Benchmark and Profiling in OP-TEE

Igor Opaniuk
Jerome Forissier
1. OP-TEE Latency benchmark

This part of the presentation introduces new OP-TEE feature: Latency Benchmark

- Quick introduction to latency benchmarks in OP-TEE
  - What it’s used for and what kind of measurements?
  - How can received results be used in OP-TEE development?
- Architecture and implementation
  - Basic prerequisites and main concerns
  - Internals: what’s going on and implementation details
  - Limitations, further tasks & “must-haves”
- Usage of optee_benchmark application
  - Usage examples
  - QEMU demo
What it’s used for?

- Due to its nature, OP-TEE is being a solution spanning over several architectural layers, where each layer contains its own complex parts. For further optimizations of such solution we need to have a tool for basic profiling of each call and provide time spent in each OP-TEE layer.

- Measurements of:
  - The roundtrip time for “dumb” TA
  - Amount of time to go through each layer:
    - libTEEC -> OP-TEE KMOD
    - OP-TEE KMOD -> OP-TEE OS core
    - OP-TEE OS core -> LibUTEEM
    - The same way back
Basic prerequisites and main concerns

- Must be minimally intrusive as it could be
- Avoid adding any additional API, which goes against the GlobalPlatform specification
- Common “time source” for all OP-TEE layers with a minimum latency when reading values:
  - ARM Performance Monitoring Units (PMU)
  - ARM Generic timers
- Common storage of timestamps for all OP-TEE layers:
  - Usage of GP 4 params/ extending to 5 GP params (TEE_Params for bypassing memrefs/values to TA)
  - Linux memory carveout
  - Dedicated part of OP-TEE (Pseudo TA), which is responsible for handling timestamp buffers
- Providing some statistics with min/max/average results.
Implementation overview

- Benchmark app opens a session to the Pseudo TA, which is responsible for handling timestamp/statistics buffers
- Benchmark app allocates shared memory for both timestamp/statistics buffers, then performs `BENCHMARK_CMD_REGISTER_MEMREF` to Benchmark PTA
- OP-TEE Kernel module checks for `BENCHMARK_CMD_REGISTER_MEMREF` and hijacks memrefs and saves it
- Finally, Benchmark PTA receives memrefs, stores them (now they are accessible within all OP-TEE OS core subsystems)
- OP-TEE benchmark app forks `client app`, bypassed via params and forks/execs it
- LibTEEC of child client application checks for Benchmark mode via `BENCHMARK_CMD_GET_MEMREF` and mmaps physical addresses received back from PTA
- After each InvokeCommand invocation statistics buffer is updated based on fullfilled timestamps
- OP-TEE benchmark app waits until child client app exits, then analyses and prints results, and deregister benchmark via `BENCHMARK_CMD_UNREGISTER` call to PTA
Implementation overview

- **Benchmark app**
- **libTEEC**
- **OP-TEE KMOD**
- **Secure World**
  - **Client TA**
  - **libUTEE**
  - **Benchmark Pseudo TA**
  - **OP-TEE OS Core**
- **Normal World**
  - **Client app**
  - **Benchmark_aux**

Steps:
1. BENCHMARK_CMD_MD_REGISTER_MEMREF
2. fork/exec
3. InvokeCommand
4. BENCHMARK_CMD_GET_MEMREF
5. resume Invoke

Hijack timestamp memref

EL0

EL1
Benchmark usage

- Build OP-TEE with `CFG_TEE_BENCHMARK=y` (also could be set in build/common.mk)
  
  `$ make all CFG_TEE_BENCHMARK=y -j4$

- Run benchmark app on the target env, bypass a client app path and its params, which you are going to benchmark:
  
  `# benchmark client_app [client_app params]`

  Example:

  `# benchmark /bin/xtest 10001`
Benchmark usage

```plaintext
0 regression_10001.16 PBKDF2 RFC 6070 6 (HMAC-SHA1)
  regression_10001.16 OK
  regression_10001 OK
+
Result of testsuite regression filtered by "10001":
  regression_10001 OK
+
225 subtests of which 0 failed
1 test case of which 0 failed
73 test cases was skipped
TEE test application done!
4. Done benchmark. Printing latency statistics:
```

<table>
<thead>
<tr>
<th>OP-TEE Layer1(vaddress) -&gt; OP-TEE Layer2(vaddress) latency</th>
<th>Min CPU cycles</th>
<th>Max CPU cycles</th>
<th>Avg CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT(0x00000000076f1d620) -&gt; KMOD(0x000000000804c8ca0)</td>
<td>67008</td>
<td>1994624</td>
<td>188288</td>
</tr>
<tr>
<td>KMOD(0x000000000804c8ca0) -&gt; CORE(0x0000000000e018968)</td>
<td>25920</td>
<td>240320</td>
<td>45120</td>
</tr>
<tr>
<td>CORE(0x000000000e018968) -&gt; CORE(0x0000000000e018aae)</td>
<td>174976</td>
<td>7764896</td>
<td>403392</td>
</tr>
<tr>
<td>CORE(0x000000000e018aae) -&gt; KMOD(0x000000000804cbd40)</td>
<td>1984</td>
<td>114112</td>
<td>30848</td>
</tr>
<tr>
<td>KMOD(0x000000000804cbd40) -&gt; CLIENT(0x00000000076f1d620)</td>
<td>30976</td>
<td>10233344</td>
<td>251840</td>
</tr>
</tbody>
</table>
```
Limitations & “must-haves”

- Measurement of interrupt latencies:
  - Execution in NW - a secure interrupt comes
  - Execution in SW - a secure interrupt comes
  - Execution in SW - a public interrupt comes

- Benchmark fast-calls and compare with std-calls latency
- And, finally, put timestamps anywhere in OP-TEE
- Benchmark multiple applications at once
- Export statistics to file for further analyzing
2. Profiling Trusted Apps with gprof

This part of the presentation introduces a new OP-TEE feature: gprof support for Trusted Applications

- Quick introduction to gprof
  - What kind of information does gprof provide? How does it help?
  - Usage (demo)
- How to use gprof with OP-TEE Trusted Applications
  - Usage (QEMU demo)
- Architecture and implementation
Learn the basics of application profiling!

gprof FOR DUMMIES

A quick tour of a 34-year-old tool that’s still useful today!

A Reference for the Rest of Us!

By Linaro
Key information provided by gprof

- **Function calls**
  - How many times does the program call `foo()`?
  - Who are the callers of `foo()`?
  - How many times each caller calls the function?
  - Who are the callees of `foo()`?
  - How many times each callee is called?
    - *Useful to detect if something is wrong with the test case and/or the algorithm*

- **Time spent, call graph view**
  - How much time is spent in function `foo()`? In its callees?
  - How much time spent in `foo()` is due to caller `bar()`?
    - *Get a better understanding of the call flow*

- **Time spent, flat profile view**
  - Which functions eat up the most CPU time? (ordered, with cumulative time)
    - *Useful to detect which functions should be optimized in priority*
Example: Linux application

main()
  2 s.

foo(2)
  2 s.

bar()
  3 s.

foo(1)
  1 s.

x3
Gprof output

main()

foo(2)

bar()

foo(1)

2 s.

3 s.

1 s.

x3
Profiling OP-TEE Trusted Applications

TAs may be profiled in a very similar way to regular Linux apps.

1. Make sure OP-TEE is compiled with gprof support
2. Compile the TA with -pg
   (QEMU) make CFG_TA_GPROF_SUPPORT=y CFLAGS_ta_arm32=-pg
3. Run the TA
   Profile data are written to /tmp/gmon-<uuid>.out
4. Run gprof, passing the TA ELF file and the profile data
   U=8aaaf200-2450-11e4-abe20002a5d5c51b
   gprof $U.elf /tmp/gmon-$U.out
Example: OP-TEE Trusted Application

```
#include <tee_internal_api.h>
#include <trace.h>

#define TIME_MS() (TIE_TimeType(t); t.seconds *= 1000; t.millis;)

#define SPIN(sec) do {
    uint64_t t0 = TIME_MS();
    while ((TIME_MS()) < t0 + sec * 1000) {
        for (_l = 0; _l < 10000; _l++)
    }
} while(0)

static void foo(int sec)
{
    IMG2("foo");
    SPIN(5);  
    for (i = 0; i < _l; i++)
        foo();
}

static void bar(void)
{
    int t;
    IMG2("bar");
    SPIN(3);
    for (i = 0; i < _l; i++)
        foo();
}

TEE_Result TA_InvokeCommandEntryPointPoint(void __unused *sess_ctx,
                                        uint32_t __unused cmd_id,
                                        uint32_t __unused param_types,
                                        TEE_Param __unused params[1])
{
    IMG2("TA_InvokeCommandEntryPointPoint");
    foo();  
    bar();
    return TEE_SUCCESS;
}
```
Gprof output

TA_InvokeCommandEntryPoint()

foo(2) 2s.

bar() 3s.

foo(1) 1s.

x3
Implementation overview

- Profiling buffer is allocated in user space inside the .bss section of the TA by the linker script
  - The DEFINED() and PROVIDE() commands of GNU ld are used to detect if the TA is instrumented, and if so, allocate space for the profiling buffer in .bss and connect the profiling hooks to our implementation
- Call graph recording occurs in user space (secure PL0/EL0)
- PC sampling occurs in “kernel” space (secure PL1/EL1)
- Profiling starts when the first session to the TA is opened
  - A pseudo-TA is invoked by libutee to start PC sampling
- Profiling ends when the last session is closed
  - The pseudo-TA is invoked to stop PC sampling, then send the profile data to Normal World
  - tee-suppllicant saves the profile to disk
Commits

- optee_os 883c4be3d11c ("Add support for user TA profiling with gprof (-pg)")
  27 files changed, 1565 insertions(+), 37 deletions(-)
- optee_client 3b4c507ce588 ("Add support for TA profiling (gprof)"")
  6 files changed, 206 insertions(+)
- Merged in master branches (not in OP-TEE 2.3.0 release)
Implementation: call graph recording

- Quite similar to the “standard” implementation (Linux libc), done in libunwinder
- Relies on instrumentation of the application
- GCC flag: -pg
- Compiler inserts a call to _mcount() or __gnu_mcount_nc() into each function’s prologue

<table>
<thead>
<tr>
<th>example.o: file format elf32-littlearm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 &lt;foo&gt;:</td>
</tr>
<tr>
<td>0:  e92d 4ff0  stmdb sp!, {r4, r5, r6, ...</td>
</tr>
<tr>
<td>4:  b08d  sub  sp, #52 ; 0x34</td>
</tr>
<tr>
<td>6:  af00  add  r7, sp, #0</td>
</tr>
<tr>
<td>8:  b500  push  {lr}</td>
</tr>
<tr>
<td>a:  f7ff fffe  bl 0 &lt;__gnu_mcount_nc&gt;</td>
</tr>
<tr>
<td>e:  60f8  str  r0, [r7, #12]</td>
</tr>
<tr>
<td>10: f240 0300  movw  r3, #0</td>
</tr>
<tr>
<td>14: f2c0 0300  movt  r3, #0</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>example.o: file format elf64-littleaarch64</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000 &lt;foo&gt;:</td>
</tr>
<tr>
<td>0:  a9bb7bfd  stp  x29, x30, [sp,-80]</td>
</tr>
<tr>
<td>4:  910003fd  mov  x29, sp</td>
</tr>
<tr>
<td>8:  aa1e03e1  mov  x1, x30</td>
</tr>
<tr>
<td>10: aa0103e0  mov  x0, x1</td>
</tr>
<tr>
<td>14: 94000000  bl 0 &lt;__mcount&gt;</td>
</tr>
<tr>
<td>18: 90000000  adrp  x0, 0 &lt;__stack_chk_guard&gt;</td>
</tr>
<tr>
<td>1c: 91000000  add  x0, x0, #0x0</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Implementation: call graph recording

- libu tee provides 32- and 64-bit implementations of these tracing functions, called `__utee_mcount()`
- The assembly code adjusts the LR values to reflect the address of the calling site and branches to a C function: `__mcount_internal()`
- The count for the corresponding arc of the call graph, i.e., the `{callee, caller}` pair, is incremented in the profiling buffer
- Further processing is done later by the gprof command
Implementation: PC sampling

- TAs are interrupted periodically (timer interrupt). Control is transferred to the OP-TEE kernel, then possibly to the Normal World kernel, then eventually to the TA again.
- The address the TA was executing when it was interrupted is recorded, and a counter for this location is incremented. Assuming periodic sampling, the weight associated with each address is proportional to the time spent executing that location. PC (Program Counter) sampling data are written directly into the userspace buffer by the kernel code.
- The time spent executing the TA is tracked so that the sampling rate is known, and sample values can be converted to seconds by gprof.
Implementation overview

Normal World

- tee-supplicant
- gprof.c

Secure World

- Profiling buffer (.bss)
- PC samples
- Call arcs
- Trusted App code
- libutee
- gprof.c
- gprof_pta.c
- RPC
  - thread_rpc_cmd()
- Task switching
  - thread_state_suspend()

Linux kernel

OP-TEE core

EL0

EL1

Gmon-*.out

Call arcs
Limitations

● Time spent in the kernel (TEE core) is not reported
  ○ This is by design
  ○ It might be possible to charge kernel time on the syscall wrappers (utee_*) -- for further study

● Some functions may be reported with a call count of zero
  ○ Typically: TEE API functions (TEE_*, TA_*), syscall wrappers (utee_*), libc [libutils] calls (memset…), or any function from files not compiled with -pg
  ○ Due to statically linking os libutee/libutils/libmpa. Functions are used (the PC sampling code finds them) but the call graph can not record them since they are not instrumented by default
  ○ The Linux example doesn’t report any such functions because libc is a shared library

● Some functions might be incorrectly identified
  ○ May cause incorrect time or call count to be reported
  ○ Does not occur when optimizations are disabled (-O0)
  ○ Bug in the PC recording code, or inherent issue with optimization?
Thank You

#BUD17
For further information: www.linaro.org
BUD17 keynotes and videos on: connect.linaro.org