



# Attacking Hexagon: Security Analysis of Qualcomm's aDSP

Dimitrios Tatsis (@dtouch3d)

tatsisd@census-labs.com

Recon Montreal 2019

# ▷ \$ whoami

- Security Researcher at CENSUS S.A.
- Reverse Engineering, Exploitation, Code Audit
- I fight Androids

# ▷ Agenda

- Introduction to Hexagon and aDSP
- System Architecture
- FastRPC Framework
- Custom code on aDSP
- Attack Surface
- Fuzzing
- Conclusions

## ▷ aDSP and Hexagon

# ▷ Qualcomm aDSP

- Low power, high performance DSP coprocessor
- Exists in all modern Qualcomm SoCs
- Hexagon Architecture
  - Same as Qualcomm baseband

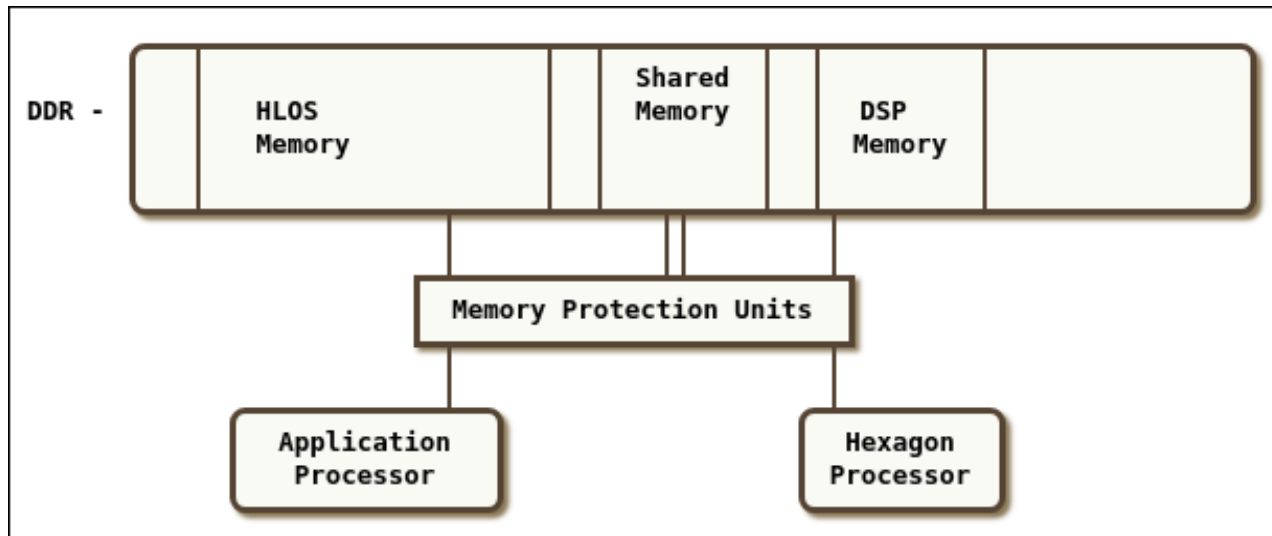
# ▷ Qualcomm aDSP

- Runs its own OS, QuRT
  - Runs Hexagon ELF files
  - Again same as Qualcomm baseband
- Provides shared objects that can be called from Android userspace in an RPC manner
- Machine Learning, Computer Vision, Audio Decoding

# ▷ Qualcomm aDSP

- Qualcomm Shared Memory Subsystem
  - Application Processor -> aDSP communication
  - Also used for other subsystems like baseband and Wi-Fi
- aDSP needs access to main system memory
  - Argument Passing
  - Results

# ▶ Qualcomm aDSP - Memory



HLOS = High Level OS (Android, Windows)

\* As shown in the Qualcomm SDK Documentation



# ▷ Qualcomm aDSP - Memory

- Memory Protection Unit
  - Makes sure aDSP can access only specific memory
- Internal aDSP MMU
  - QuRT provides page tables for address translation from virtual to physical
- Limited TLB Entries
  - Large Contiguous Buffers are preferred

# ▷ Qualcomm aDSP - Memory

- Memory Carveout
  - Android ION Allocator - Contiguous
  - Specific ION Heap
  - ION buffers can be mapped to aDSP
- SMMU
  - System Memory Management Unit
  - Analogous to IOMMU in x86
  - Buffers only appear to be contiguous

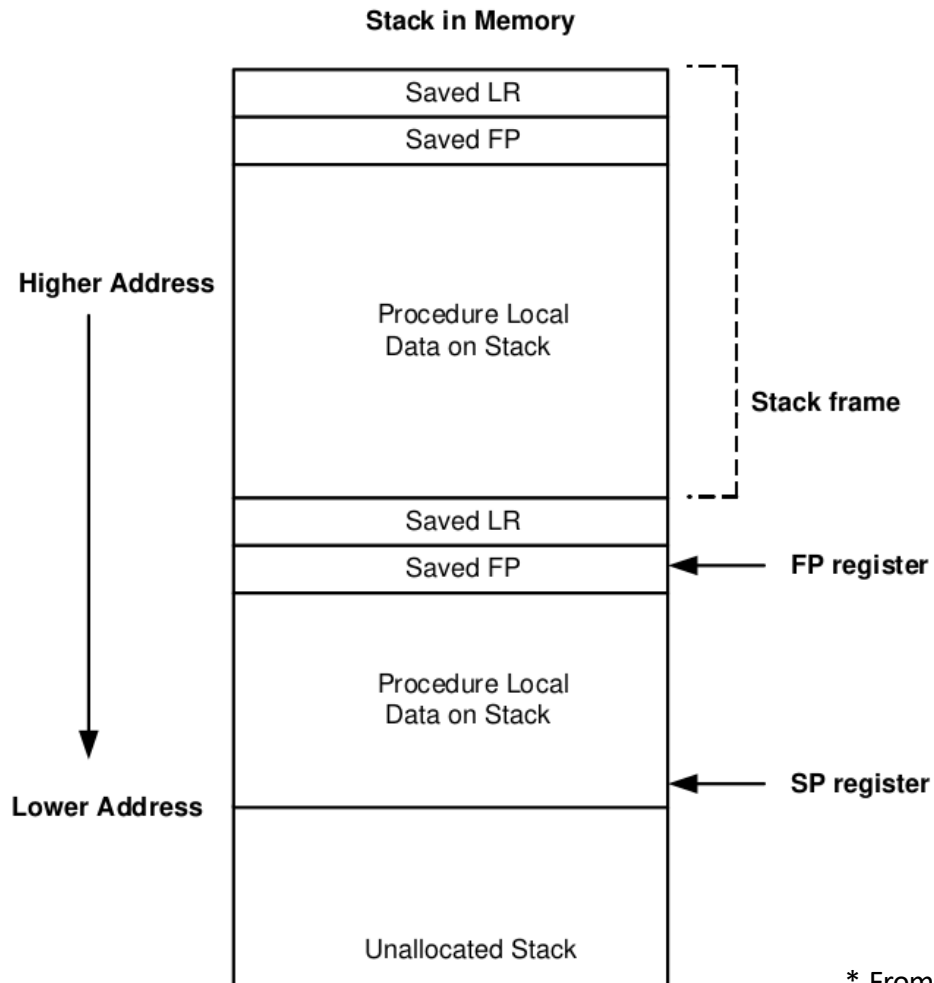
# ▷ Hexagon Architecture

- Specifically designed for DSP use cases
- VLIW 32-bit Instruction Set
- Little-endian
- Instruction Packets, compound instructions
- 4 execution units

# ▷ Hexagon Architecture

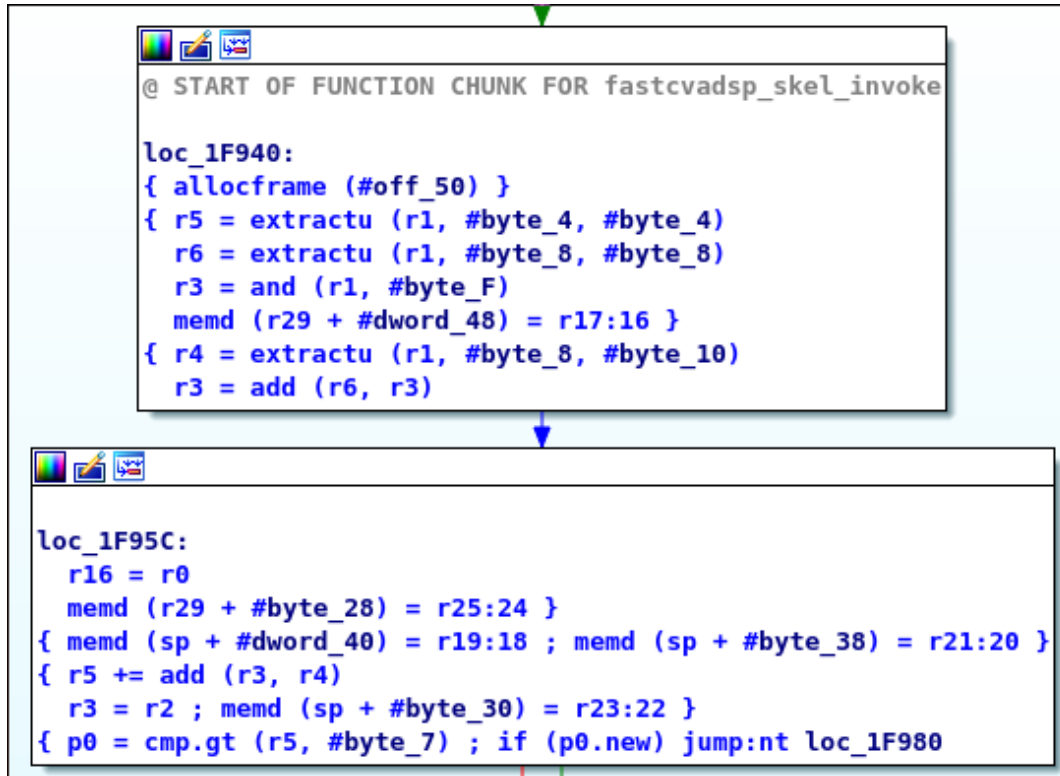
- Registers R0 – R31
- Stack Pointer, Frame Pointer, Link Register
- Special hardware synchronization primitives
- Not your typical assembly language

# ▷ Hexagon Architecture



\* From Hexagon V62 Programmers Manual

# ▷ Hexagon Architecture



- Instruction packets are denoted in { ... }
  - Instructions are executed in parallel

# ▶ Hexagon Hardware Security Mitigations

- Only on Hexagon V61 and greater
- FRAMELIMIT Register
  - In frame allocation, if  $SP < FRAMELIMIT$  throw exception
- FRAMEKEY Register
  - Return address XOR FRAMEKEY
  - Different for every hardware thread
  - Changes "regularly" as per documentation but no other information provided

# ▷ QuRT

- Qualcomm Real Time OS
- Runs on aDSP and baseband
- Privilege modes:
  - QuRT OS
  - Guest OS (root)
  - User
- Scheduling, resource management, address translation



# ▷ QuRT Mitigations

- No ASLR
- Stack cookies
- W<sup>X</sup>
  - Can't write to executable memory
  - Can't execute data memory
- Heap corruption protection

# ▷ QuRT

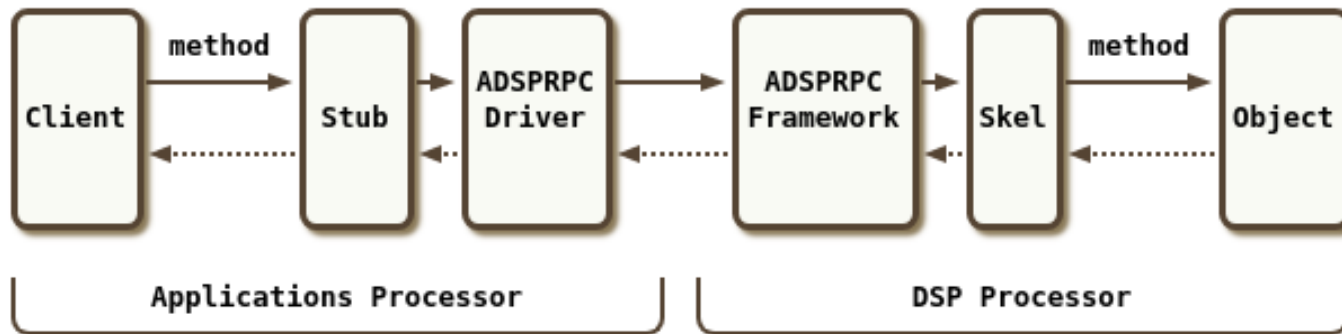
- Binary can be found in TrustZone applets folder
  - /firmware/image/
- Files: adsp.mdt, adsp.b[0-9]
- Can be reassembled by [https://github.com/laginimaine/unify\\_trustlet](https://github.com/laginimaine/unify_trustlet)

## ▶ FastRPC Framework

# ▷ FastRPC

- Communication between APPS processor and aDSP
- Qualcomm Shared Memory Subsystem
- Intermediate Libraries
  - On the Android userpace - Stub
  - On the aDSP - Skel
- Kernel Driver

# ▷ FastRPC

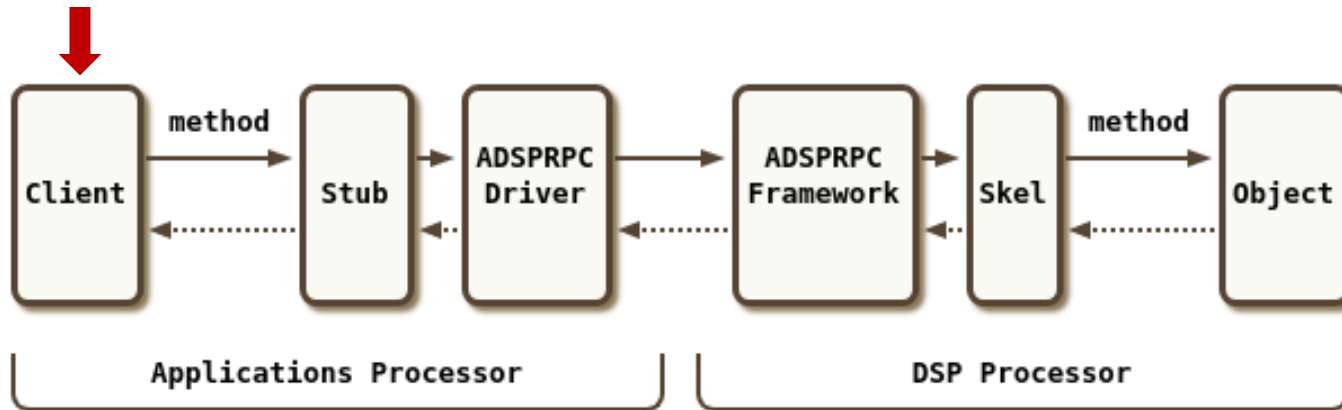


- The diagram shows the "simplified version"!

\* From Hexagon DSK Documentation

# ▷ FastRPC

You are here



- Say we want to use aDSP from an Android App
  - Windows on Arm would be pretty much the same
- Which libraries and functions can we call ?

# ▷ FastRPC – Remote Filesystem

- /vendor/lib/rfsa/adsp
- Holds all libraries accessible for RPC
- Available libraries vary between vendors

## ▷ FastRPC – Available Libraries

```
-rw-r--r-- 1 root root 1263616 2018-02-13 12:44 libfastcvadsp.so
-rw-r--r-- 1 root root 550172 2017-03-08 03:55 libfastcvadsp_skel.so
-rw-r--r-- 1 root root 82272 2017-03-08 03:55 libobjectMattingApp_skel.so
-rw-r--r-- 1 root root 99808 2017-03-08 03:55 libscveBlobDescriptor_skel.so
-rw-r--r-- 1 root root 429140 2017-03-08 03:55 libscveCleverCapture_skel.so
-rw-r--r-- 1 root root 635648 2017-03-08 03:55 libscveFaceRecognition_skel.so
-rw-r--r-- 1 root root 41780 2017-03-08 03:55 libscveObjectSegmentation_skel.so
-rw-r--r-- 1 root root 399744 2017-03-08 03:55 libscveT2T_skel.so
-rw-r--r-- 1 root root 1487612 2017-03-08 03:55 libscveTextReco_skel.so
```

- Libraries for computer vision, face recognition etc.



# ▷ FastRPC – Available Libraries

- For every library libXXXXX.so
  - XXXXXX specifies the library *name*
  - libXXXXX\_skel.so
    - Unmarshalls parameters and calls actual implementation

# ▷ FastRPC – libadsprpc.so

- Use the library name to get a remote *handle*
- We can use the handle to *invoke* a function on aDSP
- libadsprpc.so
  - `remote_handle_open("libname", &handle)`
  - `remote_handle_invoke(handle, sc, args)`

# ▷ FastRPC – libadsprpc.so

- `remote_handle_invoke(int handle, int sc, remote_arg_t* args)`
- Argument `sc`: 0xAABBCCDE
  - AA: Method index and attributes
  - BB: Number of input buffers
  - CC: Number of output buffers
  - D: Number of input handles
  - E: Number of output handles

## ▷ FastRPC – libadsprpc.so

- remote\_arg\_t\* args

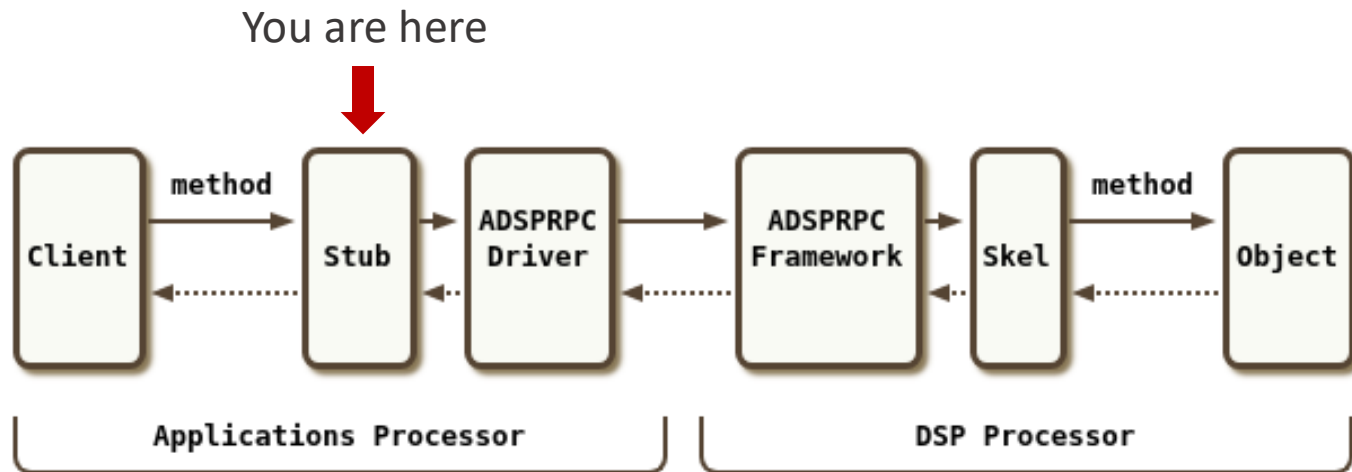
```
struct remote_buf {  
    void *pv;                /* buffer pointer */  
    ssize_t len;            /* length of buffer */  
};  
  
union remote_arg {  
    struct remote_buf buf;  /* buffer info */  
    uint32_t h;             /* remote handle */  
};
```

## ▷ FastRPC – libadsprpc.so

```
remote_arg_t args[] =
    .buf = {
        .pv = 0xdeadbee1,  /* Input #1 */
        .len = 0x1000
    },
    .buf = {
        .pv = 0xdeadbee2,  /* Input #2 */
        .len = 0x1000
    },
    .buf = {
        .pv = 0xdeadbee3,  /* Output #1 */
        .len = 0x1000
    }
}
```

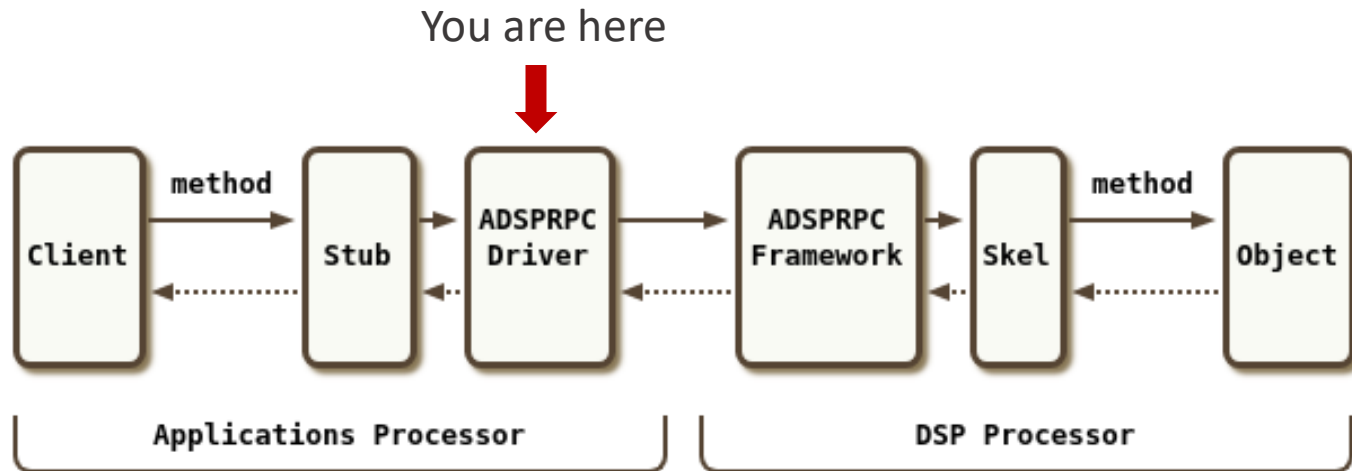
- Eg. Remote\_handle\_invoke(handle, 0x11020100, args)
  - Call method with index 0x11
  - 2 Input arguments, 1 Output argument

## ▷ FastRPC – Stub



- Autogenerated 'stub' libraries call `remote_handle_open/invoke` from `libadsprpc.so`
- Transparent to userspace
- `Remote_handle_open/invoke` are `ioctl` wrappers

# ▷ FastRPC - Kernel



- Kernel driver interface
  - `/dev/adsprpc-smd`
  - Protected by SELinux permissions
  - `ioctl()`

# ▷ FastRPC – IOCTL interface

- FASTRPC\_IOCTL\_INIT
- FASTRPC\_IOCTL\_INVOKE
- FASTRPC\_IOCTL\_MMAP
- FASTRPC\_IOCTL\_INVOKE\_FD
- FASTRPC\_IOCTL\_SETMODE



# ▷ FastRPC – IOCTL interface

- FASTRPC\_IOCTL\_INIT
- Load a user provided ELF to aDSP
  - ELF mapped to ION buffer
  - Pass ION pointer and file descriptor to ELF
  - Also pass memory buffer (?)
- libadsrpc loads '/dsp/fastrpc\_shell\_0'
- Lots of other Hexagon binaries under /dsp

# ▷ FastRPC – fastrpc\_shell\_0

- Hexagon ELF executable
  - Loads libXXXXX\_skell.so, libXXXXX.so
  - Delegates execution
  - Provides a few remote functions on its own
    - adsp\_ps – Show processes running on aDSP

## ▷ FastRPC – Kernel

- `remote_handle_open()` calls the following IOCTLs
  - `FASTRPC_IOCTL_INIT`
    - Loads 'fastrpc\_shell\_0' unto aDSP
  - `FASTRPC_IOCTL_INVOKE`
    - Invokes a remote function with a hardcoded handle!

# ▷ FastRPC – Kernel

- FASTRPC\_IOCTL\_INVOKE
  - remote\_handle\_invoke() is a thin wrapper for this
  - Same Arguments: handle, sc, remote\_args
  - Calls a remote function on aDSP

# ▷ FastRPC – Kernel

- `FASTRPC_IOCTL_INVOKE`
  - Called during `remote_handle_open()`
  - With `handle = 1`
  - A handle for system functions of some sort
  - Transfers execution to aDSP in order to get a proper handle for the library
- Actually all IOCTLs lead to a `FASTRPC_IOCTL_INVOKE` code with `handle = 1` and different method index

# ▷ FastRPC – Kernel

- Finally, a valid library handle is returned
- We can now call
  - `remote_handle_invoke(handle, sc, args)`
    - `FASTRPC_IOCTL_INVOKE`
  - Qualcomm Shared Memory Subsystem
  - But how are arguments passed to aDSP ?

## ▷ FastRPC – Kernel

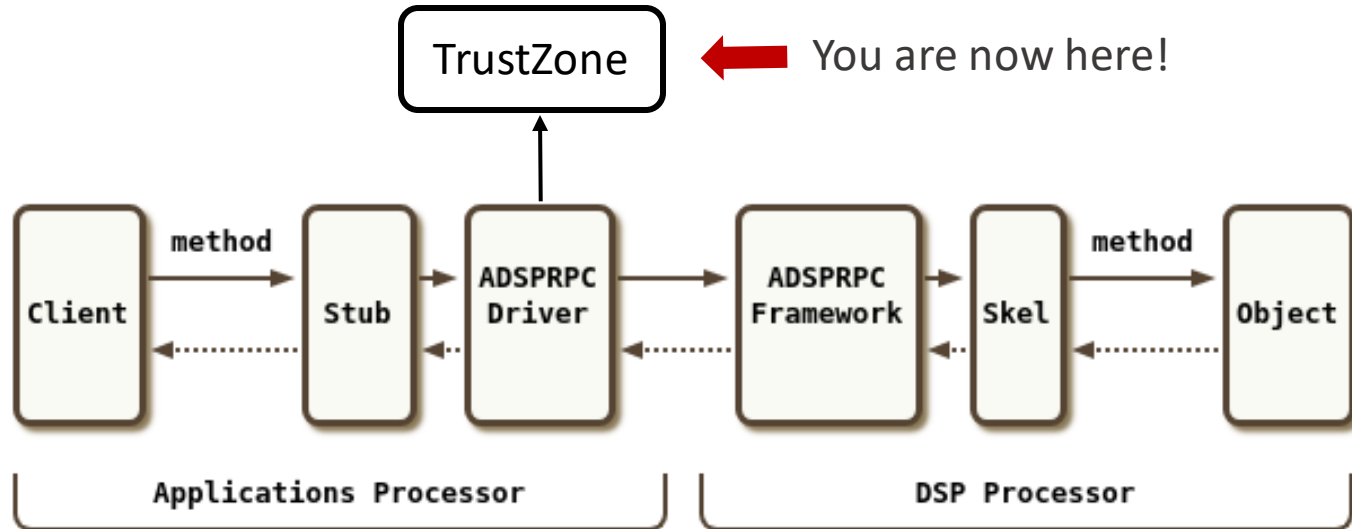
```
int hyp_assign_table(struct sg_table *table,
                    u32 *source_vm_list, int source_nelems,
                    int *dest_vmids, int *dest_perms,
                    int dest_nelems)
{
    ...
    desc.args[0] = virt_to_phys(info_list->list_head);
    desc.args[1] = info_list->list_size;
    desc.args[2] = virt_to_phys(source_vm_copy);
    desc.args[3] = sizeof(*source_vm_copy) * source_nelems;
    desc.args[4] = virt_to_phys(dest_info_list->dest_info);
    desc.args[5] = dest_info_list->list_size;
    desc.args[6] = 0;

    desc.arginfo = SCM_ARGS(7, SCM_RO, SCM_VAL, SCM_RO, SCM_VAL, SCM_RO,
                           SCM_VAL, SCM_VAL);
}
```

### ■ FASTRPC\_IOCTL\_INVOKE

- Maps remote\_args to Hexagon
- fastrpc\_buf\_alloc -> hyp\_assign\_phys -> hyp\_assign\_table
- Calling TrustZone with an SCM call

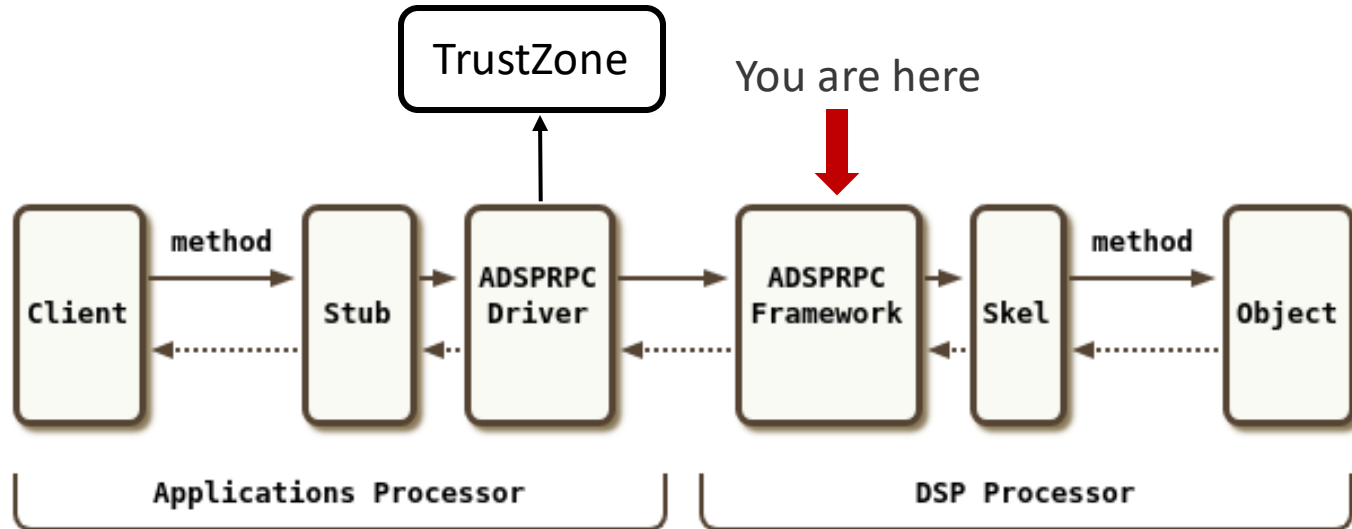
# ▷ FastRPC – TrustZone



- TrustZone
- Make argument memory accessible to aDSP
- MPU/SMMU Page Table Entries

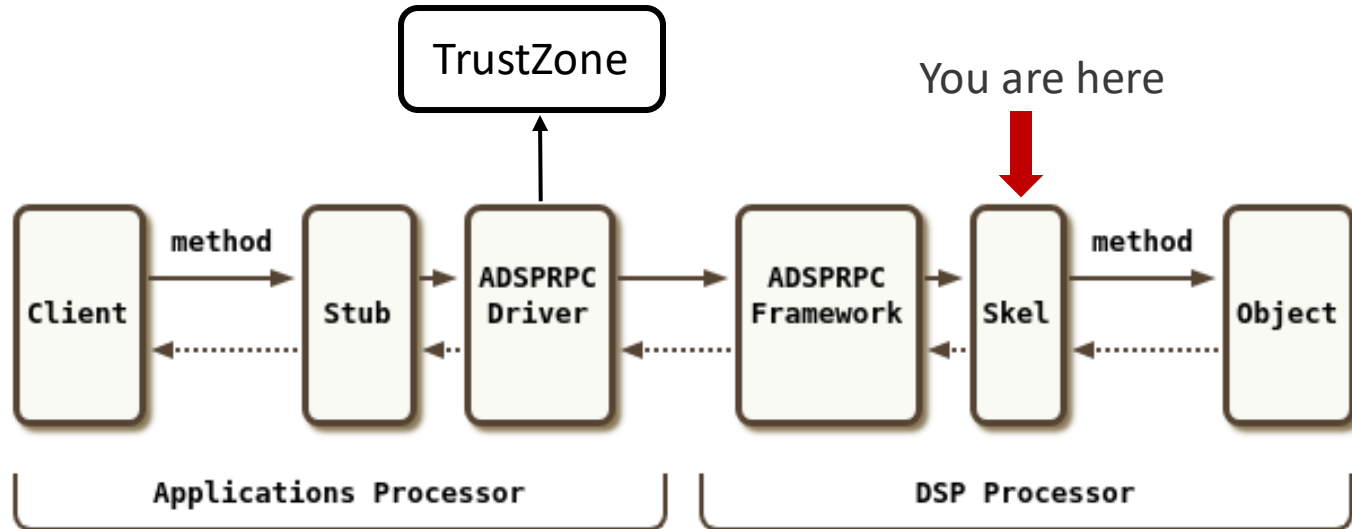


# ▷ FastRPC - QuRT



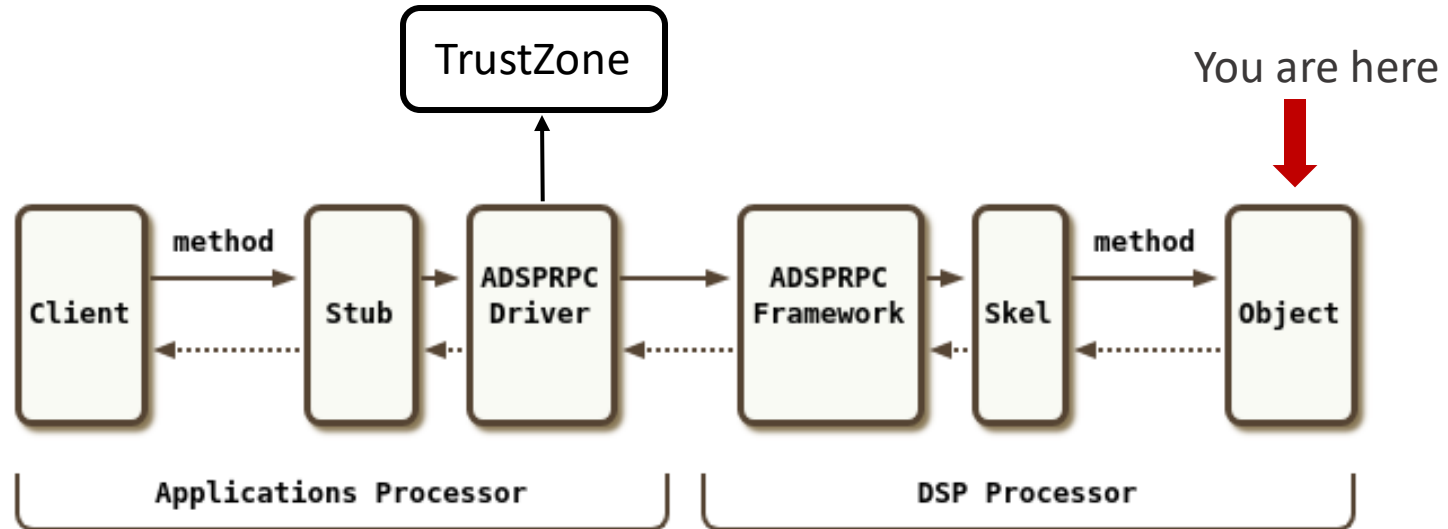
- QuRT passes execution to `fastrpc_shell_0`
- For the specific handle opened earlier, load skel library

## ▷ FastRPC - Skel



- Skel library unmarshalls arguments
- Call actual function implementation based on method index

## ▷ FastRPC - Library



- Skel library unmarshalls arguments
- Call actual function implementation based on method index

# ▷ FastRPC – Conclusion

- Now we know how to FastRPC works
- There are still many missing pieces
  - TrustZone mapping memory to aDSP
  - How QuRT delegates execution to libraries
  - We also saw calls with handle = 3 from the libadsprpc.so library but we could also perform our tests without them

▶ Custom code on aDSP

# ▷ Custom code on aDSP

- Hexagon SDK
  - Based on LLVM
  - Full toolchain - Compiler, readelf, objdump, simulator!
  - Utilities
  - Documentation

# ▷ Custom code on aDSP

- Put our code in remote filesystem and call it from userspace
- Remote filesystem is read-only
  - Get root and remount
- Remote libraries must be signed
  - Bypass sign check ?
  - Development board

# ▷ Custom code on aDSP

- Intrinsic Open-Q 820
  - ARM Development Board
  - MSM 8996/Snapdragon 820 (same as Pixel)
  - Exposes JTAG pins
  - Debug Fuse is enabled!



# ▷ Custom code on aDSP

- Debug Fuse
  - TrustZone
  - Enables execution of custom libraries on aDSP
- Create testsig.so and upload to remote filesystem
  - Generated by Hexagon SDK utilities
  - Needs device serial number
- And we can run our code on the development board

# ▷ Calculator Example

- Example code provided in Hexagon SDK
- Calculations performed on aDSP
- Python build script and custom makefiles
- Autogenerated stub/skel libraries

## ▷ Modified Example

```
int calculator_sum(int* vec, int vecLen, int64_t* out)
{
    *out = (int64_t)out;
    return 0;
}
```

```
msm8996:/vendor/bin # ./calculator
```

```
- starting calculator test
- ret = 55cf38
```

- We modify original calculator example
- We see aDSP's virtual address of 'out'

# ▷ Hardware Debugging

- Lauterbach32
  - Hardware Debugging
  - A few tens of thousands of \$
- OpenOCD and something like a Bus Blaster ?
  - No luck in my tests, but I am not the hardware type
  - There are some Lauterbach32 scripts that should be useful for bus offsets etc

# ▷ Software Debugging

- Hexagon SDK says debugging is supported on MSM8998 development boards
  - Not tested since I had MSM8996
- Qualcomm DIAG interface
  - Also used in baseband and Wi-Fi research
- Inject our own debugger in aDSP similar to "Exploring Qualcomm Baseband via ModKit" presentation

## ▷ Attack Surface

# ▷ Attack Surface

- Android Apps
  - stub libraries (marshalling)
- Kernel Driver
- aDSP
  - skel libraries (unmarshalling)
  - Implementation libraries

# ▷ Attack Surface

- Remotely, an attacker could send data that could be handled by aDSP/FastRPC code
  - Eg. Send audio/video that needs further processing
  - Browsers, messengers, etc
  - Attack on marshalling/unmarshaling libraries and implementation libraries on aDSP
- Locally, an attacker could also attack the kernel driver directly



# ▷ Attack Surface

- aDSP
  - A large number of libraries are exposed to userspace
  - Audio/video decoding, numerical calculations
    - Always a red flag for exploitation
  - System functions
- Open Question: Even after successful exploitation, do we cross a security boundary ?

# ▷ Attack Surface

- Exploiting a library on aDSP, we are in QuRT userspace
  - QuRT privilege escalation ?
  - TrustZone communication ?
- MPU blocks aDSP from accessing the whole memory
  - Maybe that's more than enough ?
- In newer SoCs, there are also cDSP and mDSP
  - Compute DSP, modem DSP
  - Offload work to baseband processor just like aDSP!

# ▷ Fuzzing

# ▷ FastCV

- Computer Vision Library by Qualcomm
- Provides ARM, GPU and Hexagon implementations
- Present on many Qualcomm Android devices

# ▷ FastCV

- 500+ available functions
  - Matrix multiplication
  - Hamming Distance
  - Allocate/deallocate structures
  - etc
- Available on aDSP through the "fastcvdsp" handle

# ▷ FastCV

- On remote filesystem:
  - Libfastcvadsp\_skel.so
    - Parameter unmarshall
  - Libfastcvadsp.so
    - Actual implementation
- Hexagon disassembler ?

# ▷ Hexagon Disassemblers

- IDA Pro and Ghidra do not support Hexagon natively
- hexagon-llvm-objdump
  - Provided by Hexagon SDK
  - Does NOT work for some binaries (?)
- <https://github.com/programa-stic/hexag00n>
  - Some immediate operands are decoded incorrectly
  - Ask me how I know

# ▷ Hexagon Disassemblers

- Radare2
  - Supported in newer versions including instruction packets
- Capstone internal build
  - Not public :(
- <https://github.com/gsmk/hexagon>
  - Less issues than the others
  - Register pairs are "different" than separate registers



# ▷ Ghidra Hexagon Support

- Ghidra makes adding support for new architecture easier
- SLEIGH Processor Specification Language
- Bonus: Decompiler
- I have implemented a few opcodes but there is a long way to go

# ▷ Ghidra Hexagon Support

```
define token instr(32)
    iclass = (28, 31)
    Rs = (16, 20)
    Rd = (0, 4)
    s16_lo = (5, 13)
    s16_hi = (21, 27)
;

:^ Rd = "add"(Rs, s16) is iclass=0b1011 & Rs & Rd
    & s16_hi & s16_lo [ s16 = (s16_hi << 9) + s16_lo; ]
{
    Rd = Rs + s16;
}
```

- Calculate immediate value, model instruction behavior inside braces
- Caret "^" denotes that Rd is not actually an instruction mnemonic
- Question to you: how to set "add" as the mnemonic ?

# ▷ Ghidra Hexagon Support

```
undefined std_toupper()  
undefined r0:1 <RETURN>  
std_toupper  
0007e824 e2 73 e0 bf r2 = add(r0,0xff9f)
```

- Verified with gmsk/hexagon, radare2
- Still a long way to go

# ▷ FastCV Skel library

```
.globl fastcvdsp_skel_invoke
fastcvdsp_skel_invoke:

@ FUNCTION CHUNK AT LOAD:0001F940 SIZE 000001C4 BYTES
@ FUNCTION CHUNK AT LOAD:0001FB10 SIZE 00000218 BYTES

{ allocframe (#off_58) }

loc_1ED14:
{ immext (#0x62B80)
  r7 = add (pc, ##loc_62B94)
  r2 = r0 ; memd (sp + #off_50) = r17:16 } @ r2 = sc
{ r5 = extractu (r2, #byte_5, #byte_18) @ r5 = Method Index
  memd (sp + #dword_48) = r19:18 ; memd (sp + #dword_40) = r21:20 }
{ r6 = r1 ; memd (sp + #byte_38) = r23:22 } @ r1 = remote_args_ptr
```

- We use gmsk/hexagon plugin in IDA
- Every skel library has a skel\_invoke function
- R0 = sc, R1 = remote\_args pointer

## ▷ FastCV Skel library

```
loc_1ED44:                @ 0x810c0
    r3 = memw (r7 + ##0xFFFF814) }
{ p0 = cmp.gtu (r5, #byte_1F)
  immext (#0xFFFF800)
  r4 = memw (r7 + ##0xFFFF818) } @ = 0x810c4
{ immext (#0xFFFF800)
  r16 = memw (r7 + ##0xFFFF81C) @ = 0x810c8
  immext (#0xFFFF800)
  r17 = memw (r7 + ##0xFFFF820) } @ = 0x810cc
{ if (p0) jump loc_1F880 }
```

- If method index > 0x1F return

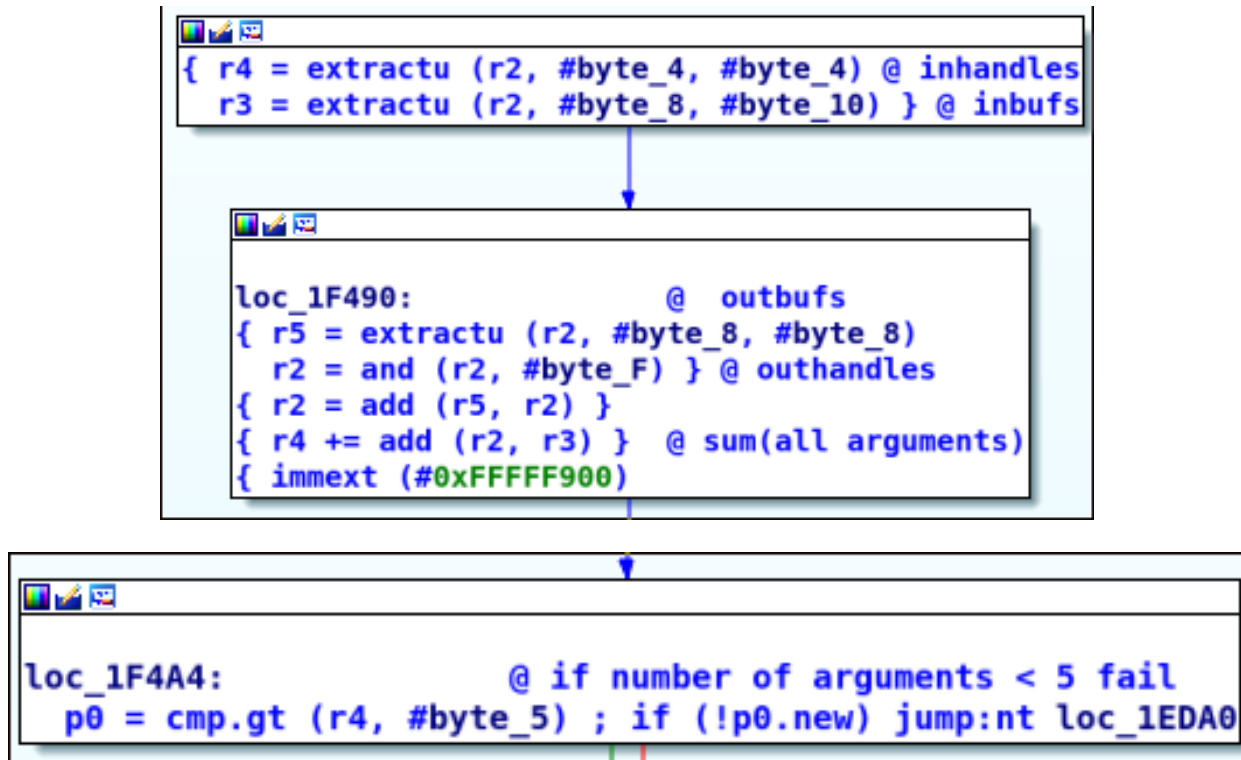
# ▷ FastCV Skel library

```
{ immext (#0xB40)
  r7 = add (pc, ##byte_B50) } @ PC = 0x1f8bc

                                @ DATA XREF: fastcvdsp_fcvICPJacobianErrorSE3f32Q+1DC+o
{ r5 = memw (r7 + r5 << #byte_2) } @ offset = *(0x1f8bc + (index << 2))
{ r5 = add (r5, r7) }
{ jumpr r5 }                    @ jump (0x1f8bc + offset)
```

- Else (if method index  $\leq$  0x1F):
  - Get offset from PC + (method index  $\ll$  2)
  - Add offset to PC and jump
  - Let's take offset 0xFFFFBBD0

# ▷ FastCV Skel library



- Validate number of arguments is correct

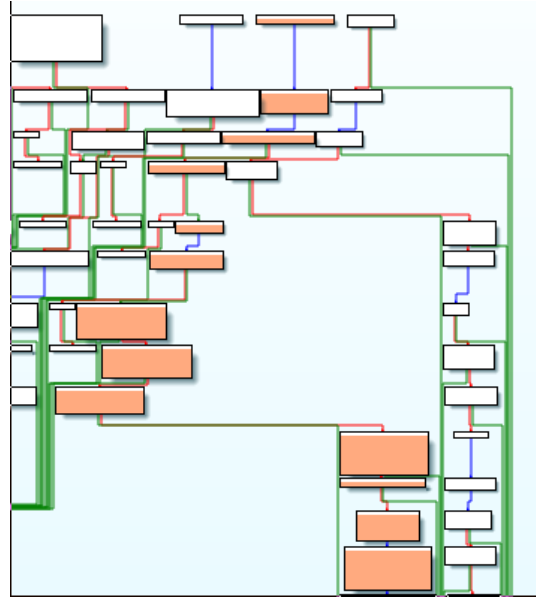
## ▷ FastCV Skel library

```
if (p0.new) r2 = memw (r6 + #byte_4) }  
{ r4 = #byte_14 @ r2 = remote_args[0].len  
  if (!cmp.gtu (r4.new, r2)) jump:t loc_1F4BC }
```

- Check if length of first remote\_arg > 14



# ▷ FastCV Skel library



- More checks for argument lengths
- Unmarshalling parameters, arithmetic shifts, etc
- A few basic blocks later ...

## ▷ FastCV Skel library

```
loc_1F5A0:  
    r0 = r19  
    r22 = memw (r6 + r3 << #byte_3)  
    r20 = memw (r6 + #byte_10) }  
{ r21 = memw (r6 + #byte_8) }  
{ call sub_4B240 }  
{ r5:4 = combine (r18, r19)  
  r3:2 = combine (r17, r20)  
  r1:0 = combine (r16, r21)  
  r6 = add (r29, #off_20) }  
{ call fastcvdsp_fcvCrossProduct3x1f32Q  
  memw (sp + #byte_0) = r23 ; memw (sp + #byte_4) = r6 }
```

- Finally call fastcvdsp\_fcvCrossProduct3x1f32Q

# ▷ FastCV Fuzzing

- We know how to call functions on the aDSP
- We analyzed how FastCV expects arguments
- A large number of complex functions are exposed
- Let's create the simplest fuzzer ever for FastCV

# ▷ FastCV Fuzzing

- Get a remote handle for FastCV
- Buffers with random data, but how many ? Method index?
  - For a sleepless night, parse FastCV header file, get expected number of argument, create a proper 'sc'
  - Reverse FastCV stub libraries and get 'sc' for each function

# ▷ FastCV Fuzzing

- We don't *need* any of this
- Skel library does not complain if we send more arguments than it expects!
- Try random method index ( $\leq 0x1F$ ) and hope for the best

# ▷ FastCV Fuzzing

```
130|msm8996:/data/local/tmp # ./fastrpc-fuzz
[+] Got handle: 0xa9f0d530, ret: 0x0
[+] invoke function index: 170, sc: 0xa080800, ret: e
[+] invoke function index: 195, sc: 0x3080800, ret: 0
[+] invoke function index: 104, sc: 0x8080800, ret: e
[+] invoke function index: 185, sc: 0x19080800, ret: e
[+] invoke function index: 120, sc: 0x18080800, ret: e
[+] invoke function index: 40, sc: 0x8080800, ret: e
[+] invoke function index: 89, sc: 0x19080800, ret: e
[+] invoke function index: 4, sc: 0x4080800, ret: ffffffff
[+] invoke function index: 29, sc: 0x1d080800, ret: 27
[+] invoke function index: 99, sc: 0x3080800, ret: 27
[+] invoke function index: 71, sc: 0x7080800, ret: 27
[+] invoke function index: 152, sc: 0x18080800, ret: 27
[+] invoke function index: 105, sc: 0x9080800, ret: 27
```

- After a few calls we get `-1` as return value
- Then only `0x27` ???

# ▷ FastCV Fuzzing

```
#define FASTRPC_ENOSUCH 39

static int fastrpc_internal_invoke(struct fastrpc_file *fl, uint32_t mode,
                                  uint32_t kernel,
                                  struct fastrpc_ioctl_invoke_fd *invokefd)
{
    ...
    if (!kernel) {
        VERIFY(err, 0 == context_restore_interrupted(fl, invokefd,
                                                    &ctx));

        if (err)
            goto bail;
        if (fl->sctx->smmu.faults)
            err = FASTRPC_ENOSUCH;
    }
}
```

- Kernel sets up SMMU for aDSP
- Sets fault handler for SMMU
- If fault return  $0x27 = 39$

# ▷ Fuzzing

- No luck in FastCV
- Let's try for the shrouded in mystery handle #1
- No need to `remote_handle_open`, we can invoke this handle directly just like the kernel!



# ▷ Crashes

```
[1563109.065921] Fatal error on adsp!  
[1563109.068361] adsp subsystem failure reason:      :Excep  :0:Exception detected:frpck_0_0.  
[1563109.083902] L-Notify: General: 8  
[1563109.195904] Kernel panic - not syncing: subsys-restart: Resetting the SoC - adsp crashed.
```

- System reboots on our development board with Android 7
- Tested on Pixel 3 with Snapdragon 845 (SD845) does not crash with latest firmware
- Evaluation
  - Analyze QuRT
  - Find function handler for handle = 1
  - Hexagon Simulator
  - Debug

# ▷ Conclusions

- aDSP is a very interesting exploitation target
- We can now fuzz libraries on aDSP
- Run our own code on aDSP for further investigation
- There is a lot of research waiting to be done here

# ▷ Future Work

- Proper disassembler/decompiler
- Investigate security boundary
- Debug
- Modern SoCs offer subsystems similar to aDSP
  - Apple Neural Engine
  - Google Pixel Visual Core
  - Huawei Neural Processing Unit

# ▷ References

1. A Journey Into Hexagon Dissecting a Qualcomm Baseband, Seamus Burke DEF CON 26 2018
2. Exploring Qualcomm Baseband via ModKit, Tencent Blade Team, CanSecWest 2018
3. Baseband exploitation in 2013: Hexagon challenges, Ralf-Philipp Weinmann PacSec 2013

*Thank you!*



**CENSUS**  
IT Security Works