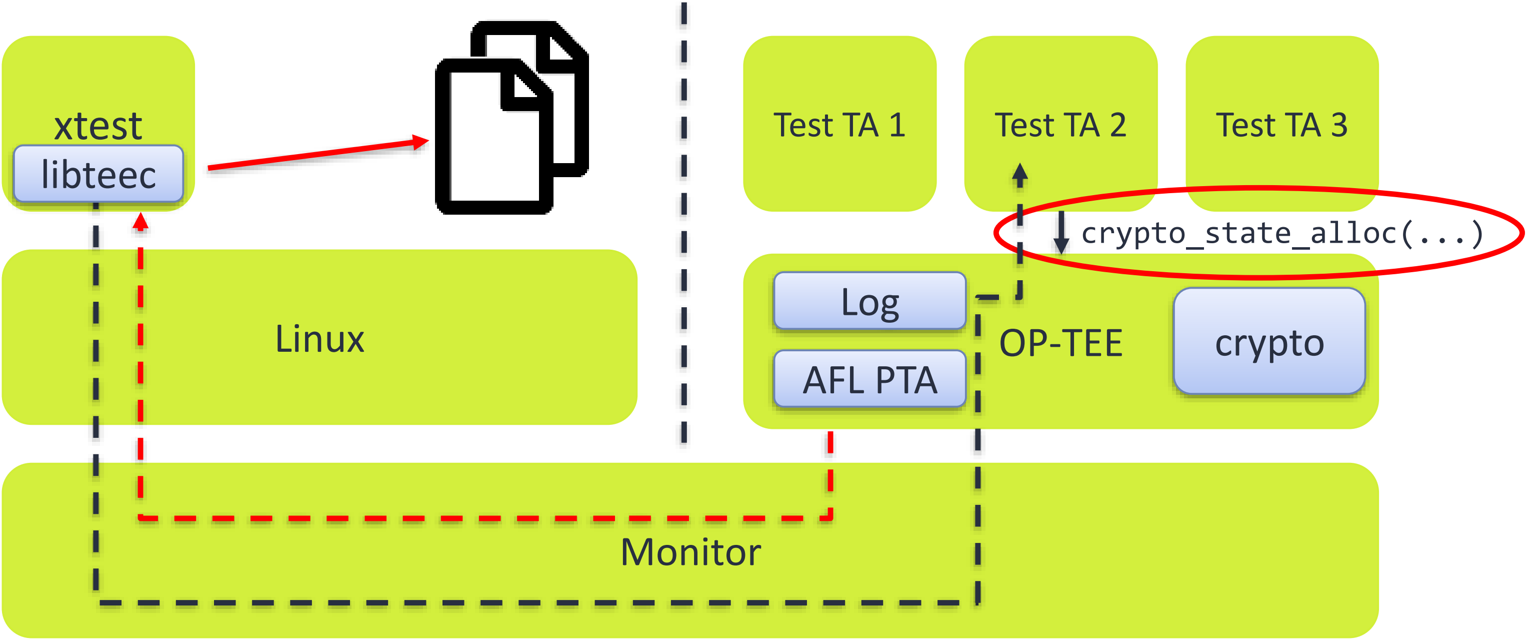
The optee_test git contains the source code for the TEE sanity testsuite in Linux using the ARM(R) TrustZone(R) technology. It is distributed under the GPLv2 and BSD 2-clause open-source licenses. For a general overview of OP-TEE, please see the Notice.md file.

Can we use the test suite to seed AFL?

Test suite

- Contains thousands of (regression) tests
- **Covers pretty much all syscalls!**

Test case → corpus



Icons © Font Awesome CC BY 4.0

Test case → corpus

1. Log raw function call
2. Extract argument semantics
3. Encode behavior, not concrete values

Test case → corpus

```
SYSCALL_INFO syscalls[] = {
  DEF_CALL(log,              SCN_LOG,              2, { ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(1), ARG_VALUE })
  DEF_CALL(panic,            SCN_PANIC,            1, { ARG_VALUE })
  DEF_CALL(get_property,     SCN_GET_PROPERTY,     7, { ARG_VALUE, ARG_VALUE, ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(3),
                                                          ARG_VALUE_INOUT_PTR, ARG_VALUE_INOUT_PTR, ARG_VALUE,
                                                          ARG_VALUE_OUT_PTR  })

  DEF_CALL(get_time,         SCN_GET_TIME,         2, { ARG_VALUE, ARG_BUF_OUT_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(set_ta_time,      SCN_SET_TA_TIME,      1, { ARG_BUF_IN_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5, { ARG_VALUE, ARG_VALUE, ARG_HANDLE, ARG_HANDLE,
                                                          ARG_HANDLE_OUT_PTR })

  [...]
};
```

```
    cryp_obj_alloc[27](a0000010, 80, 40000dfc)
     [*0x40000dfc => 1e4660]
    cryp_state_alloc[15](10000110, 0, 1e4660, 0, 40020a88)
     [*0x40020a88 => 1e44e0]
    cryp_obj_alloc[27](a0000010, 80, 40000e6c)
     [*0x40000e6c => 1e3fa0]
    cryp_obj_populate[30](1e3fa0, *40000df0:18, 1)
     attr 0 { id: c0000000, a: 40023290, b: 10 }
    cryp_obj_reset[29](1e4660)
    cryp_obj_copy[31](1e4660, 1e3fa0)
    cipher_init[21](1e44e0, *40024270:10, 10)
    cipher_update[22](1e44e0, *400222b0:10, 10, *400222b0:10,
                      40000e38=10)
```

Use expected argument count to log raw calls

Test case → corpus

```
SYSCALL_INFO syscalls[] = {
  DEF_CALL(log,             SCN_LOG,             2, { ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(1), ARG_VALUE })
  DEF_CALL(panic,           SCN_PANIC,           1, { ARG_VALUE })
  DEF_CALL(get_property,    SCN_GET_PROPERTY,    7, { ARG_VALUE, ARG_VALUE, ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(3),
                                                      ARG_VALUE_INOUT_PTR, ARG_VALUE_INOUT_PTR, ARG_VALUE,
                                                      ARG_VALUE_OUT_PTR  })

  DEF_CALL(get_time,        SCN_GET_TIME,        2, { ARG_VALUE, ARG_BUF_OUT_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(set_ta_time,     SCN_SET_TA_TIME,     1, { ARG_BUF_IN_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5, { ARG_VALUE, ARG_VALUE, ARG_HANDLE, ARG_HANDLE,
                                                      ARG_HANDLE_OUT_PTR })

  [...]
};
```

Extract argument semantics

```
cryp_obj_alloc[27](a0000010, 80, 40000dfc)
 [*0x40000dfc => 1e4660]
cryp_state_alloc[15](10000110, 0, 1e4660, 0, 40020a88)
 [*0x40020a88 => 1e44e0]
cryp_obj_alloc[27](a0000010, 80, 40000e6c)
 [*0x40000e6c => 1e3fa0]
cryp_obj_populate[30](1e3fa0, *40000df0:18, 1)
 attr 0 { id: c0000000, a: 40023290, b: 10 }
cryp_obj_reset[29](1e4660)
cryp_obj_copy[31](1e4660, 1e3fa0)
cipher_init[21](1e44e0, *40024270:10, 10)
cipher_update[22](1e44e0, *400222b0:10, 10, *400222b0:10,
                  40000e38=10)
```

Test case → corpus

```
SYSCALL_INFO syscalls[] = {
  DEF_CALL(log,              SCN_LOG,              2, { ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(1), ARG_VALUE })
  DEF_CALL(panic,            SCN_PANIC,            1, { ARG_VALUE })
  DEF_CALL(get_property,     SCN_GET_PROPERTY,     7, { ARG_VALUE, ARG_VALUE, ARG_BUF_IN_ADDR | ARG_BUF_LEN_ARG(3),
                                                        ARG_VALUE_INOUT_PTR, ARG_VALUE_INOUT_PTR, ARG_VALUE,
                                                        ARG_VALUE_OUT_PTR  })

  DEF_CALL(get_time,         SCN_GET_TIME,         2, { ARG_VALUE, ARG_BUF_OUT_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(set_ta_time,      SCN_SET_TA_TIME,      1, { ARG_BUF_IN_ADDR | ARG_BUF_SIZE(sizeof(TEE_Time)) })
  DEF_CALL(cryp_state_alloc, SCN_CRYP_STATE_ALLOC, 5, { ARG_VALUE, ARG_VALUE, ARG_HANDLE, ARG_HANDLE,
                                                        ARG_HANDLE_OUT_PTR })

  [...]
};
```

Generate behavioral description of test case

```
cryp_obj_alloc[27](a0000010, 80, 40000dfc)
 [*0x40000dfc => 1e4660]
cryp_state_alloc[15](10000110, 0, 1e4660, 0, 40020a88)
 [*0x40020a88 => 1e44e0]
cryp_obj_alloc[27](a0000010, 80, 40000e6c)
 [*0x40000e6c => 1e3fa0]
cryp_obj_populate[30](1e3fa0, *40000df0:18, 1)
 attr 0 { id: c0000000, a: 40023290, b: 10 }
cryp_obj_reset[29](1e4660)
cryp_obj_copy[31](1e4660, 1e3fa0)
cipher_init[21](1e44e0, *40024270:10, 10)
cipher_update[22](1e44e0, *400222b0:10, 10, *400222b0:10,
                  40000e38=10)
```

```
00000000: 1b00 0000 4406 0000 1000 00a0 8000 0000
00000010: 0080 0000 0f00 0000 4447 0600 1001 0010
00000020: 0000 0000 0040 0000 0000 0000 0180 0000
00000030: 1b00 0000 4406 0000 1000 00a0 8000 0000
00000040: 0280 0000 1e00 0000 c704 0000 0240 0000
00000050: 0000 00c0 b000 0001 0100 0000 1d00 0000
00000060: 0700 0000 0040 0000 1f00 0000 7700 0000
00000070: 0040 0000 0240 0000 1500 0000 a704 0000
00000080: 0140 0000 c000 0001 1000 0000 1600 0000
00000090: a764 0a00 0140 0000 d000 0001 1000 0000
000000a0: 0300 0100 e000 8000 ff00 0000 0000 0000
000000b0: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000
000000d0: 4865 6c6c 6f20 436f 6e6e 6563 7421 2100
000000e0: 1000 0000 0000 0000
```

Test case → corpus

```
00000000: 1b00 0000 4406 0000 1000 00a0 8000 0000
00000010: 0080 0000 0f00 0000 4447 0600 1001 0010
00000020: 0000 0000 0040 0000 0000 0000 0180 0000
00000030: 1b00 0000 4406 0000 1000 00a0 8000 0000
00000040: 0280 0000 1e00 0000 c704 0000 0240 0000
00000050: 0000 00c0 b000 0001 0100 0000 1d00 0000
00000060: 0700 0000 0040 0000 1f00 0000 7700 0000
00000070: 0040 0000 0240 0000 1500 0000 a704 0000
00000080: 0140 0000 c000 0001 1000 0000 1600 0000
00000090: a764 0a00 0140 0000 d000 0001 1000 0000
000000a0: 0300 0100 e000 8000 ff00 0000 0000 0000
000000b0: 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000
000000d0: 4865 6c6c 6f20 436f 6e6e 6563 7421 2100
000000e0: 1000 0000 0000 0000
```

```
b[0] = malloc(8);
cryp_obj_alloc(0xa0000010, 0x80, b[0]);
b[1] = malloc(8);
cryp_state_alloc(0x10000110, 0x0, *((uint32_t*)b[0]), 0x0, b[1]);
b[2] = malloc(8);
cryp_obj_alloc(0xa0000010, 0x80, b[2]);
cryp_obj_populate(*((uint32_t*)b[2]),
                  {c0000000, "\x00\x01\x02[...]\x0d\x0e\x0f"},
                  0x1);
cryp_obj_reset(*((uint32_t*)b[0]));
cryp_obj_copy(*((uint32_t*)b[0]), *((uint32_t*)b[2]));
t[1] = malloc(16);
memcpy(t[1], "\x00[...]\x00", 16);
cipher_init(*((uint32_t*)b[1]), t[1], 0x10);
free(t[1]);
t[1] = malloc(16);
memcpy(t[1], "Hello Connect!!\x00", 16);
b[3] = malloc(16);
t[4] = malloc(8);
memcpy(t[4], "\x10\x00\x00\x00\x00\x00\x00\x00", 8);
cipher_update(*((uint32_t*)b[1]), t[1], 0x10, b[3], t[4]);
```

Test case → corpus

```
* regression_6001 Test TEE_CreatePersistentObject
o regression_6001.1 Storage id: 00000001
Write trace to /tmp/trace/filevMd213
  regression_6001.1 OK
o regression_6001.2 Storage id: 80000000
Write trace to /tmp/trace/filecVP4s4
  regression_6001.2 OK
  regression_6001 OK

* regression_6002 Test TEE_OpenPersistentObject
o regression_6002.1 Storage id: 00000001
Write trace to /tmp/trace/filej3Ssal
  regression_6002.1 OK
o regression_6002.2 Storage id: 80000000
Write trace to /tmp/trace/fileS5ZNYT
  regression_6002.2 OK
  regression_6002 OK

* regression_6003 Test TEE_ReadObjectData
o regression_6003.1 Storage id: 00000001
Write trace to /tmp/trace/fileTOTS1E
  regression_6003.1 OK
o regression_6003.2 Storage id: 80000000
Write trace to /tmp/trace/fileABYL1H
  regression_6003.2 OK
  regression_6003 OK

* regression_6004 Test TEE_WriteObjectData
o regression_6004.1 Storage id: 00000001
Write trace to /tmp/trace/fileeS1ZaV
  regression_6004.1 OK
o regression_6004.2 Storage id: 80000000
```

Progress

- Working fuzzer, but…
  - Trace (in)stability
  - Code coverage
  - Performance

Trace (in)stability

- Same input should always result in same bitmap output
- However:
    - Threading
    - Interrupts
    - RPC calls
    - Global state

→ AFL thinks input results in new code path while it doesn’t!

Trace (in)stability

```c
struct tee_ta_session {
    TAILQ_ENTRY(tee_ta_session) link;
    TAILQ_ENTRY(tee_ta_session) link_tsd;
    struct tee_ta_ctx *ctx;
    TEE_Identity clnt_id;
    bool cancel;
    bool cancel_mask;
    TEE_Time cancel_time;
    void *user_ctx;
    uint32_t ref_count;
    struct condvar refc_cv;
    struct condvar lock_cv;
    int lock_thread;
    bool unlink;
#if defined(CFG_AFL_ENABLE)
    struct afl_ctx* afl_ctx;
    struct afl_svc_trace_ctx* svc_trace_ctx;
#endif
};
```

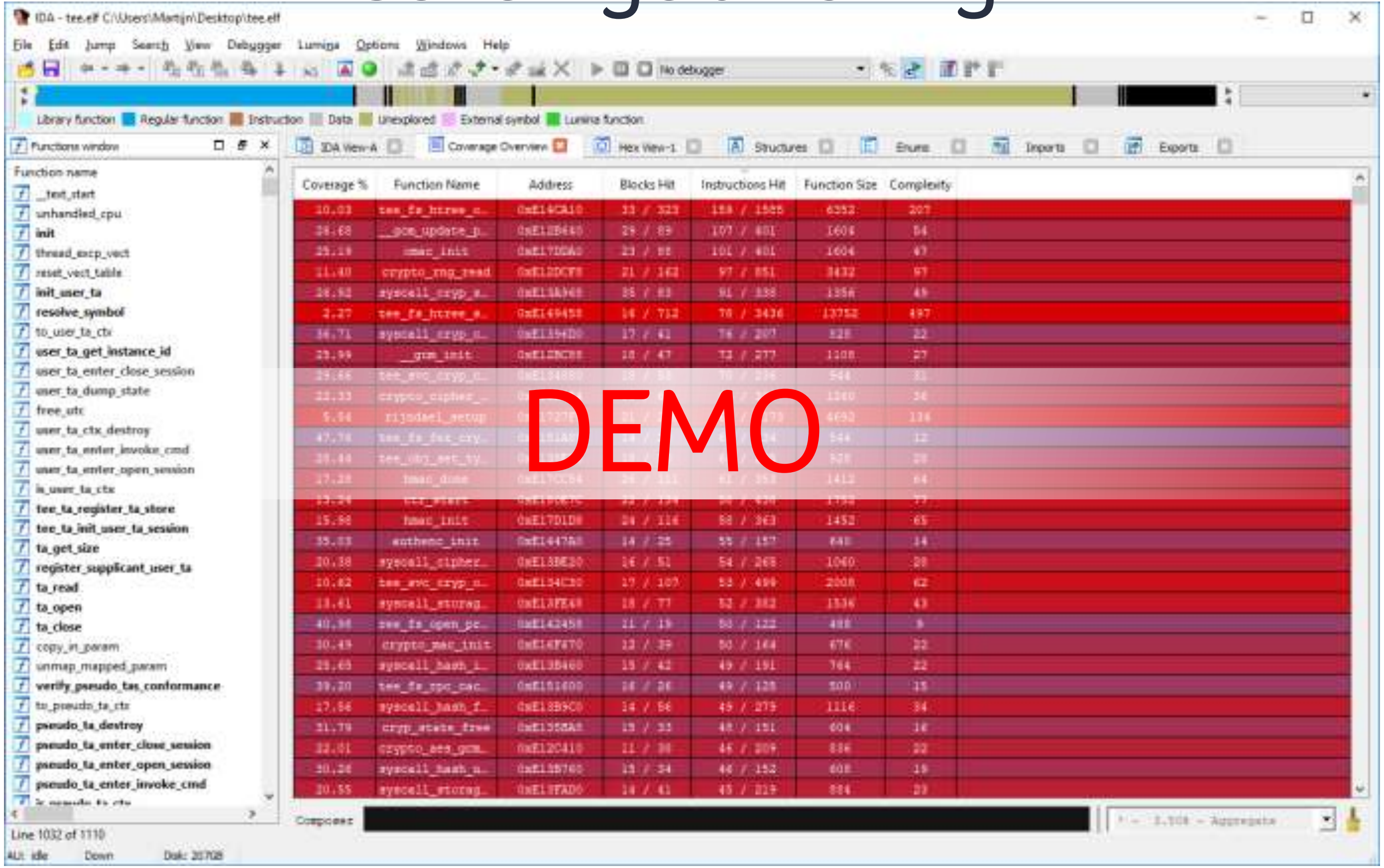
`tpidrro_el0 (MSR)`

```c
typedef struct afl_ctx {
    bool enabled;
    char bitmap[MAP_SIZE];
    uint64_t prev_loc;
};
```

But which parts did we fuzz?

We know already which parts are covered by each input.
Can we aggregate and visualize this information?

Coverage tracking



DEMO

Lighthouse plugin for IDA Pro - https://github.com/gaasedelen/lighthouse

Performance

- QEMU ~30 execs/sec
- HiKey ~250-300 execs/sec

- Ideally? 1000 - 10000 execs/sec!

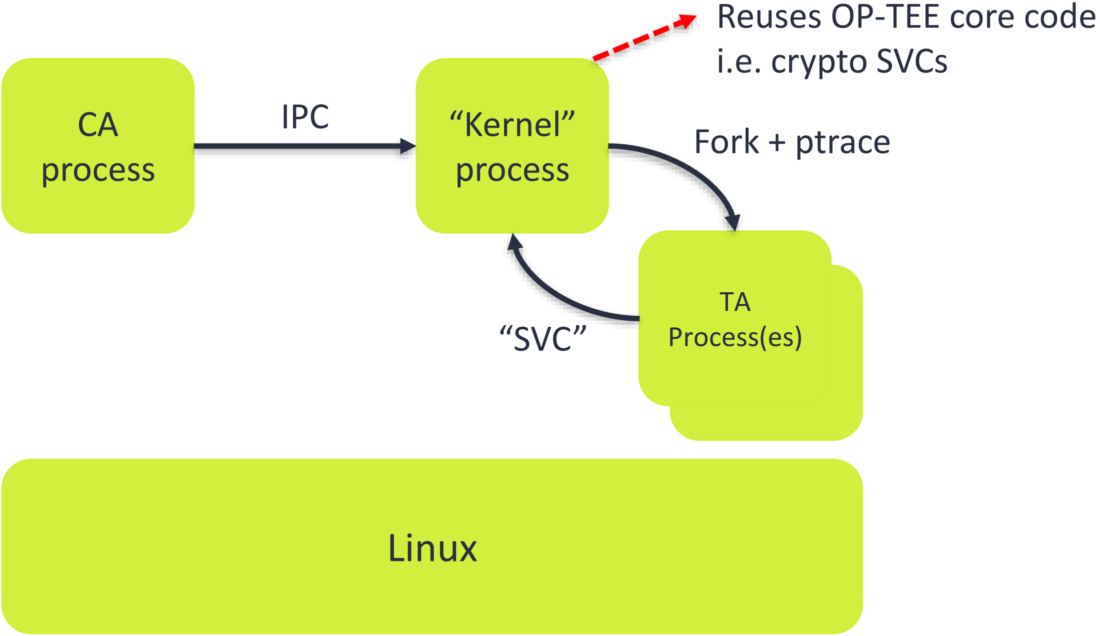
- Could we do something radically different?

About me

- Senior Security Analyst @ Riscure (2015 – now)
  - Code reviews & pentests of security critical sw components of embedded systems, hardware evals (chip design reviews, fault injection testing), sw security trainer
  - Evaluating security of (proprietary) TEEs since late 2015

- Actively researching fuzzing & simulation in last 6 years
  - Distributed fuzzing (Disfuzz AFL, 2014)
  - Fuzzing Windows drivers under Linux using kernel emulation (WKE, 2015)
  - **OP-TEE emulator for Linux (2017)**
  - Fault Injection attack simulation (FiSim, 2018)
  - Custom simulators for fuzzing & dynamic analysis of bootloaders, RTOS, Android services, …

OP-TEE emulator for Linux


Reuses OP-TEE core code
i.e. crypto SVCs

CA
process

IPC

“Kernel”
process

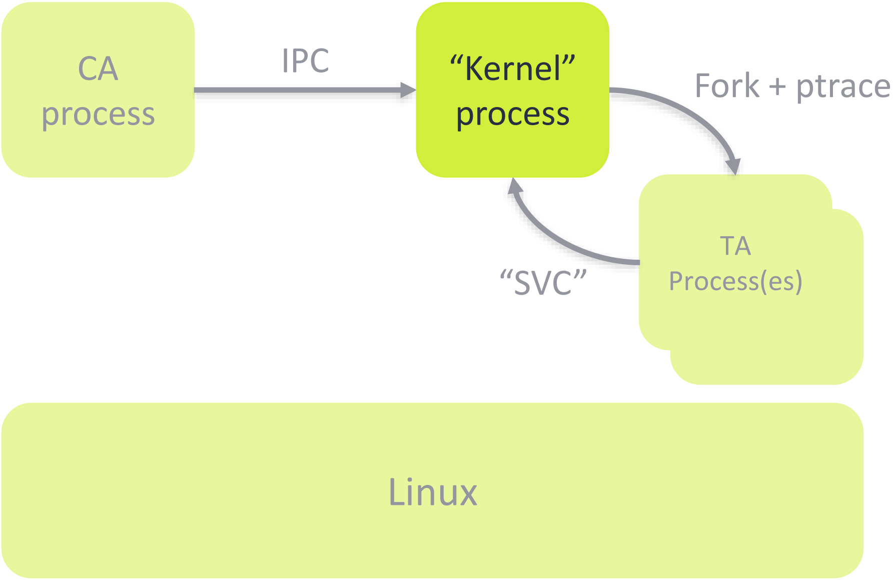
Fork + ptrace

TA
Process(es)

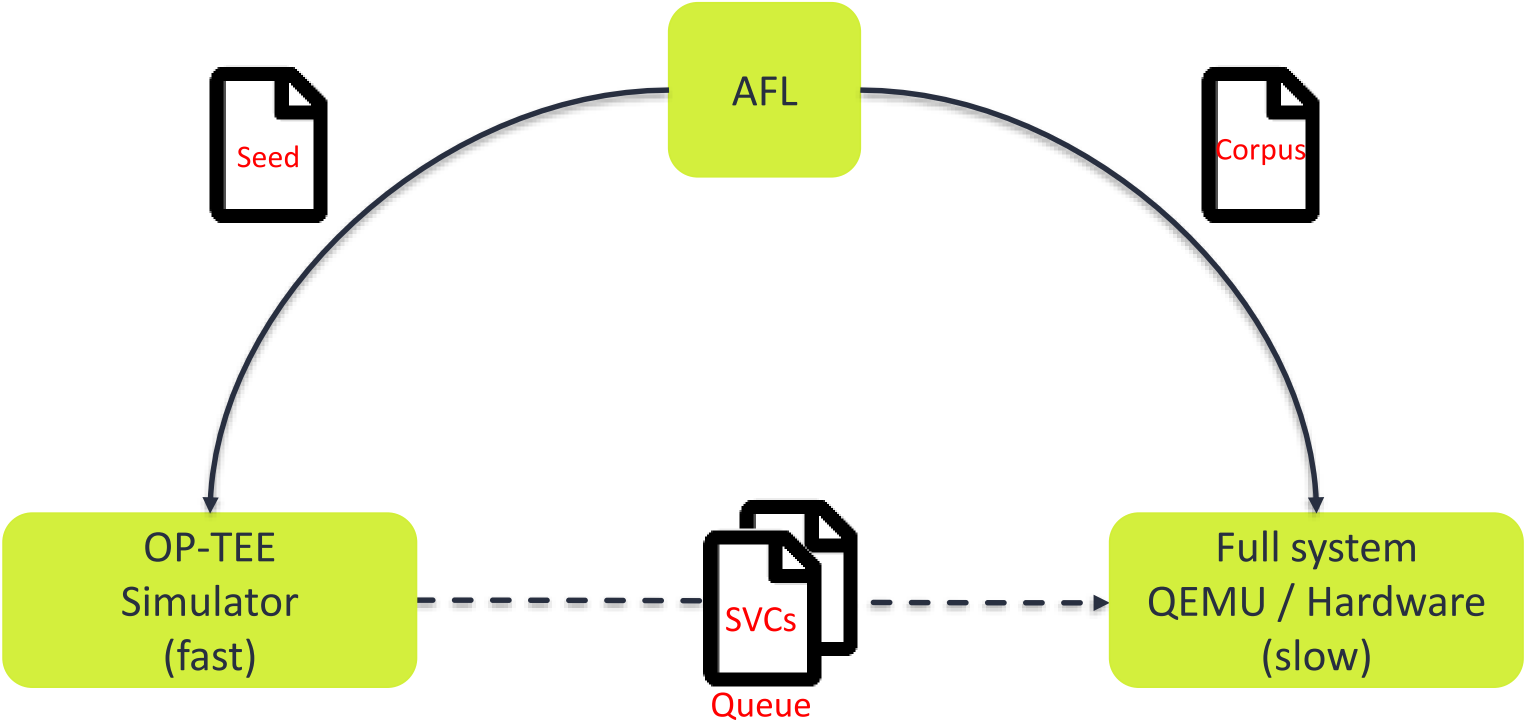
“SVC”

Linux

OP-TEE emulator for Linux



Idea


AFL
Seed
Corpus
OP-TEE
Simulator
(fast)
SVCs
Queue
Full system
QEMU / Hardware
(slow)

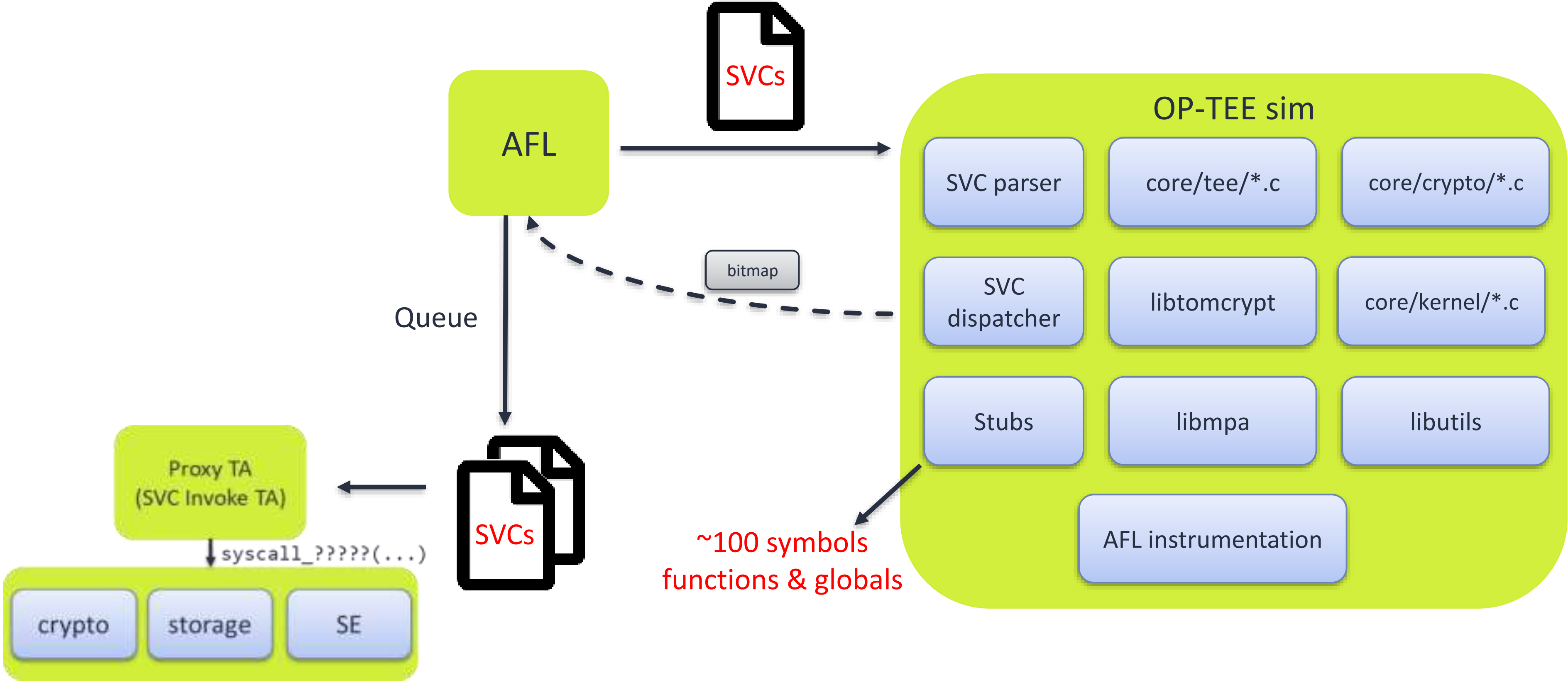
Idea

- Run OP-TEE as user-mode Linux process
- Find inputs that trigger new code paths
- Use found inputs to seed real fuzzer

Approach

- Port (almost) entire OP-TEE kernel as Linux process
- Accepts on stdin sequence of SVCs to execute
- Use AFL to find unique SVC sequences
  - Ignore crashes as they are mostly meaningless
- Use AFL queue to seed real fuzzer

Design



AFL

SVCs

OP-TEE sim

SVC parser

core/tee/*.c

core/crypto/*.c

SVC dispatcher

libtomcrypt

core/kernel/*.c

Stubs

libmpa

libutils

AFL instrumentation

bitmap

Queue

SVCs

Proxy TA (SVC Invoke TA)

`syscall_?????(...)`

crypto

storage

SE

~100 symbols functions & globals

Improving results

- Rebuild essential kernel data structures

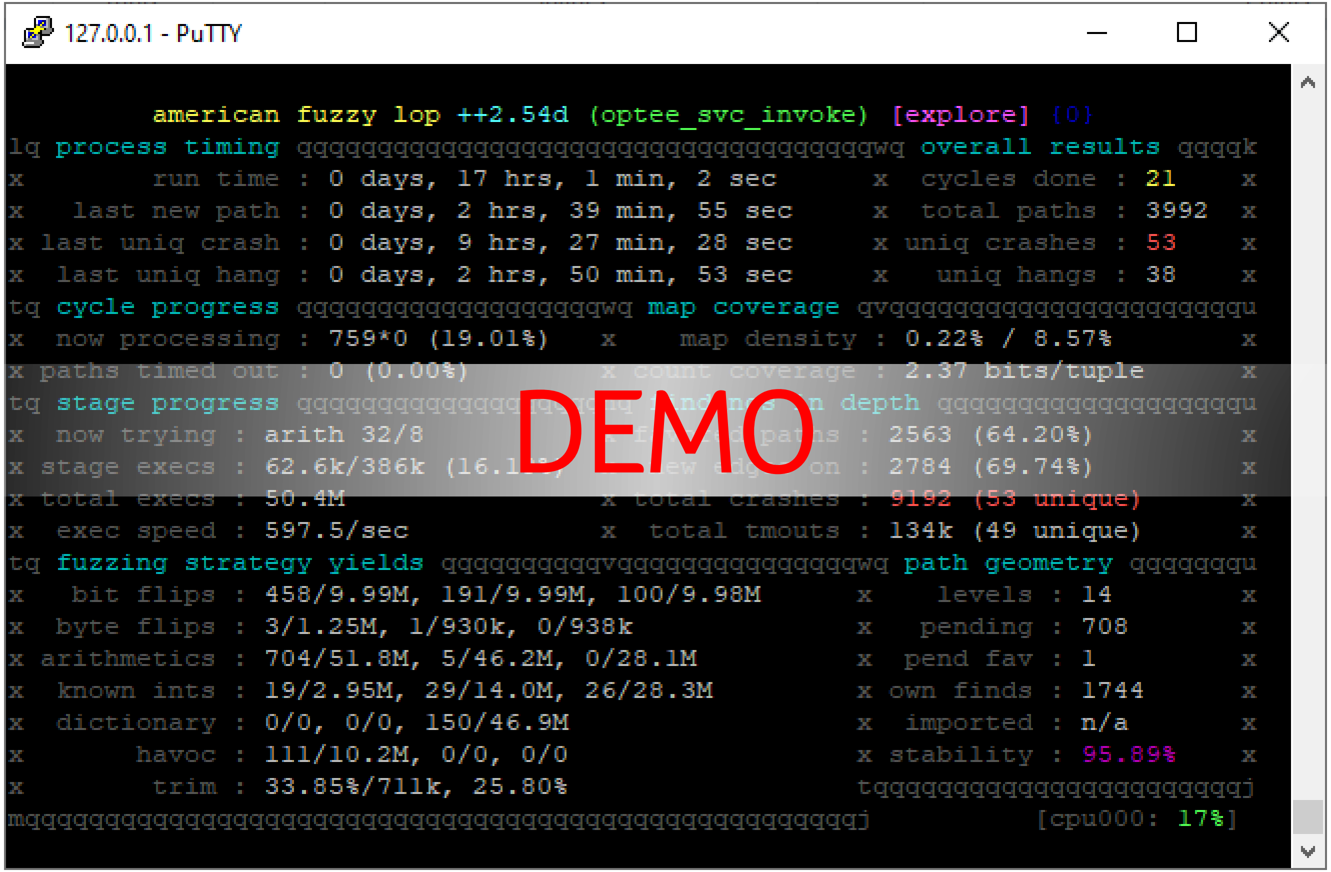
- Custom memory allocator “user-mode” memory
  - Register allocation in TA session ctx
  - Guard page after allocation (optional)
  - Unmap on free (optional)

- Optimize initialization
  - Fork loop only receives SVCs and executes them
  - Reuse process 1000x
  - Manually resetting global state (handles, objects, etc.) → deterministic exec

- Stubbing slow code that doesn’t impact behavior i.e. PRNG

- Coverage information using lcov

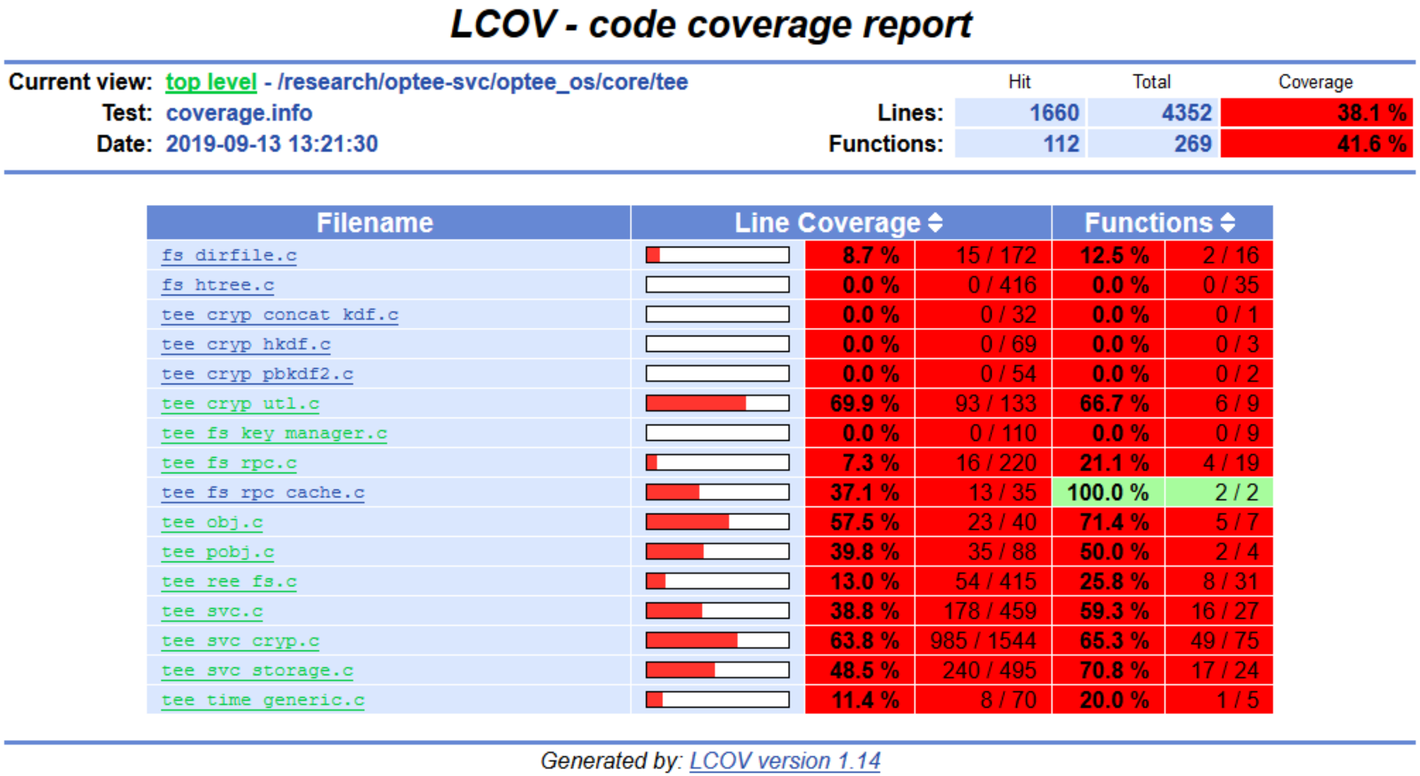
OP-TEE Sim



```
         american fuzzy lop ++2.54d (optee_svc_invoke) [explore] {0}
lq process timing qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqwq overall results qqqqk
x        run time : 0 days, 17 hrs, 1 min, 2 sec      x  cycles done : 21    x
x   last new path : 0 days, 2 hrs, 39 min, 55 sec     x  total paths : 3992  x
x last uniq crash : 0 days, 9 hrs, 27 min, 28 sec     x uniq crashes : 53    x
x  last uniq hang : 0 days, 2 hrs, 50 min, 53 sec     x   uniq hangs : 38    x
tq cycle progress qqqqqqqqqqqqqqqqqqqwq map coverage qvqqqqqqqqqqqqqqqqqqqqqqu
x  now processing : 759*0 (19.01%)   x    map density : 0.22% / 8.57%        x
x paths timed out : 0 (0.00%)        x count coverage : 2.37 bits/tuple      x
tq stage progress qqqqqqqqqqqqqqqqqqqwq findings in depth qqqqqqqqqqqqqqqqqqqu
x  now trying : arith 32/8           x favored paths : 2563 (64.20%)         x
x stage execs : 62.6k/386k (16.1     x  new edges on : 2784 (69.74%)         x
x total execs : 50.4M                x total crashes : 9192 (53 unique)      x
x  exec speed : 597.5/sec            x  total tmouts : 134k (49 unique)      x
tq fuzzing strategy yields qqqqqqqqqqvqqqqqqqqqqqqqqqwq path geometry qqqqqqqu
x   bit flips : 458/9.99M, 191/9.99M, 100/9.98M      x    levels : 14        x
x  byte flips : 3/1.25M, 1/930k, 0/938k              x   pending : 708       x
x arithmetics : 704/51.8M, 5/46.2M, 0/28.1M          x  pend fav : 1         x
x  known ints : 19/2.95M, 29/14.0M, 26/28.3M         x own finds : 1744      x
x  dictionary : 0/0, 0/0, 150/46.9M                  x  imported : n/a       x
x       havoc : 111/10.2M, 0/0, 0/0                  x stability : 95.89%    x
x        trim : 33.85%/711k, 25.80%                  tqqqqqqqqqqqqqqqqqqqqqqqj
mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj          [cpu000: 17%]
```

DEMO

lcov



LCOV - code coverage report

Current view: top level - /research/optee-svc/optee_os/core/tee
Test: coverage.info
Date: 2019-09-13 13:21:30

	Hit	Total	Coverage
Lines:	1660	4352	38.1 %
Functions:	112	269	41.6 %

Filename	Line Coverage			Functions	
fs_dirfile.c		8.7 %	15 / 172	12.5 %	2 / 16
fs_htree.c		0.0 %	0 / 416	0.0 %	0 / 35
tee_cryp_concat_kdf.c		0.0 %	0 / 32	0.0 %	0 / 1
tee_cryp_hkdf.c		0.0 %	0 / 69	0.0 %	0 / 3
tee_cryp_pbkdf2.c		0.0 %	0 / 54	0.0 %	0 / 2
tee_cryp_utl.c		69.9 %	93 / 133	66.7 %	6 / 9
tee_fs_key_manager.c		0.0 %	0 / 110	0.0 %	0 / 9
tee_fs_rpc.c		7.3 %	16 / 220	21.1 %	4 / 19
tee_fs_rpc_cache.c		37.1 %	13 / 35	100.0 %	2 / 2
tee_obj.c		57.5 %	23 / 40	71.4 %	5 / 7
tee_pobj.c		39.8 %	35 / 88	50.0 %	2 / 4
tee_ree_fs.c		13.0 %	54 / 415	25.8 %	8 / 31
tee_svc.c		38.8 %	178 / 459	59.3 %	16 / 27
tee_svc_cryp.c		63.8 %	985 / 1544	65.3 %	49 / 75
tee_svc_storage.c		48.5 %	240 / 495	70.8 %	17 / 24
tee_time_generic.c		11.4 %	8 / 70	20.0 %	1 / 5

Generated by: LCOV version 1.14

Limitations

- RPC mechanism not implemented
  - Cripples secure storage

- No TA support
  - No fuzzing of TA loader, cross TA calls

- Several mechanisms stubbed
  - Mutexes, spinlocks, threads, …

Results

- 10x more unique inputs

- Up to 10x faster
    - 15 exec/s (doing real crypto) – 2500+ exec/s per CPU core
    - Scales on SMP systems!

- Found several new bugs!
    - Some issues are much easier to debug / find with the simulator!

Current status

- Most code open-sourced
  - Repository: https://github.com/Riscure/optee_fuzzer

- Upstreaming difficult / infeasible
  - Requires invasive core modifications
  - Licensing

- Needs to be rebased against latest version

- More things to triage than time allows ☹

Thank you! Any questions?

Martijn Bogaard
Senior Security Analyst
martijn@riscure.com / @jmartijnb