TROOPERS

Securing the Airwaves

Emulation, Fuzzing, and Reverse Engineering of iPhone Baseband Firmware

Bruno, Luca, Rachna {bruno,glockow,rachna}@srlabs.de

Security Research Labs

Nice to meet you :)

Bruno Produit

- Background in low-level security and cryptography
- Experience in fuzzing, telco and hardware hacking
- GitHub @brunoproduit

Luca Glockow

- Background in application and device security
- Experience in hacking hard- and software in telco
- GitHub @luglo

Rachna Shriwas

- Background in device testing, fuzzing and code assurance
- GitHub @rachsrl

We start where other research projects stopped

> *Unlike other baseband implementations, Qualcomm leverages a fully-custom architecture known as Hexagon […]* ***Unfortunately, tooling for this architecture is sparse, and especially full-system emulators are lacking****.*

Hernandez et al. 2022, “FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware”

We investigate the other chip everyone has in their pocket

Our goals

Create Transparency

Understand, document, and share Hexagon security insights

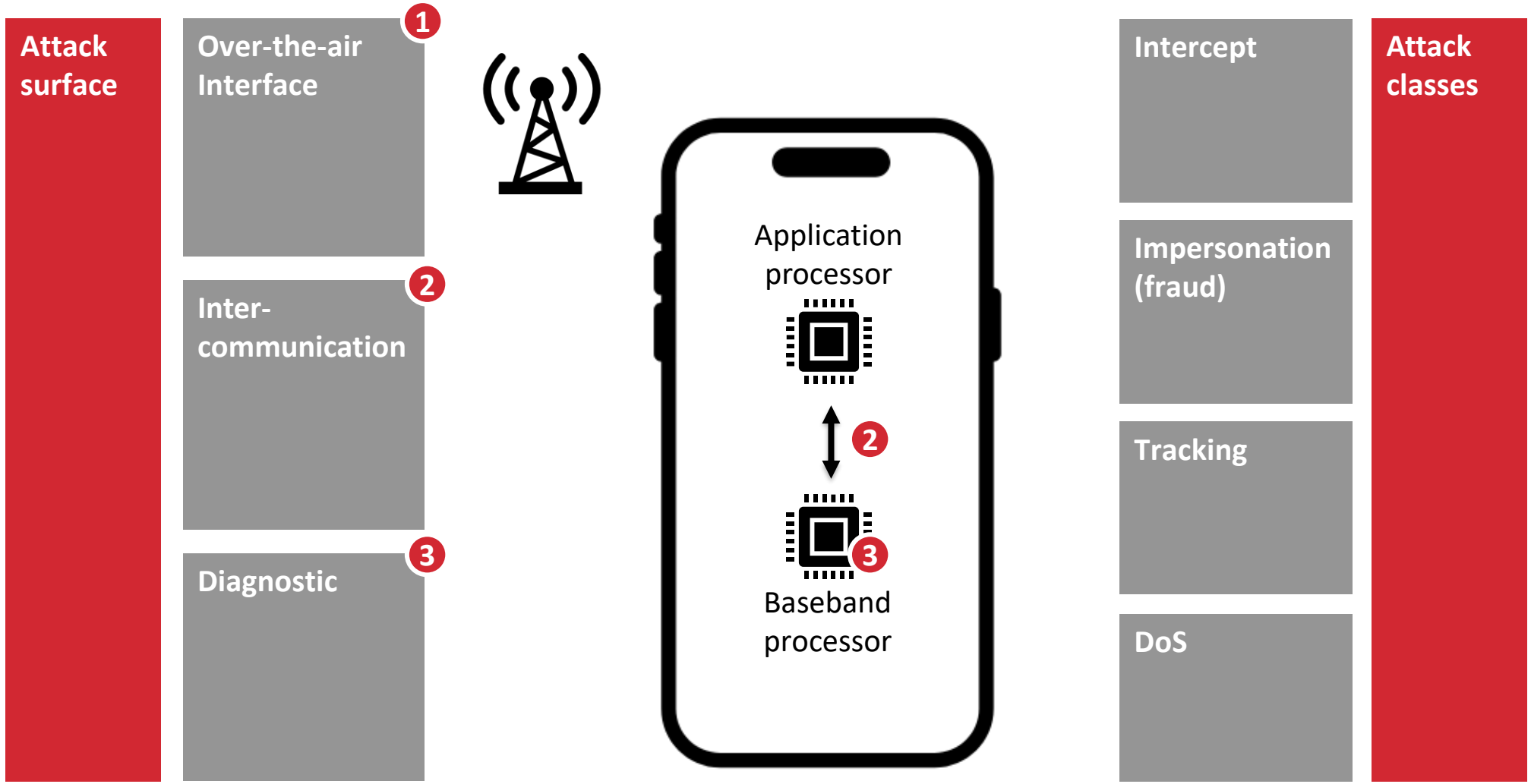
Enable vulnerability research

Help secure Qualcomm-based phones against exploitation

Release Tooling

Open source fuzzing setup and tooling for Hexagon basebands

The Qualcomm baseband exposes multiple interfaces, making it vulnerable to diverse attacks



Existing baseband research does not cover Hexagon baseband full system emulation

		Emulation	Fuzzing	Hexagon
Reverse Engineer	▪ **Reversing Hexagon:** Burke 2018 [1] ▪ **Reversing DIAG:** Esage 2020 [2]			✓
Hardware Fuzz	▪ **Fuzzing on Hexagon hardware:** Gong & Zhang 2021 [3]		✓	✓
Emulation Fuzz	▪ **Advanced rehosted baseband fuzzing:** Maier et al 2020 [4], Hernandez et al 2022 [5]	✓	✓	
Emulation Hexagon Fuzz	**The gap we want to fill**	✓	✓	✓

[1] Burke 2018, A Journey into Hexagon
[2] Esage 2020, Advanced Hexagon DIAG
[3] Gong & Zhang 2021, In-Depth Analyzing and Fuzzing for Qualcomm Hexagon Processor
[4] Maier et al. 2020, BaseSAFE: baseband sanitized fuzzing through emulation
[5] Hernandez et al. 2022, FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware

Agenda

Introduction: Hexagon baseband

From research to tooling

Demo: Fuzzing Hexagon

Opening up baseband security

Hexagon is a whole new DSP-focussed CPU architecture, prompting research from the ground again

Challenges for researchers	
Custom CPU architecture	▪ **Lacks support** in most tooling ▪ **Tedious** to reverse
Custom OS and software stack	▪ **Proprietary** tooling ▪ **Adaptation** of common attack techniques required
Custom communication protocols	▪ **No free** tooling or documentation exists

Design Highlights

- **Optimized for parallel execution** (VLIW)
- **DSP-specialized** (vs general CPU)
- **Quirky registers** (chicken, duck, goose…)

```
6    #ifndef _HEXAGONTYPES_H_
473  typedef enum global_register_t
502      G_REG_ACC1,
503      G_REG_CHICKEN,
504      G_REG_STFINST,
```

- **Privilege mode separation**: monitor, guest, user
- **Instruction packets**, group of parallel instructions

```
A2_addi
{
J2_call
A2_tfrsi
}
J2_cmpeq
```

Agenda

Introduction: Hexagon baseband

From research to tooling

Demo: Fuzzing Hexagon

Opening up baseband security

We need the firmware and a customized emulator to run Hexagon on a different machine

QEMU:
Emulate baseband on unrestricted hardware

Ghidra:
Integrate Hexagon processors in Ghidra

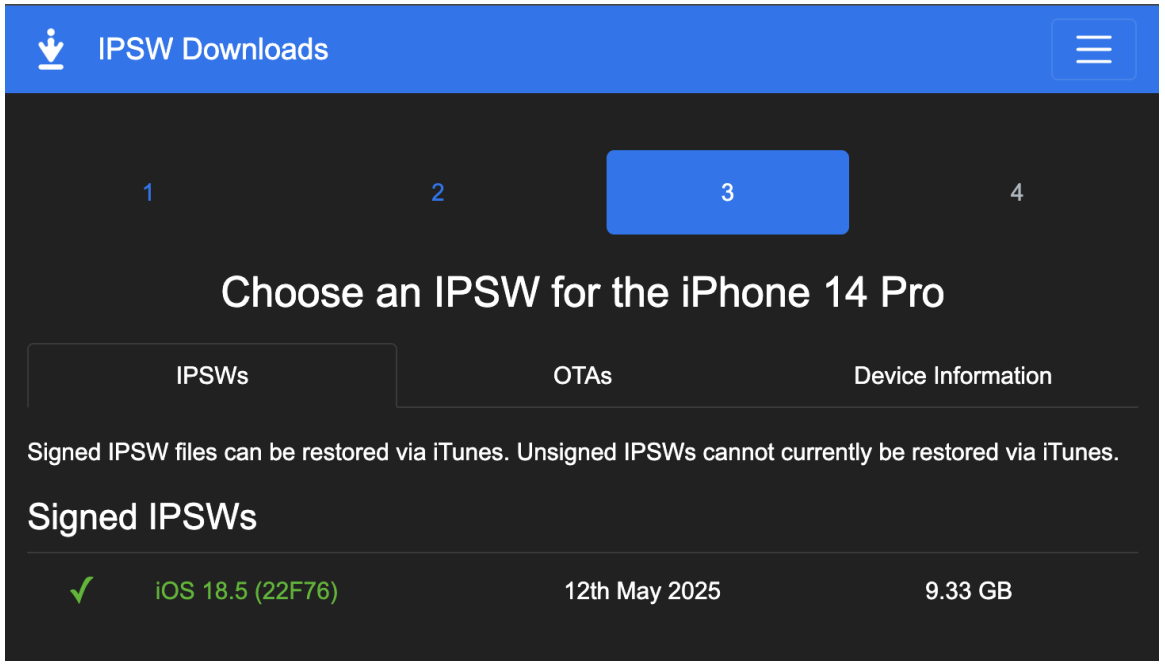
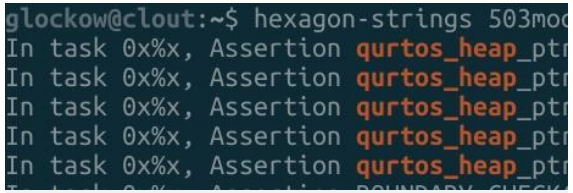
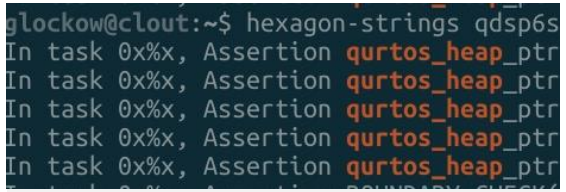
Clade:
Reverse-engineer compression algorithm

LibAFL:
Coverage guided fuzzing with QEMU

Firmware Harness:
Hook subsystem for analysis & fuzzing

Obtain firmware → **Get full system Emulation** → **Patch for our device**

The firmware is the initial step in our emulation effort

Step	Details	Screenshots
Obtain firmware	▪ **Download publicly accessible iOS IPSW firmware files (e.g. from ipsw.me)** ▪ Extract baseband bundle from the IPSW archive and **identify QUALCOMM DSP6 executable (qdsp6sw.mbn)**	
Analyze firmware	▪ **Identical strings** confirm, firmware shares code ▪ **Build on the same Qualcomm Hexagon SDK**	 *Router firmware*  *iPhone firmware*

Finding (Q)emu: Qualcomm works on Hexagon full system emulation

Step	Details	Screenshots
Find full system emulator	▪ **hexagon-softmmu not supported** by QEMU ▪ **QEMU fork** maintained by Qualcomm Innovation Center ▪ **hexagon-softmmu implementation** on WIP branch	
Run emulator	▪ **Compile and run** qemu-system-hexagon `$ qemu-system-hexagon -monitor stdio -display none -kernel qdsp6sw.mbn`	
Progress boot	▪ **More logs appear** in reversed firmware code	

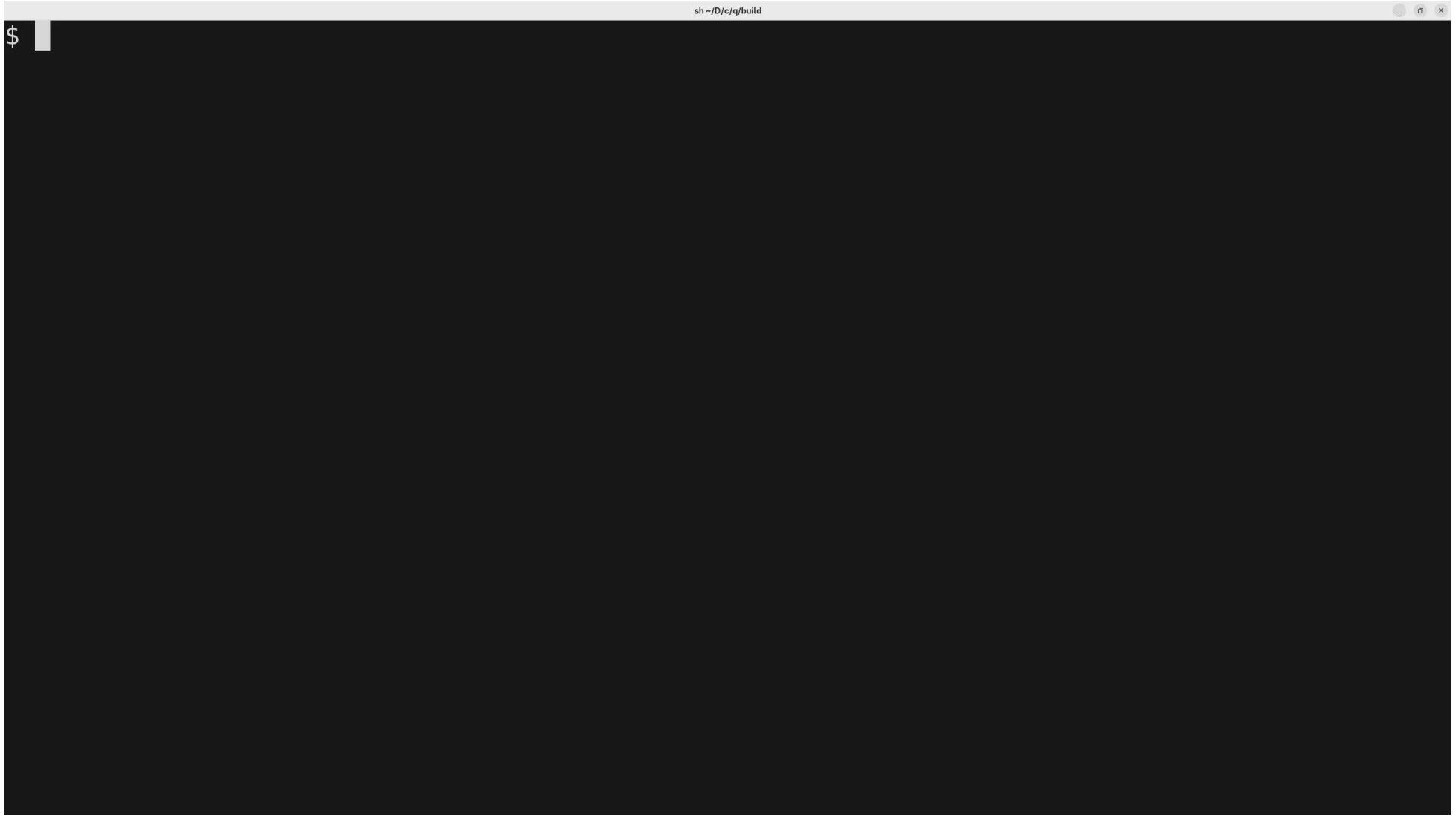
```
QEMU 10.0.50 monitor - type 'h
(qemu) QURT kernel started
QURT kernel init cache params
```

Boot logs in QEMU

```
qurt_printf(s_QURT_kernel_started_fe11a57e);
prob_set_boot_status(0x19);
iVar8 = qurt_printf(s_QURT_kernel_init_cache
FUN_fe112a10(iVar8,param_2,iVar7,ppuVar13,para
uVar6 = FUN_fe118a20();
uVar4 = 0;
DAT_fe108300 = 1;
some_struct_contruction_subtree
          (uVar6,param_2,-0x1ef7d00,ppuVar13,p
           (char)in_R31,param_7);
uVar11 = (undefined)param_2;
prob_set_boot_status(0x1a);
iVar7 = qurt_printf(s_QURT_root_task_started_
```

Boot logs in decompiled code

Without QEMU modification, firmware boot fails with only two lines of output



We control the machine but not its software - reversed bug hunting

Obtain firmware → Get full system Emulation → Patch for our device

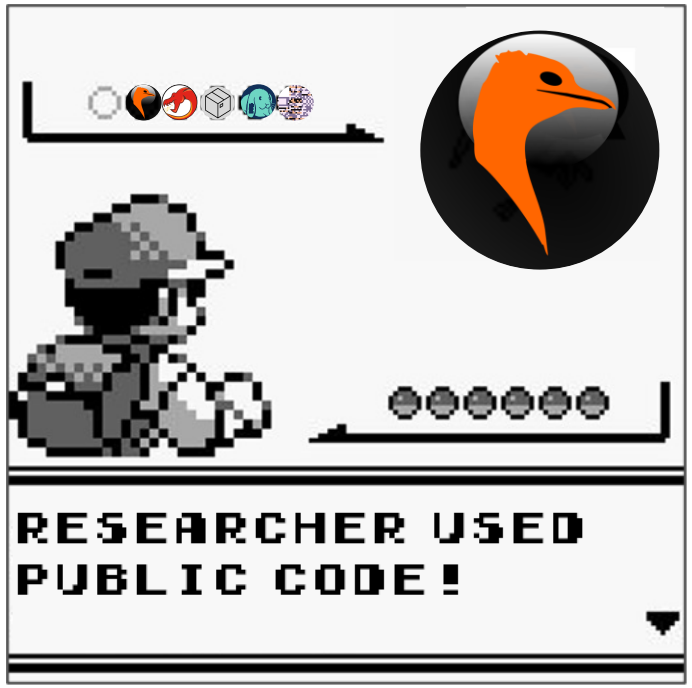
Step	Details	Screenshots
Dynamic Debugging	▪ **Reading the source** and QEMUs **logging infrastructure** is extremely helpful	

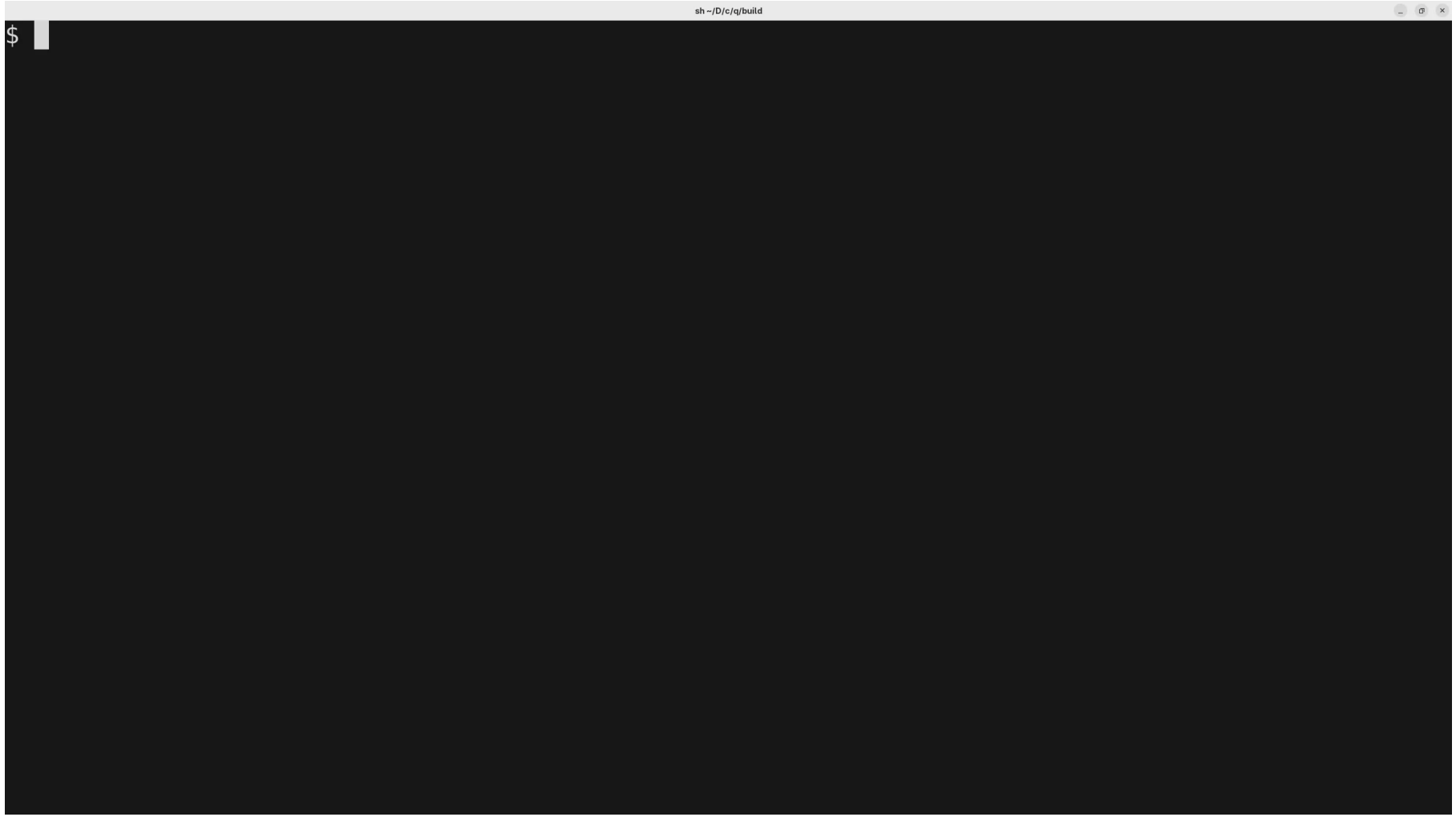
Example problem	▪ **Boot is stuck** without error
	`$ qemu-system-hexagon -display none -kernel qdsp6sw.mbn -d mmu`
	▪ **TLB issue** and its location indicated by QEMU logs
	▪ **TLB index** out of bounds
	▪ **Unexpected behaviour** triggered
		`hexagon_tlb_fill: tid = 0x0, pc = 0xfe002f20,`
		`e = 4, MMU_DATA_LOAD , probe = 0, MMU_KERN`
		`TLB miss RW exception (0x6) caught: Cause code`
		`0xfe002f20, BADVA = 0xfe104a88`
		`hex_tlb_lock: 0`
		TLB logs during emulation

Fix	▪ **Identify max** used TLB size ▪ **Modify** `target/hexagon/hex_mmu.h`	`#define NUM_TLB_ENTRIES 192` *TLB size in QEMU machine definition*

Hacking around the TLB size yields one more line of boot log

Obtain firmware → Get full system Emulation → Patch for our device





After the emulator runs, we deeply inspect the firmware with Ghidra

QEMU:
Emulate baseband on unrestricted hardware

Ghidra:
Integrate Hexagon processors in Ghidra

Clade:
Reverse-engineer compression algorithm

LibAFL:
Coverage guided fuzzing with QEMU

Firmware Harness:
Hook subsystem for analysis & fuzzing

Configure memory map → **Compare disassembly** → **Train Ghidra manually**

Reversing the firmware with Ghidra involves modifying the memory map

Step	Details	Screenshots
Problems with reversing using Ghidra	▪ Use Ghidra with Hexagon plugin ▪ **Misidentified** memory segments ▪ **Requires** manual correction	
Fix: Configure memory map manually	▪ **Memory segments defined** by QEMU machine ▪ **Extract memory segment** details from the firmware (`mtree`) ▪ **Adjust memory map** to set correct start and end addresses for the segments	

```
(qemu) info mtree
address-space: cpu-memory-0
address-space: cpu-memory-1
address-space: cpu-memory-2
address-space: cpu-memory-3
address-space: cpu-memory-4
address-space: cpu-memory-5
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000000000-00000000ffffffff (prio 0, ram): lpddr4.ram
    00000000d8180000-00000000d81801ff (prio 0, rom): config_table.rom
    00000000d81e0000-00000000d81effff (prio 0, i/o): fast
    00000000d8400000-00000000d87fffff (prio 0, ram): vtcm.ram
    00000000fab20000-00000000fab20fff (prio 0, i/o): qutimer
    00000000fc910000-00000000fc910fff (prio 0, i/o): l2vic
    00000000fc921000-00000000fc922fff (prio 0, i/o): qutimer_views
    0000000910000000-000000091fffffff (prio 0, ram): cpz.ram
```

Memory segments information using mtree

config_table.rom	d8180000	d81801ff
fast	d81e0000	d81e0fff
vctm.ram	d8400000	d87fffff
qutimer	fab20000	fab20fff
l2vic	fc910000	fc910fff
qutimer_views	fc921000	fc922fff
segment_4	fe000000	fe003fff
segment_31	fe200000	ff7e1fff
_elfHeader	_elfHeader::0...	_elfHeader::0...
_elfProgramHeaders	_elfProgramHe...	_elfProgramHe...
cpz.ram	cpz.ram::9100...	cpz.ram::91ff...
unallocated_0	unallocated_0...	unallocated_0...
lpddr4.ram	lpddr4.ram::0...	lpddr4.ram::f...

Memory mapping in Ghidra

Comparing the disassembled code with Hexagon tools output refines the reversing efforts

Step	Details	Screenshots
Problem	▪ **Gap:** Ghidra misses registers, memory addresses and VLIW boundaries ▪ **Poor readability:** Opcodes are less readable compared to `objdump` output	

```
                        LAB_c0312040                                    XREF[1]:
c0312040 04 60 00 7c  {  A2_combineii      R5R4 0x0 0x1
c0312044 01 40 00 78     A2_tfrsi          R1 0x0
c0312048 81 41 01 3c     S4_storeirb_io    R1 0x3 0x1
c031204c 02 c8 01 3c     S4_storeirb_io    R1 DAT_00000010 0x2
c0312050 06 40 c2 91  {  L2_loadrd_io      R7R6 R2 0x0
```

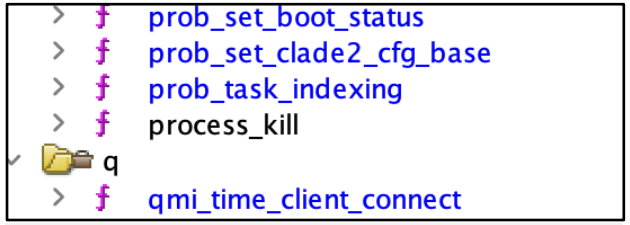
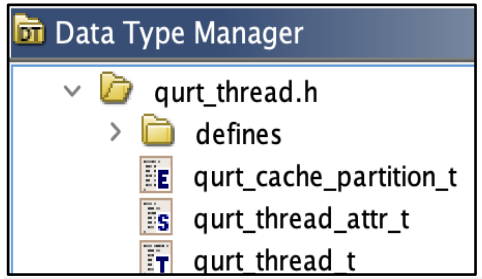
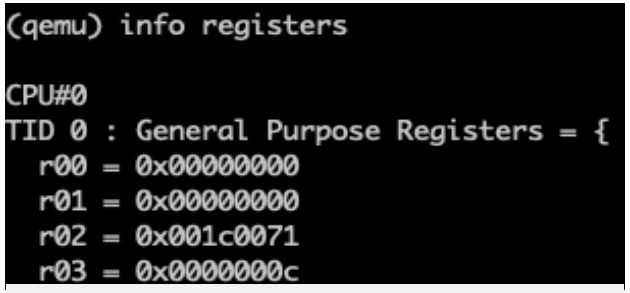
Disassembled in Ghidra with hexagon-plugin

Fix	▪ **Validate:** Use Hexagon SDK's tool `hexagon-llvm-objdump` to validate disassembled code in Ghidra ▪ **Update** the register names in Ghidra, if needed	

```
c0312040:       04 60 00 7c     7c006004 {      r5:4 = combine(#0,#1)
c0312044:       01 40 00 78     78004001        r1 = #0
c0312048:       81 41 01 3c     3c014181        memb(r1+#3) = #1
c031204c:       02 c8 01 3c     3c01c802        memb(r1+#16) = #2 }
c0312050:       06 40 c2 91     91c24006 {      r7:6 = memd(r2+#0)
```

Disassembled with hexagon-llvm-objdump (official Hexagon-SDK tool)

Renaming structs and functions in Ghidra takes time but enhances code comprehension

Step	Details	Screenshots
Refine function naming	▪ **Manually rename functions** based on our knowledge and the print statements in the code	 *Renamed functions in Ghidra*
Define data-structs	▪ **Configure data types and structs** based on open-source code and header definitions	 *Configured structs in Ghidra*
Analyze register states	▪ **Analyze runtime register values** using QEMU to compare with the code logic	 *Runtime memory analysis*

Bonus: Mapping boot flow with the decompiled code calls for automation

Step	Details
Capture the boot trace	▪ **Capture full trace** with QEMU monitor and save to a file `./qemu-system-hexagon -kernel qdsp6sw.mbn -monitor stdio -d exec 2>trace.txt`
Create a Ghidra script	▪ **Parse** the trace using a Python script ▪ **Highlight:** Color each memory address reached during the boot process
Optimize the script	▪ **Scale:** Update script to support multiple threads ▪ Different colors for different threads

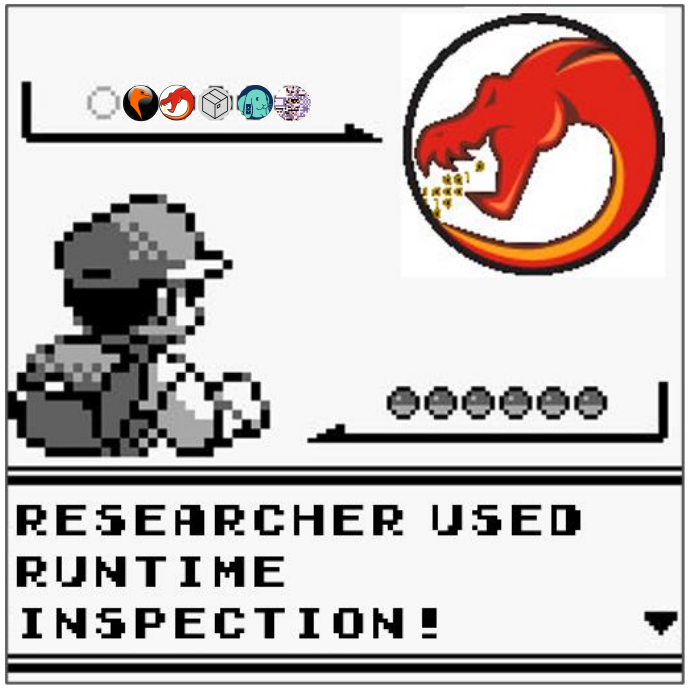
Screenshots

```
fe102b3c 00 c0 41 a0  {  Y2_dccleaninva          R1
fe102b40 00 c0 81 91  {  L2_loadri_io            param_1 R1=>DAT_fe100044 0x0
fe102b44 00 c0 9f 52  {  J2_jumpr                R31

                         **************************************************************
                         *                          FUNCTION                          *
                         **************************************************************
                         uint __stdcall FUN_fe102b48(uint param_1)
                            assume endloop = 0x0
                            assume immext = 0xffffffff
                            assume pkt_next = 0xfe102b4c
                            assume pkt_start = 0xfe102b48
     uint                   R0:4          <RETURN>
     uint                   R0:4          param_1
                         FUN_fe102b48                                  XREF[2]:     FUN_
                                                                                    FUN_

fe102b48 02 c0 86 6e  {  Y2_tfrscrr              R2 S6_SSR
fe102b4c 00 40 7f 00  {  A4_ext                  0x7f00000
fe102b50 00 cc 02 de     S4_andi_asl_ri          R2 0x7f00000 0xc
fe102b54 a2 cc 40 8e  {  S2_lsr_i_r_or           R2 param_1 0xc
fe102b58 01 c0 82 6c  {  Y2_tlbp                 R1 R2
fe102b5c 00 5f 01 85  {  S2_tstbit_i             P0 R1 0x1f
fe102b60 00 c8 5f 53     J2_jumprtnew            P0 R31
fe102b64 00 c8 01 8d  {  S2_extractu             param_1 R1 0x8 0x0
fe102b68 00 c0 9f 52  {  J2_jumpr                R31
fe102b6c e0 ff df 78  {  A2_tfrsi                R0 -0x1
fe102b70 00 c0 9f 52  {  J2_jumpr                R31
```

Colored opcodes in Ghidra

We can now follow along emulator execution in our Ghidra setup

Configure memory map → Compare disassembly → Train Ghidra manually




```
fe10c080 4a 75 fe 5b  {  J2_call          prob_set_boot_status
fe10c084 00 c3 00 78     A2_tfrsi         uVar4 0x18
fe10c088 14 59 00 5a  {  J2_call          qurt_printf
fe10c08c 95 46 e1 0f     A4_ext           s_0x%x,_BADVA:_0x%x,_SSR:_0x%x,_SP_fe11a53b+5
fe10c090 c0 c7 00 78     A2_tfrsi         uVar4=>s_QURT_kernel_started_fe11a57e s_QURT_k...
fe10c094 40 75 fe 5b  {  J2_call          prob_set_boot_status
fe10c098 20 c3 00 78     A2_tfrsi         uVar4 0x19
fe10c09c 0a 59 00 5a  {  J2_call          qurt_printf
fe10c0a0 96 46 e1 0f     A4_ext           s_RT_kernel_started_fe11a57e+2
fe10c0a4 60 c2 00 78     A2_tfrsi         iVar8=>s_QURT_kernel_init_cache_params_fe11a59...
fe10c0a8 b4 f4 00 5a  {  J2_call          FUN_fe112a10
fe10c0ac ba e4 01 5a  {  J2_call          FUN_fe118a20
fe10c0b0 0c 42 e1 0f  {  A4_ext           DAT_fe108300
fe10c0b4 02 c0 00 78     A2_tfrsi         uVar12=>DAT_fe108300 DAT_fe108300
fe10c0b8 5c 6b 00 5a  {  J2_call          some_struct_contruction_subtree
fe10c0bc 01 c0 42 3c     S4_storeiri_io   uVar12=>DAT_fe108300 0x0 0x1
fe10c0c0 2a 75 fe 5b  {  J2_call          prob_set_boot_status
fe10c0c4 40 c3 00 78     A2_tfrsi         iVar8 0x1a
fe10c0c8 f4 58 00 5a  {  J2_call          qurt_printf
fe10c0cc 96 46 e1 0f     A4_ext           s_RT_kernel_started_fe11a57e+2
fe10c0d0 40 c6 00 78     A2_tfrsi         iVar8=>s_QURT_root_task_started_fe11a5b2 s_QUR...
fe10c0d4 a6 ee ff 5b  {  J2_call          clade_related
fe10c0d8 4c f1 ff 5b  {  J2_call          clade_related2
```

Colored opcodes in Ghidra reached during the boot flow

```
prob_set_boot_status(0x18);
qurt_printf(s_QURT_kernel_started_fe11a57e);
prob_set_boot_status(0x19);
iVar8 = qurt_printf(s_QURT_kernel_init_cache_params_fe11a593);
FUN_fe112a10(iVar8,param_2,iVar7,ppuVar13,param_5,param_6);
uVar6 = FUN_fe118a20();
uVar4 = 0;
DAT_fe108300 = 1;
some_struct_contruction_subtree
          (uVar6,param_2,-0x1ef7d00,ppuVar13,param_5,param_6,unaff_R16,unaff_R17,unaff_R30,
           (char)in_R31,param_7);
uVar11 = (undefined)param_2;
prob_set_boot_status(0x1a);
iVar7 = qurt_printf(s_QURT_root_task_started_fe11a5b2);
uVar5 = clade_related((char)iVar7,uVar11,uVar4,(char)ppuVar13,(char)param_5,(char)param_6,
                      CONCAT44(unaff_R17,unaff_R16),CONCAT44(in_R31,unaff_R30));
clade_related2(uVar5,uVar11,uVar4,(char)ppuVar13,(char)param_5,(char)param_6,unaff_R16,unaff_R17
               ,unaff_R30,(char)in_R31,param_7,param_8);
```

Matching decompiled code

We can observe different compression mechanisms in the firmware, our next stage to tackle

QEMU:
Emulate baseband on unrestricted hardware

Ghidra:
Integrate Hexagon processors in Ghidra

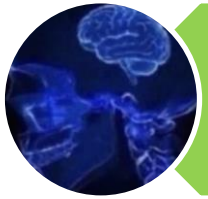
Clade:
Reverse-engineer compression algorithm

Compressed segments → **Decompression** → **Skip decompression**

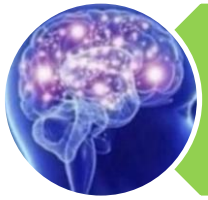
LibAFL:
Coverage guided fuzzing with QEMU

Firmware Harness:
Hook subsystem for analysis & fuzzing

Various proprietary compression algorithms slow the pace of reverse engineering efforts



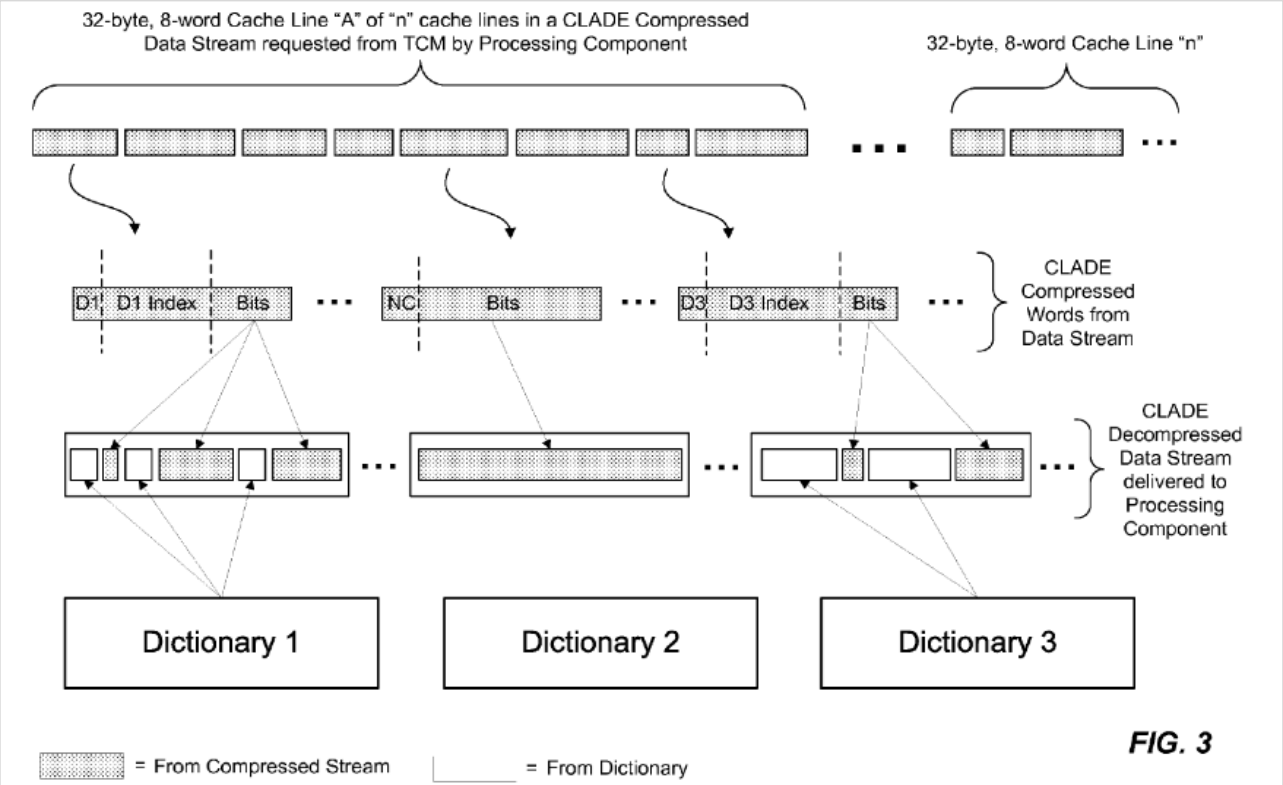
Deltacompress: Used to compress data in Qualcomm firmware; tools available for decompression



Q6zip: Used to compress code in Qualcomm firmware; tools available for decompression

CLADE: Replaces Q6zip; requires CLADE dict and config for decompression

CLADE2: Enhanced version of clade; utilizes hardware for decompression

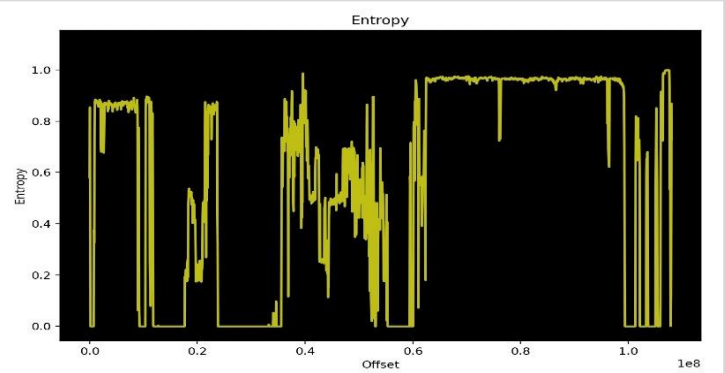
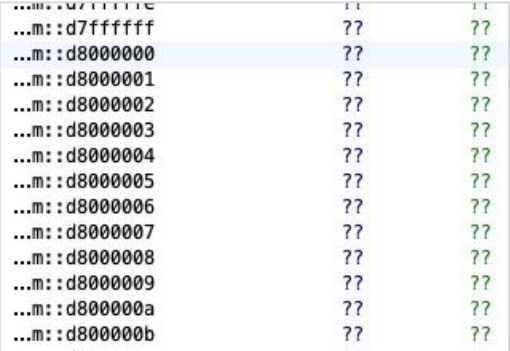
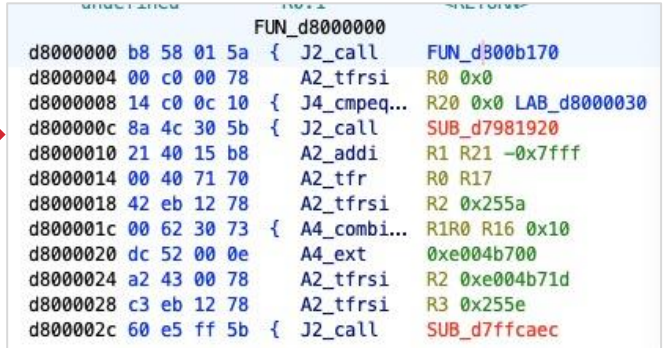


CLADE compression algorithm

https://patentimages.storage.googleapis.com/5a/0e/7b/68e1c19996e217/US9300320.pdf

The manual decompression of clade section achieves partial progress, but full resolution requires additional analysis

Compressed segments → **Decompression** → Skip decompression

Step	Details	Screenshots
Decompress firmware sections	▪ **Binwalk and binutils identify** compressed sections ▪ **Public tooling decompresses** sections successfully	 *Binwalk output of iPhone baseband*
Import sections with Ghidra	▪ **Add decompressed sections** with Ghidra Memory Manager	 *Before decompression*  *After decompression*


```
...m::d7ffffff   ??   ??
...m::d8000000   ??   ??
...m::d8000001   ??   ??
...m::d8000002   ??   ??
...m::d8000003   ??   ??
...m::d8000004   ??   ??
...m::d8000005   ??   ??
...m::d8000006   ??   ??
...m::d8000007   ??   ??
...m::d8000008   ??   ??
...m::d8000009   ??   ??
...m::d800000a   ??   ??
...m::d800000b   ??   ??
```

```
                    FUN_d8000000
d8000000 b8 58 01 5a  {  J2_call     FUN_d800b170
d8000004 00 c0 00 78     A2_tfrsi    R0 0x0
d8000008 14 c0 0c 10  {  J4_cmpeq...  R20 0x0 LAB_d8000030
d800000c 8a 4c 30 5b  {  J2_call     SUB_d7981920
d8000010 21 40 15 b8     A2_addi     R1 R21 -0x7fff
d8000014 00 40 71 70     A2_tfr      R0 R17
d8000018 42 eb 12 78     A2_tfrsi    R2 0x255a
d800001c 00 62 30 73  {  A4_combi...  R1R0 R16 0x10
d8000020 dc 52 00 0e     A4_ext      0xe004b700
d8000024 a2 43 00 78     A2_tfrsi    R2 0xe004b71d
d8000028 c3 eb 12 78     A2_tfrsi    R3 0x255e
d800002c 60 e5 ff 5b  {  J2_call     SUB_d7ffcaec
```

Using hexagon-lldb, enables dynamic analysis on the firmware

Step	Details	Screenshots
Manual flow correction	Start emulation ↓ Emulate until eternal loop/crash ↓ Reverse engineer failed codeflow ↓ Set breakpoint in LLDB and hotfix codeflow ↓ Continue booting ↓ In user mode ? — No → Emulate until eternal loop/crash; Yes ↓ Done: User mode reached	

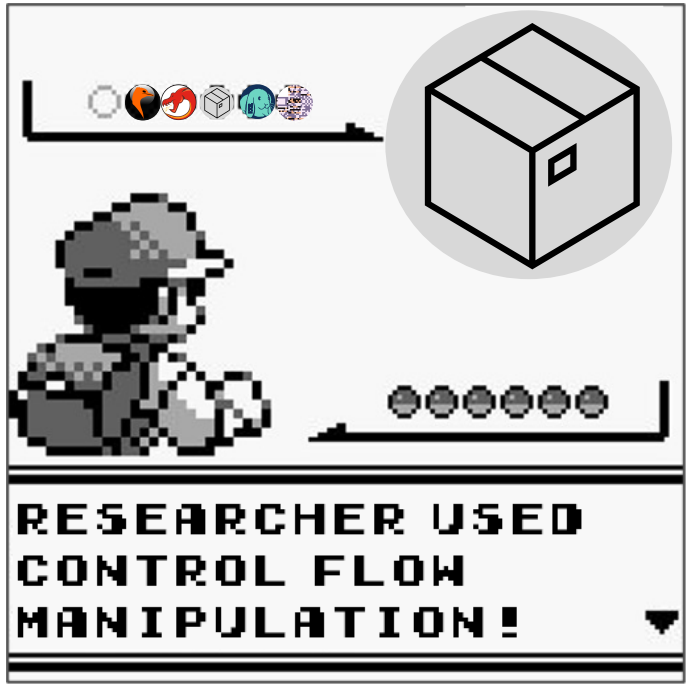
```
init static mapping: ppn 0x00005780, vpn 0x00005780, pages 0x00000040
init static mapping: ppn 0x00004900, vpn 0x00004900, pages 0x00000010
finished static mem
App Images Init
```

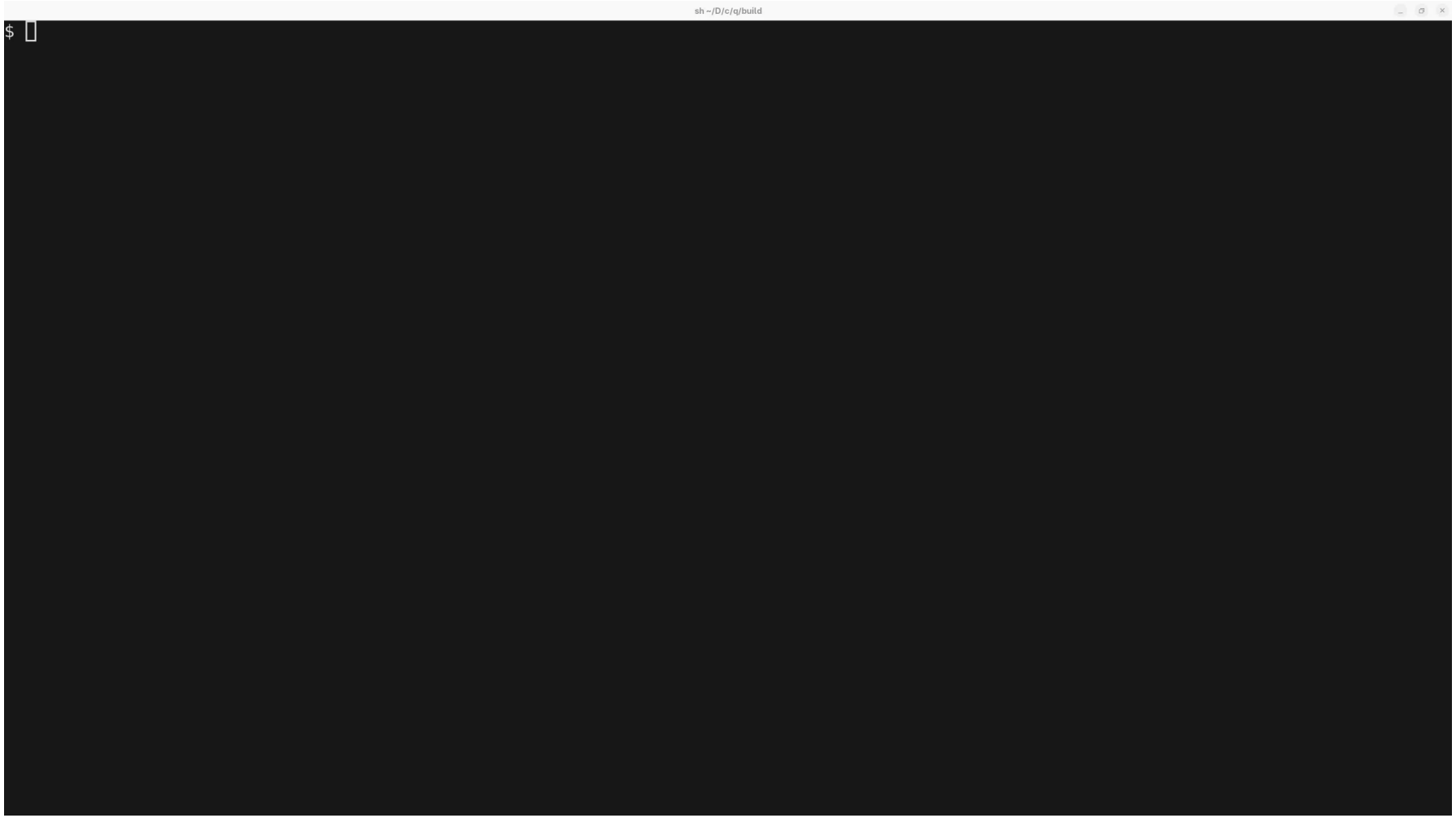
Finished initializing memory areas

```
(lldb) sysstatus
modectl:  0x003d003f
bestwait: 0x1ff / 511 (dec)
schedcfg: 0x0000010f - int #0f / 15 (dec), EN:enabled
syscfg:   0x0095807f
TID Prio Mode  Priv    Cause    Set            Unset
--- ---- ----- ------- ----- -------------  -------------
  0  255 WAIT  Monitor  0x43 EX,GM,IE        SS,UM,XE
  1    1 RUN   Monitor  0x1d EX,GM,IE,UM     SS,XE
  2  255 WAIT  Monitor  0x43 EX,GM,IE        SS,UM,XE
  3  255 WAIT  Monitor  0x43 EX,GM,IE        SS,UM,XE
  4  255 WAIT  Monitor  0x43 EX,GM,IE        SS,UM,XE
```

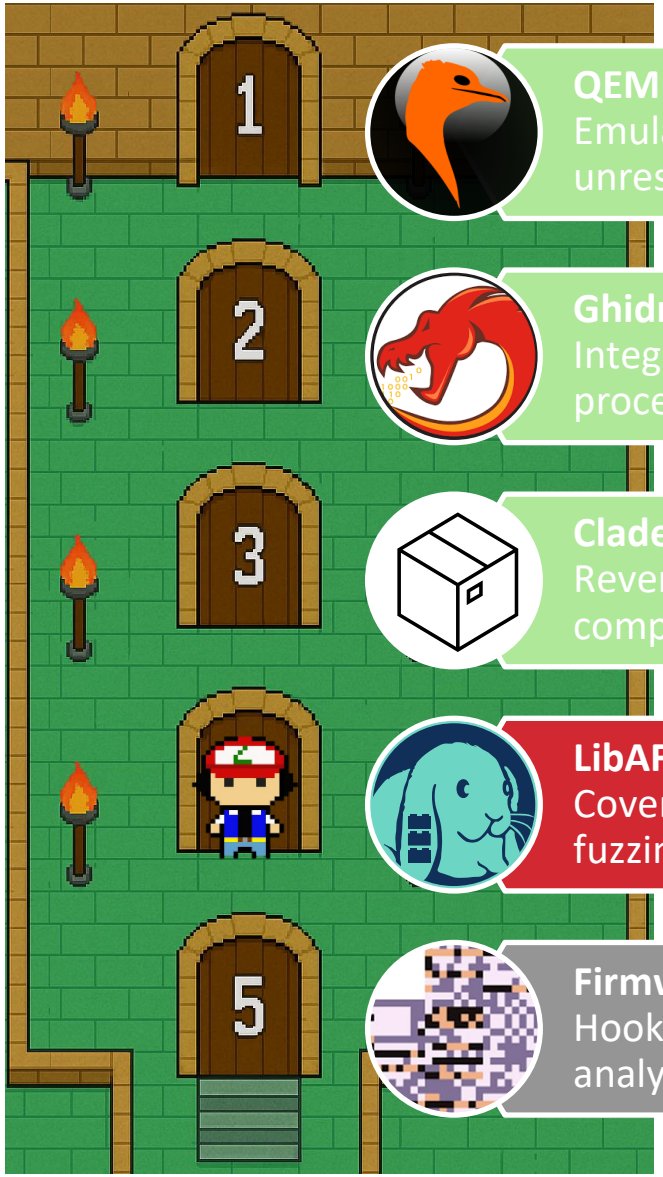
Custom LLDB status commands for Hexagon

Connecting LLDB to QEMU, we can solve code flow errors and skip parts where needed





After controlling the flow manually, we want full control and fuzzing with LibAFL



QEMU:
Emulate baseband on unrestricted hardware

Ghidra:
Integrate Hexagon processors in Ghidra

Clade:
Reverse-engineer compression algorithm

LibAFL:
Coverage guided fuzzing with QEMU

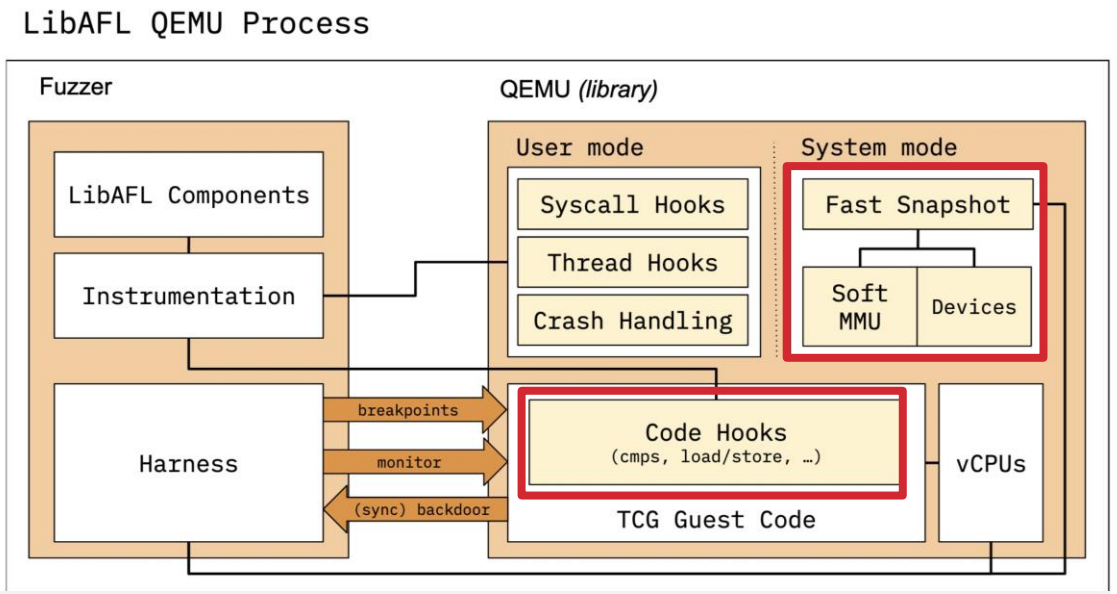
Merge codebase → **Instrumentation** → **Harness**

Firmware Harness:
Hook subsystem for analysis & fuzzing

LibAFL's QEMU provides exactly what we need but merging it with Hexagon QEMU is hard

Step	Details
libafl-qemu-bridge	▪ **Hooks** for control flow manipulation ▪ **Snapshots** of the process ▪ **User mode emulation** implemented for hexagon ❗ **No common ancestor** with Hexagon QEMU
Merge branches manually	▪ **Diff bridge** against QEMU base version ▪ **Apply diff** on Hexagon QEMU ▪ **Fix** compile errors
Feature integration	▪ **Patch** in support for Hexagon system mode support

Screenshots



LibAFL QEMU architecture diagram

LibAFL provides hooks, which we use to instrument the baseband boot procedure

Merge codebase → **Instrumentation** → Harness

Step	Details	Screenshots
Define breakpoints	▪ **Rust function hook** in the fuzzer to handle breakpoints ▪ **Introspection** in common functions like printf ▪ **JSON configuration** for breakpoints	
Compare with disassembled code	▪ **Colored trace analysis** guides flow manipulation	
Control flow manipulation	▪ **Boot optimizations** by skipping memory zeroing and device initialization functions ▪ **Boot progression**	

```
"breakpoints": [
  {
    "name": "qurt_println",
    "address": "0xfe10f2b0",
    "handler": "HandlePrintln"
  },
```

JSON config

```
QURT kernel started
QURT kernel init cache params
QURT root task started
DLPager_rw_swap_vaddr a0000000, DLPager_rw_swap_size 50000
QuRTOS heap init: start cb3bc000 size 0x80000
QuRTOS ISLAND heap init: start bfea0200 size 0x2800
Init memory pools...
Found Virtual Pool in overrides. size 124, pool id 4096
Adding pages 0x00001000->0x000bfe00 to Default Virtual pool.
0 name DEFAULT_PHYSPOOL pool cb3c4490 island 0, collapse_type 0
Found requested Pool ID in overrides. size 88, pool id 0
Using over-ride ranges for pool index 0; number of ranges 10
        0: 0x900dc000-->0x90100000
1 name HWIO_POOL pool cb3c4570 island 0, collapse_type 0
Using configured ranges for pool index 1; number of ranges 1
        0: 0x09000000-->0x0a000000
11 name CLADE_DICT pool cb3c4e30 island 0, collapse_type 0
Found requested Pool ID in overrides. size 8, pool id 11
```

Boot logs after control flow manipulation

Creating a harness requires taming a 108MB proprietary codebase

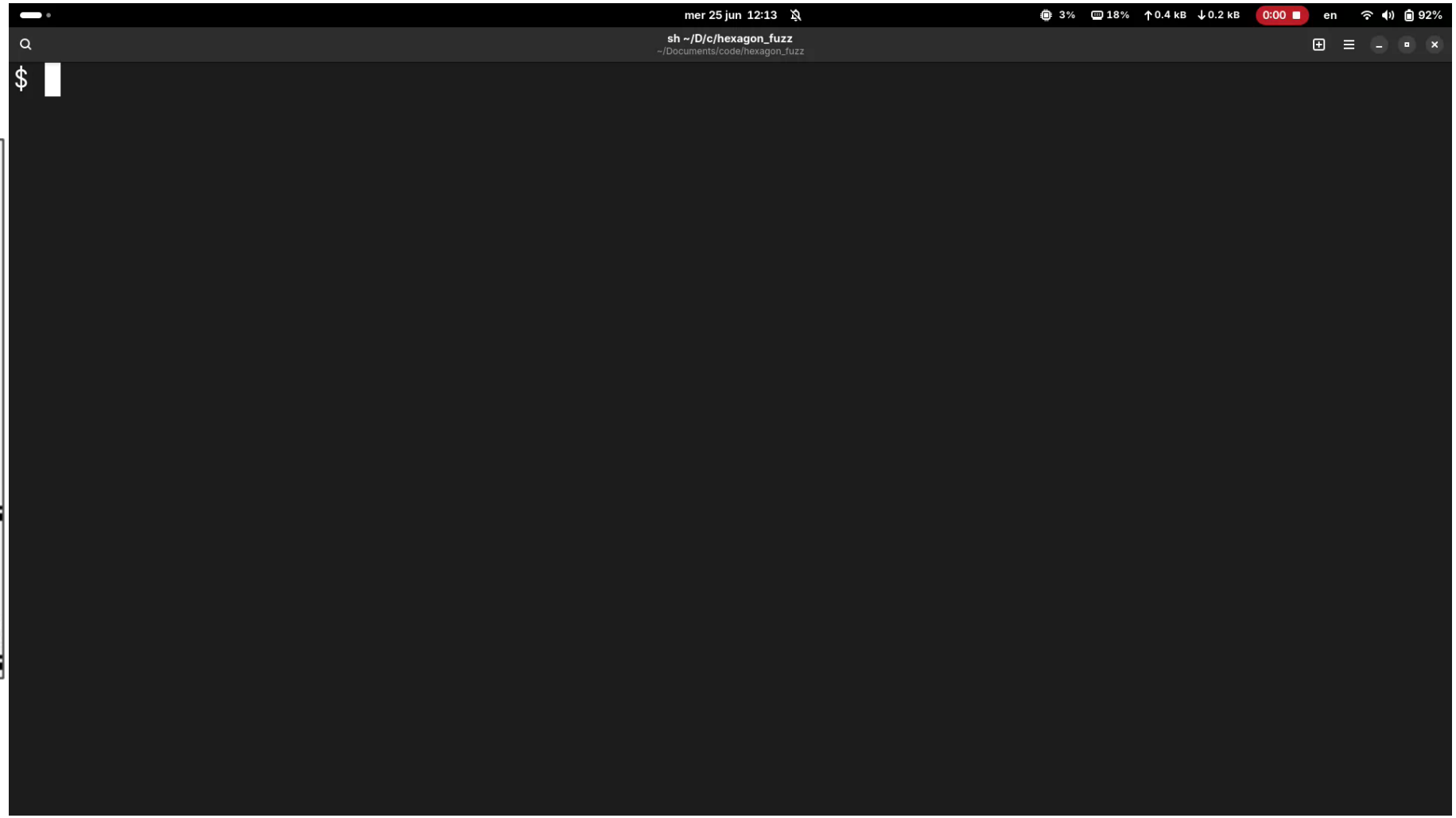
Merge codebase → Instrumentation → Harness

Step	Details	Screenshots
Codebase navigation	▪ **70k strings, 30k functions** most without references ▪ **Cross reference** with source code from SDK and internet ▪ **Utilize Ghidra scripting** to recover function names ▪ **Compare with Qualcomm tooling** e.g. QXDM monitoring	

Thread Name	Thread ID	Proirity	PID	MCPS	Delta CPU %	Total CPU %	Stack Size (bytes)
ml1_mgr	148	98	0	0.9	0.16	0.24	9216
RFLM_QLNK_OFD1	106	51	0	0.79	0.18	0.21	4088
LFW_SCHD_CMN_1	3052	27	0	0.51	0.09	0.13	8112
VSTMR irq40	20d2	77	0	0.48	0.12	0.13	8192
RFLM_CCS	f7	33	0	0.4	0.1	0.1	4016
rf_fe	176	99	0	0.36	0.11	0.09	32768
gsm_msgr_t1	142	99	0	0.29	0.07	0.08	4608
slpc_worker	17d	131	0	0.32	0.07	0.08	8192
Prof TP	25d04a	124	0	0.25	0.07	0.07	4096
GFW_GENERIC_1	106e	40	0	0.23	0.15	0.06	8192
GFWRF_TRIG2_A0	1070	27	0	0.23	0.1	0.06	4096
GFWRF_TRIG1_A0	1071	27	0	0.23	0.06	0.06	4096
DPC_Task	1000	126	0	0.47	0.05	0.06	4096
RFLM_QLNK	107	33	0	0.18	0.05	0.05	4088
gsm_rr	120	164	0	0.18	0.04	0.05	384

Running processes in QXDM

With integration of LibAFL we successfully boot the Hexagon baseband and are ready to fuzz



With this setup we are ready to explore the unknown

QEMU:
Emulate baseband on unrestricted hardware

Ghidra:
Integrate Hexagon processors in Ghidra

Clade:
Reverse-engineer compression algorithm

LibAFL:
Coverage guided fuzzing with QEMU

Firmware Harness:
Hook subsystem for analysis & fuzzing

Agenda

Introduction: Hexagon baseband

From research to tooling

Demo: Fuzzing Hexagon

Opening up baseband security

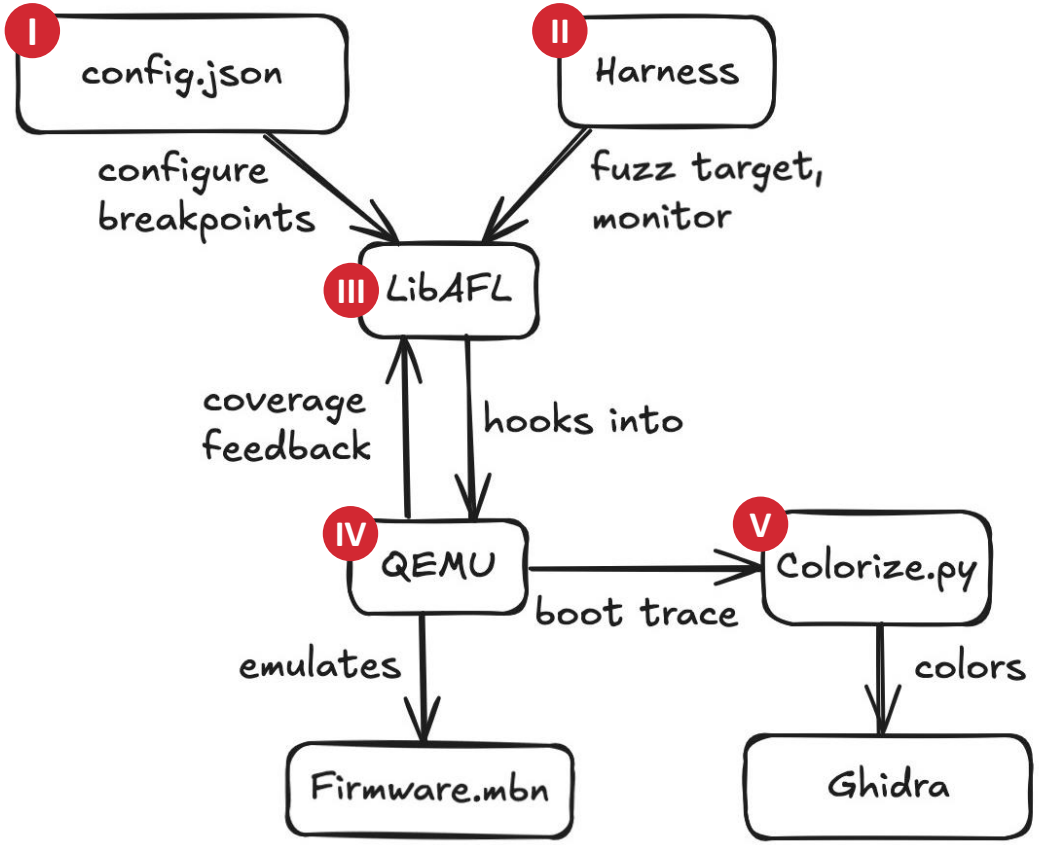
Fuzzing the Hexagon baseband: we share our tooling

Architecture Components

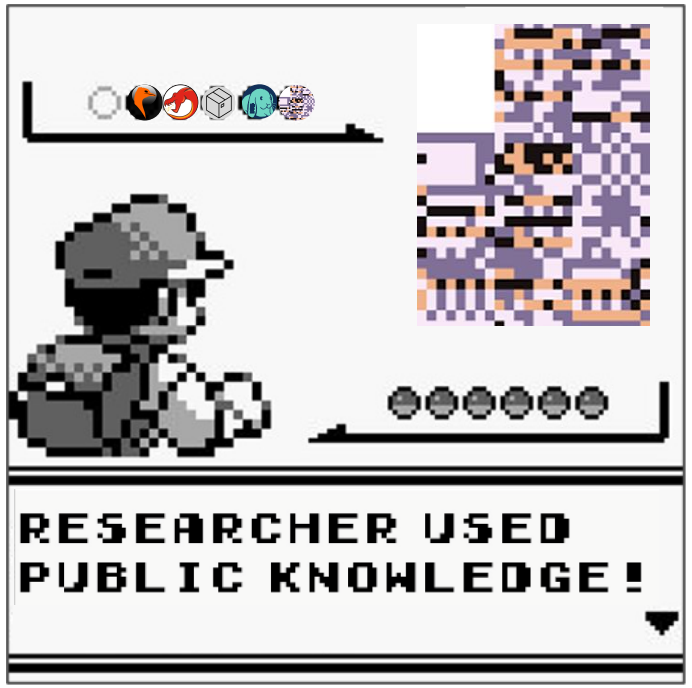
Our Hexagon baseband fuzzer consists of following components:

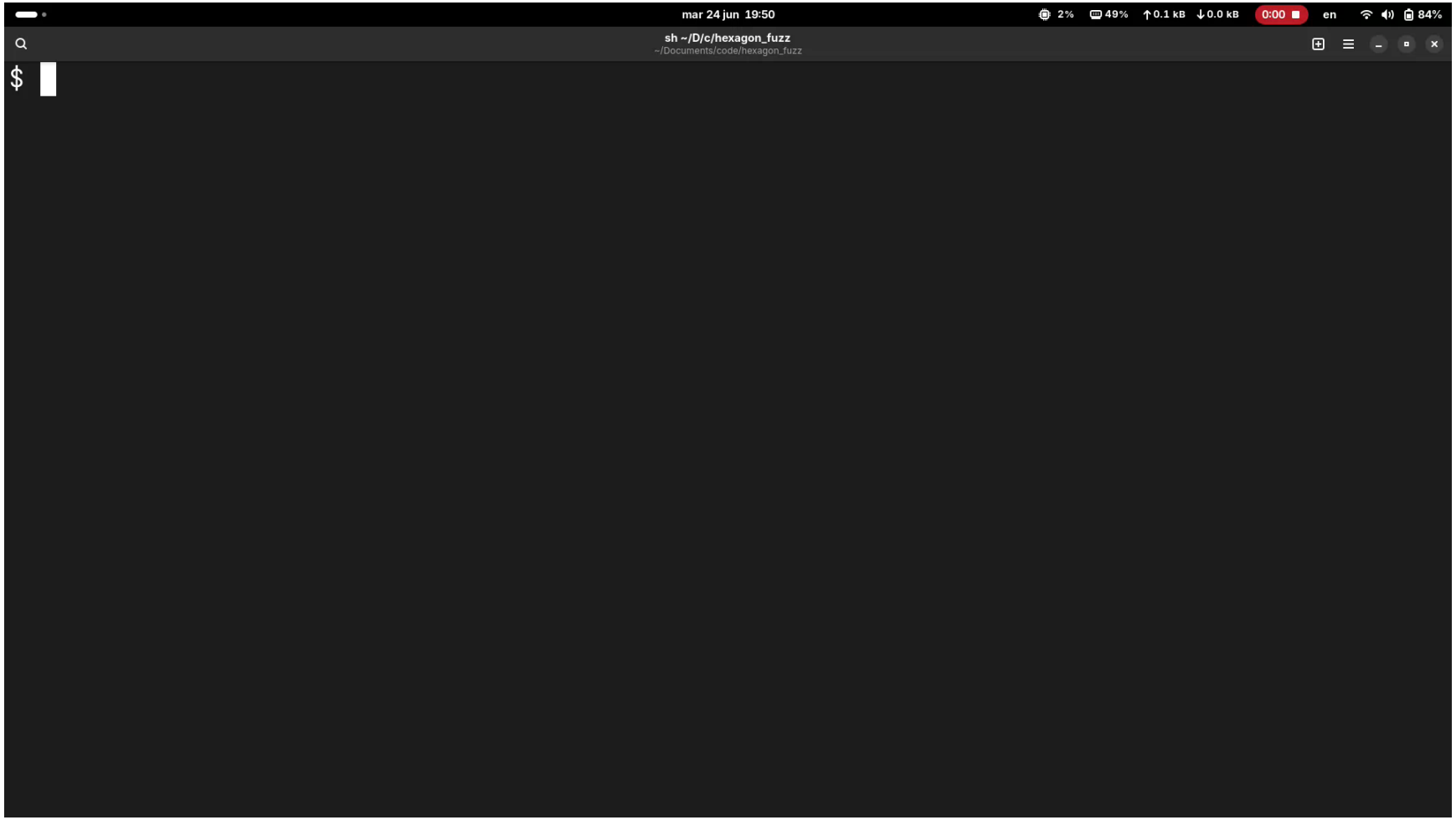
- I **Config.** Set fuzzing parameters and breakpoint addresses
- II **Harness.** Rust harness to interact with LibAFL to set breakpoints, fuzz target, monitor process and process coverage
- III **LibAFL.** Hook into firmware and introspect
- IV **QEMU.** Emulate the Hexagon firmware and get coverage feedback
- V **Ghidra scripts.** Improve understanding of decompiled code and progress boot

Architecture Diagram



Demo


RESEARCHER USED
PUBLIC KNOWLEDGE!


mar 24 jun 19:50
2%
49%
0.1 kB
0.0 kB
0:00
en
84%
sh ~/D/c/hexagon_fuzz
~/Documents/code/hexagon_fuzz
$

Agenda

Introduction: Hexagon baseband

From research to tooling

Demo: Fuzzing Hexagon

Opening up baseband security

We refine and integrate our tool to enable successful vulnerability research

Goal	Approach
Targeted fuzzing	**1. Map critical functions** **2. Map exposure:** Identify functions that can be triggered from phone/SIM **3. Optimize fuzzing harness** to dynamically adapt to the target function
Optimize CLADE2 decompression	**1. Decompress CLADE2 segments** to enable proper task initialization **2. Ghidra integration:** Integrate decompressed output with Ghidra to refine decompiled code
On-device verification	**Dynamic testing on iPhone:** Trigger crashes, eg. using an SDR
Integrate with existing tools	**Integrate Hexagon fuzzer** with FirmWire

GitHub

srlabs/hexagon_fuzz

Thank you

- Androm3da
- CUB3D
- domenukk
- Janne
- nlitsme
- nsr
- Mzakocs
- toshipiazza
- …