In-Depth Analyzing and Fuzzing for Qualcomm Hexagon Processor

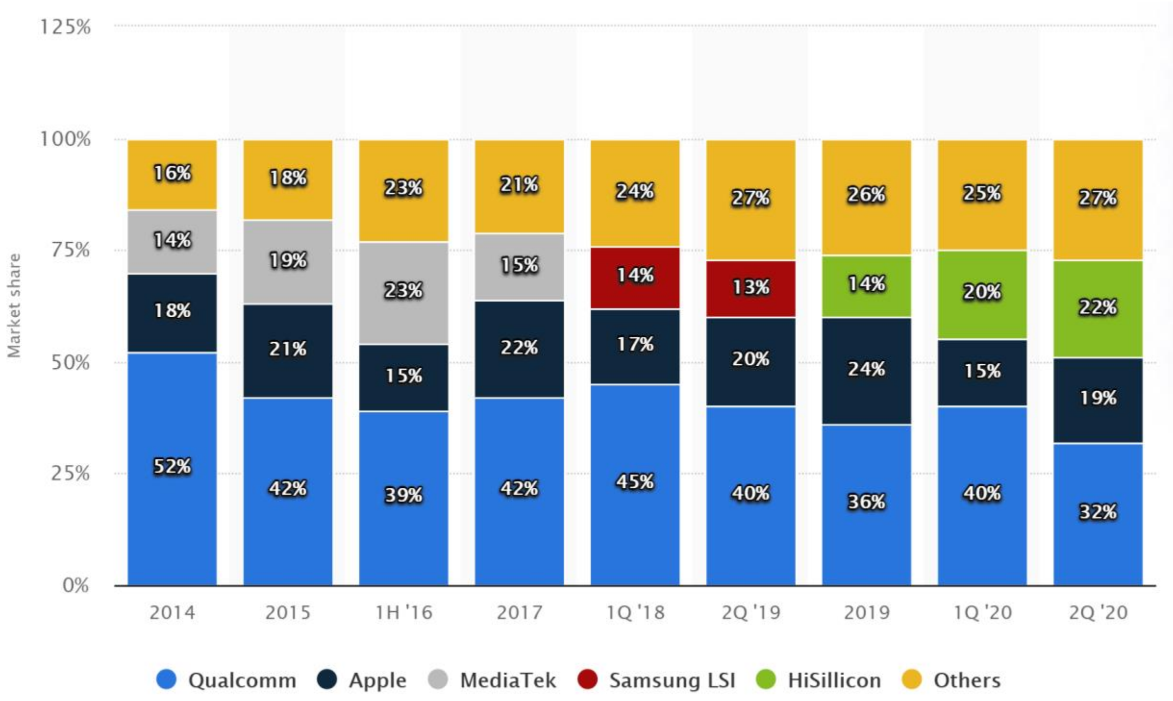
Xiling Gong of Google
Bo Zhang of Tencent Blade Team

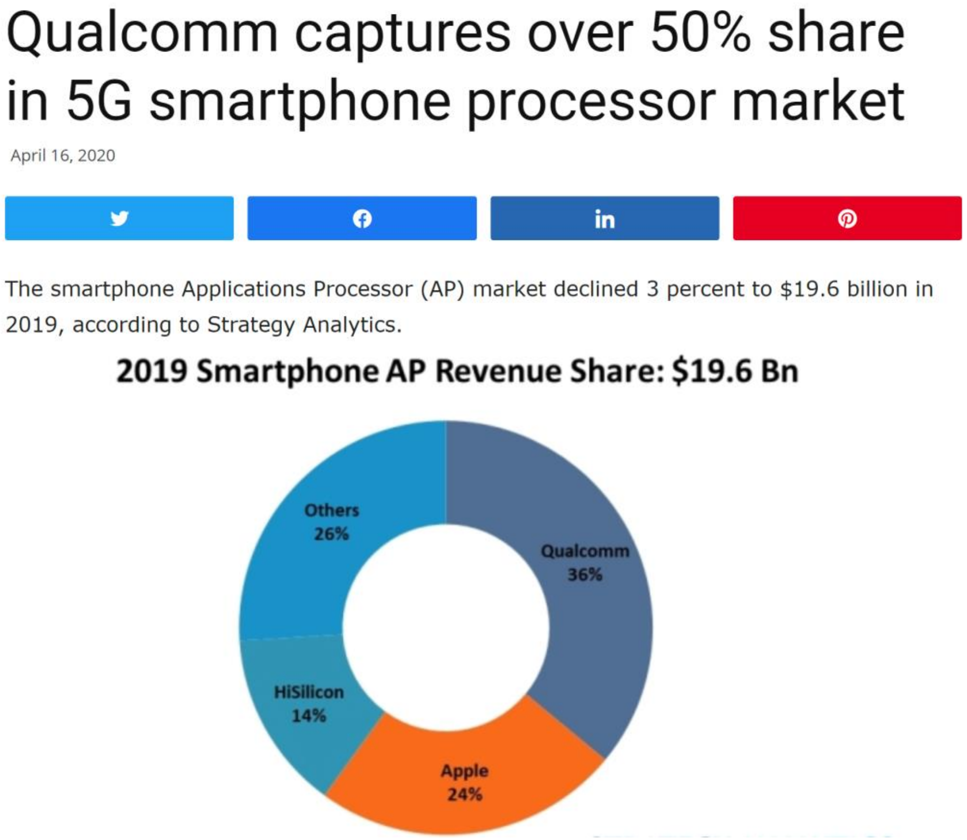
- This presentation belongs to Tencent Blade Team
- Xiling Gong is on behalf of himself

Agenda

- Background
  - Why Fuzzing Qualcomm Hexagon
  - Hexagon Basic
- The Hexagon Fuzzer
  - Possible Solutions and Tradeoff
  - Our Solution and Why
  - Overall Architecture
  - Key Components Explanation
  - Trouble Shooting
  - Fruits
- Demo

Background





Qualcomm captures over 50% share in 5G smartphone processor market

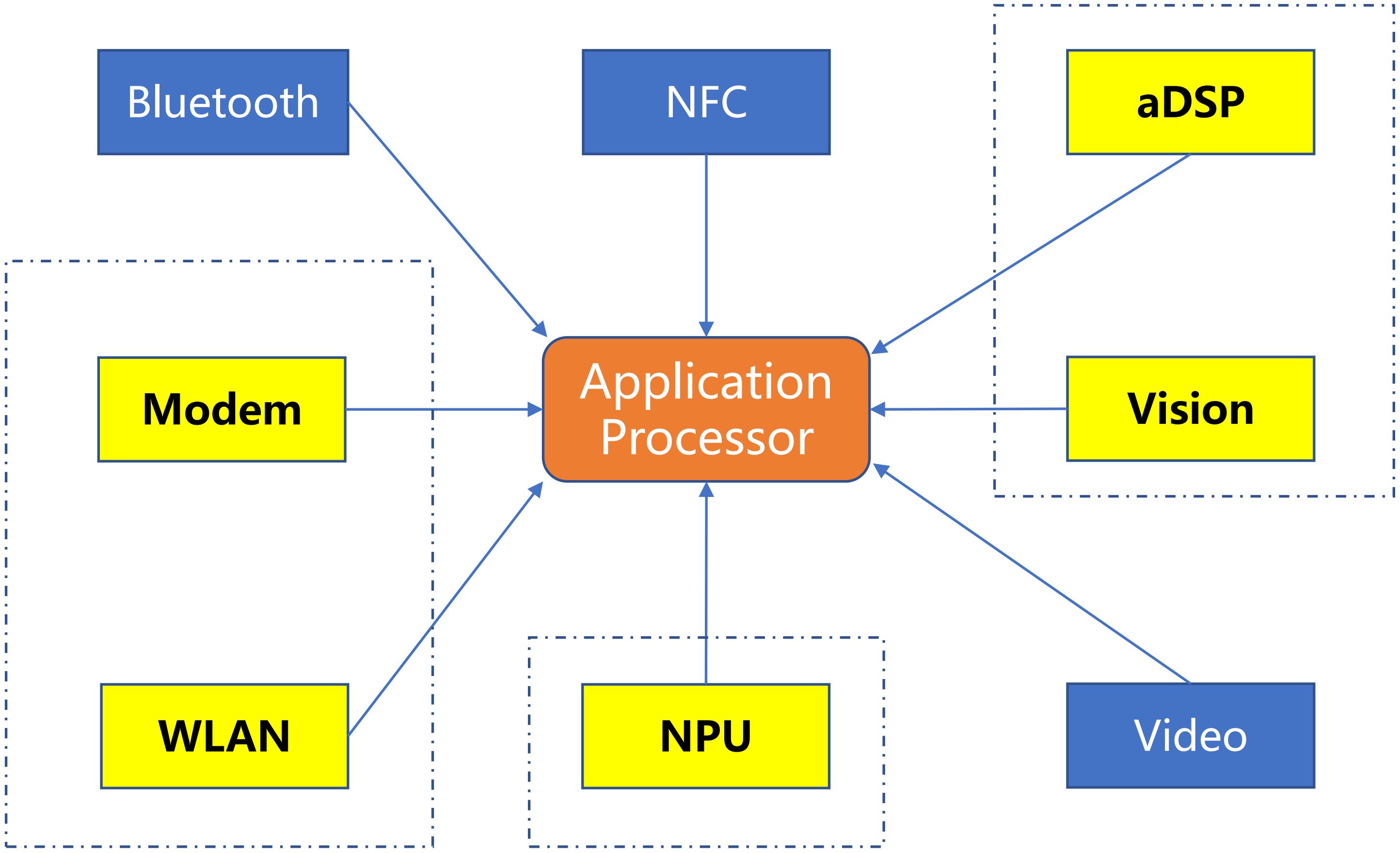
April 16, 2020

The smartphone Applications Processor (AP) market declined 3 percent to $19.6 billion in 2019, according to Strategy Analytics.

2019 Smartphone AP Revenue Share: $19.6 Bn

https://www.statista.com/statistics/233415/global-market-share-of-applications-processor-suppliers/
https://www.telecomlead.com/telecom-chips/qualcomm-captures-over-50-share-in-5g-smartphone-processor-market-94776

Qualcomm SOC



Subsystems using Hexagon

Baseband (Modem, WLAN)
aDSP (Audio, Camera, and other stuffs)
NPU (AI)

So Why Hexagon?

Hexagon is widely used in Qualcomm platform

Especially, Baseband/aDSP are pretty high value targets

Why Fuzzing Hexagon?

- Closed source
- No Hexagon decompiler
- No known effective Hexagon fuzzer (Coverage guided)
- Really complicated system (Baseband)
- Suitable for Fuzzing (aDSP)
- Feasible (will show you in this presentation)

→ Big Potential

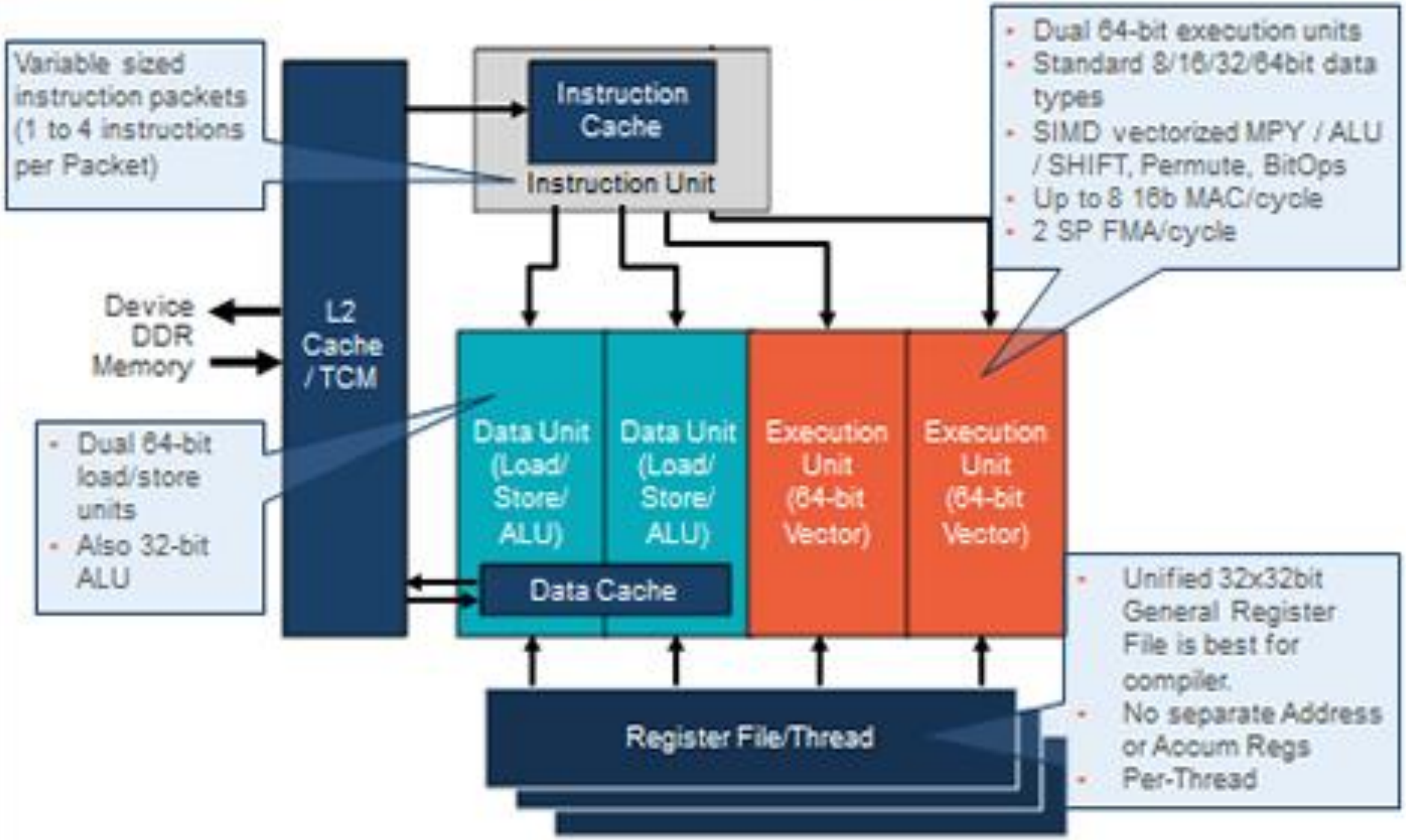
Hexagon Basic

- A Journey into Hexagon: Dissecting Qualcomm Basebands, 2018, Seamus Burke
- Exploring Qualcomm Baseband via ModKit, 2018, Tencent Blade Team
- Attacking Hexagon: Security Analysis of Qualcomm’s aDSP, 2019, Dimitrios Tatsis
- Advanced Hexagon Diag and getting started with baseband vulnerability research, 2020, Alisa Esage

Hexagon DSP Processor

- Memory
  - Program code and data are stored in a unified 32-bit address space
  - little-endian
- Registers
  - 32 32-bit general purpose registers can be accessed as single registers or as 64-bit register pairs
- Parallel Execution
  - Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution
  - Each packet contains from 1 to 4 instructions
- Cache Memory
  - Separate L1 instruction and data caches exist for program code and data
  - Unified L2 cache
- Virtual Memory
  - Real-Time OS (**QuRT**) handles the virtual-to-physical memory mapping
  - Virtual Memory supports the memory management and protection

VLIW: Area & power efficient multi-issue


Instruction Cache
Instruction Unit
L2 Cache / TCM
Device DDR Memory
Data Unit (Load/ Store/ ALU)
Data Unit (Load/ Store/ ALU)
Execution Unit (64-bit Vector)
Execution Unit (64-bit Vector)
Data Cache
Register File/Thread

Variable sized instruction packets (1 to 4 instructions per Packet)

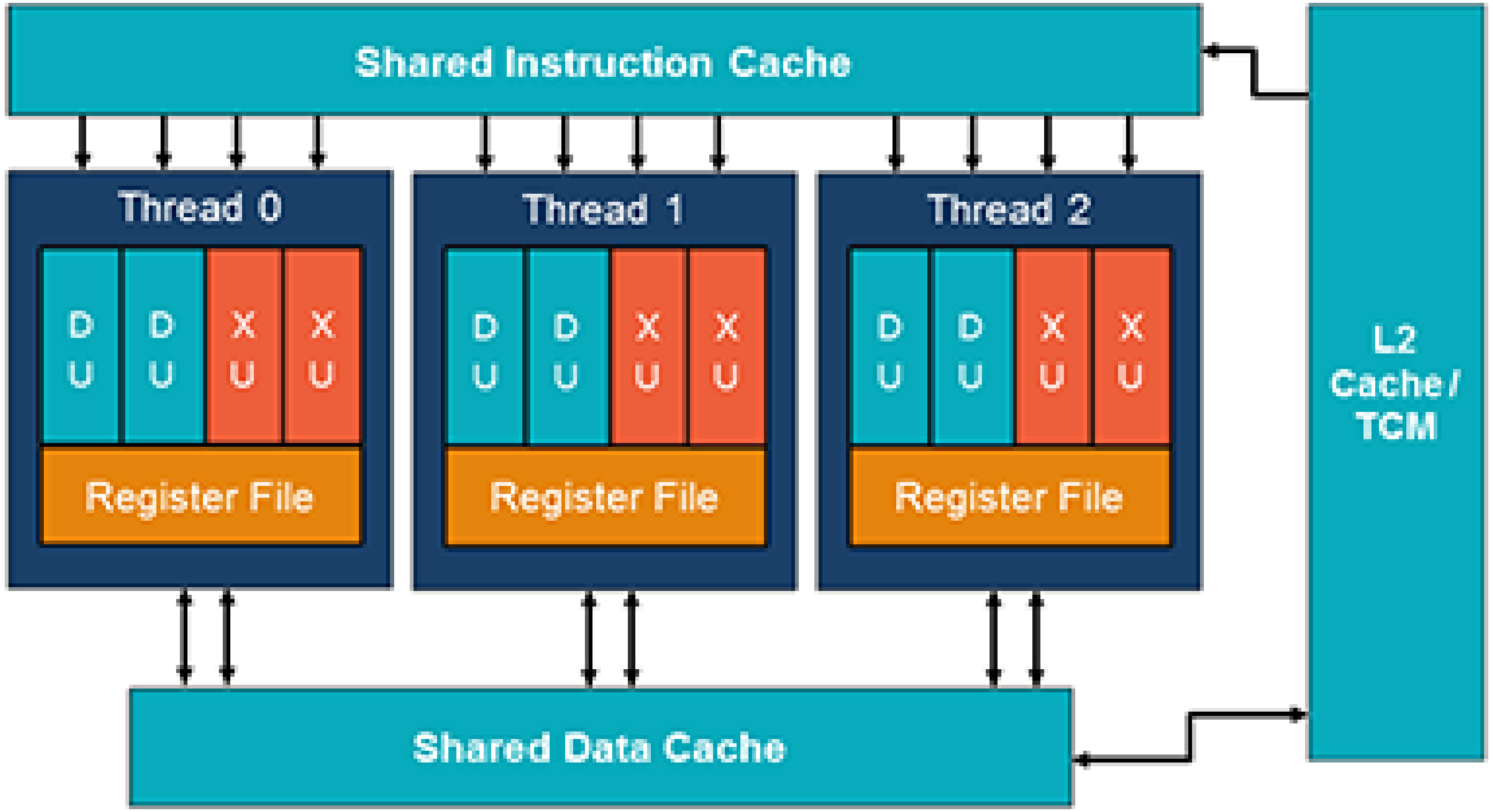
- Dual 64-bit execution units
- Standard 8/16/32/64bit data types
- SIMD vectorized MPY / ALU / SHIFT, Permute, BitOps
- Up to 8 16b MAC/cycle
- 2 SP FMA/cycle

- Dual 64-bit load/store units
- Also 32-bit ALU

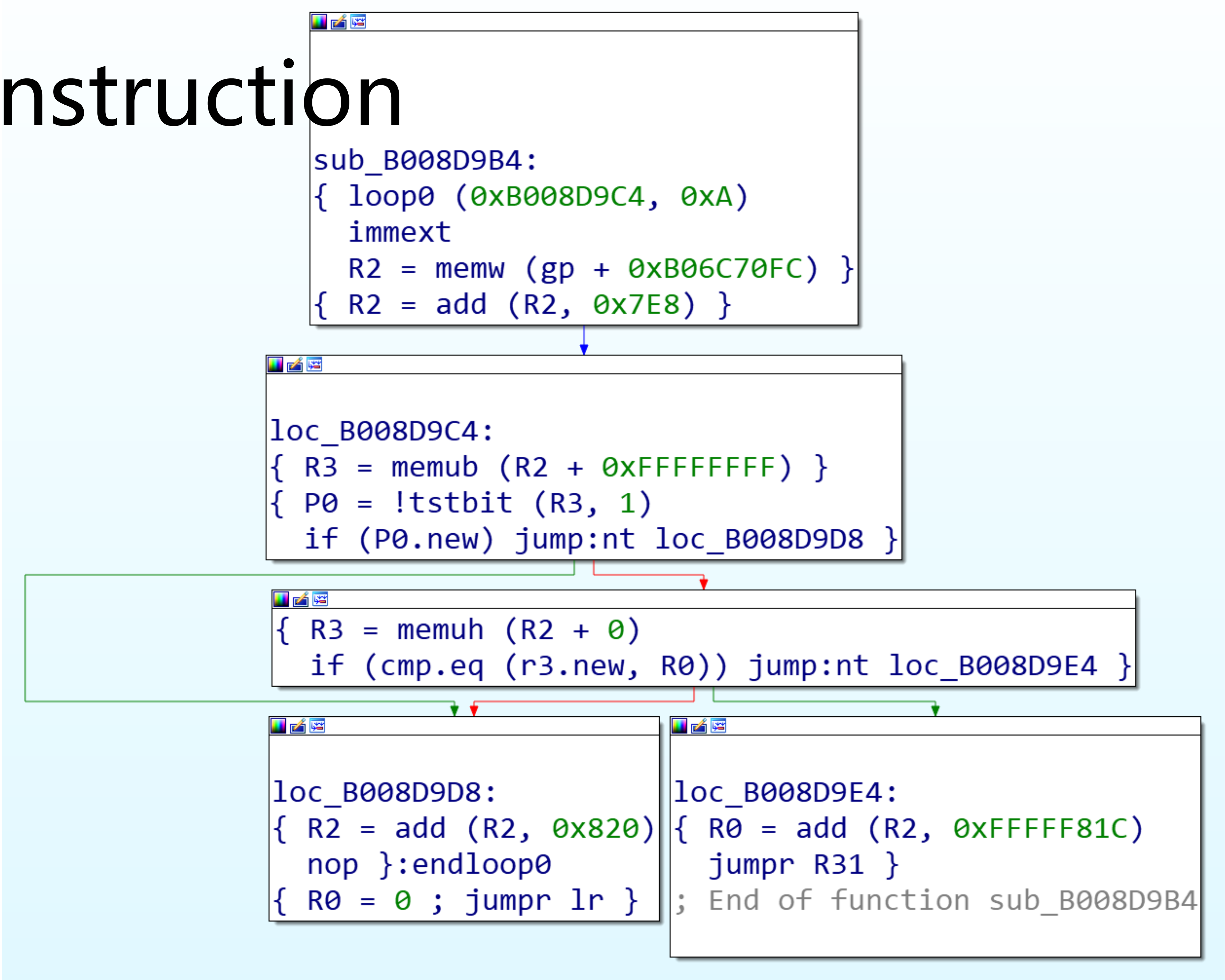
- Unified 32x32bit General Register File is best for compiler.
- No separate Address or Accum Regs
- Per-Thread

Programmer's view of Hexagon DSP HW multi-threading

- Hexagon V5 includes three hardware threads
- Architected to look like a multi-core with communication through shared memory



Hexagon Instruction



```
sub_B008D9B4:
{ loop0 (0xB008D9C4, 0xA)
  immext
  R2 = memw (gp + 0xB06C70FC) }
{ R2 = add (R2, 0x7E8) }
```

```
loc_B008D9C4:
{ R3 = memub (R2 + 0xFFFFFFFF) }
{ P0 = !tstbit (R3, 1)
  if (P0.new) jump:nt loc_B008D9D8 }
```

```
{ R3 = memuh (R2 + 0)
  if (cmp.eq (r3.new, R0)) jump:nt loc_B008D9E4 }
```

```
loc_B008D9D8:
{ R2 = add (R2, 0x820)
  nop }:endloop0
{ R0 = 0 ; jumpr lr }
```

```
loc_B008D9E4:
{ R0 = add (R2, 0xFFFFF81C)
  jumpr R31 }
; End of function sub_B008D9B4
```

Agenda

- Background
    - Why Qualcomm Hexagon
    - Hexagon basic
- The Hexagon Fuzzer
    - Possible Solution and Tradeoff
    - Our solution and why
    - Overall Architecture
    - Key Components Explanation
    - Trouble Shooting
    - Fruits
- Demo

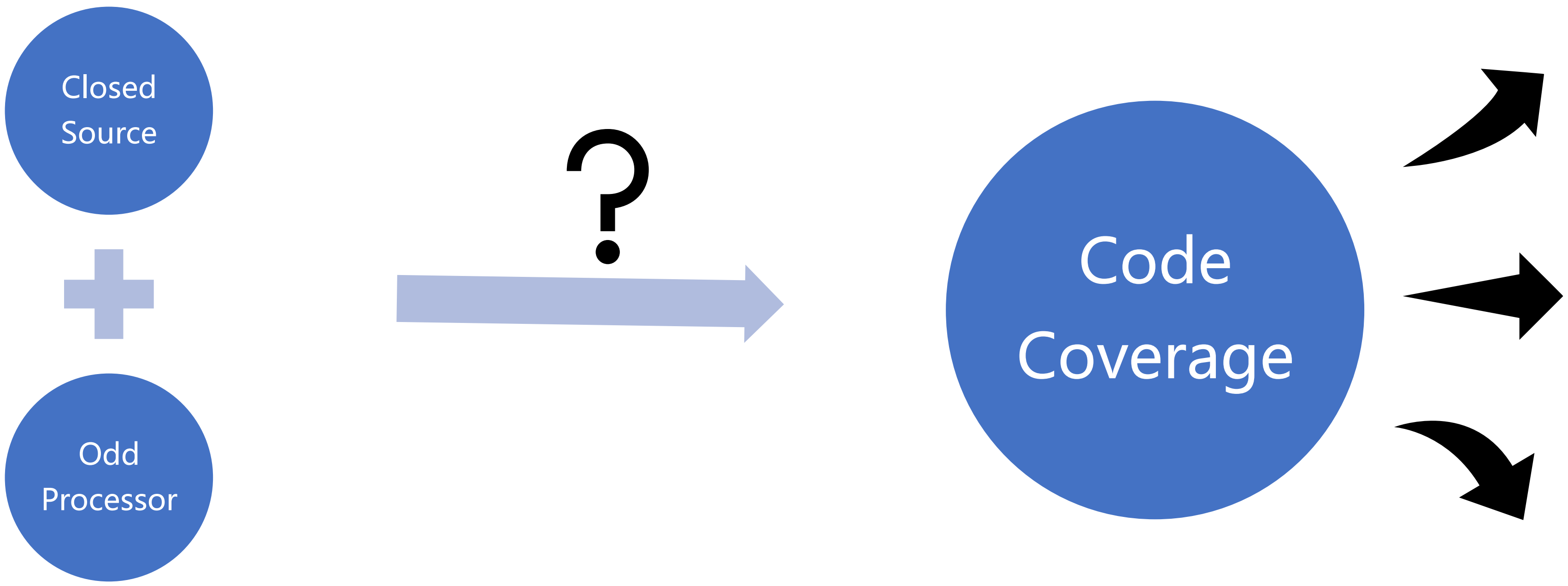
Possible Solution Of Fuzzer

Actually a problem of closed source target with odd processor

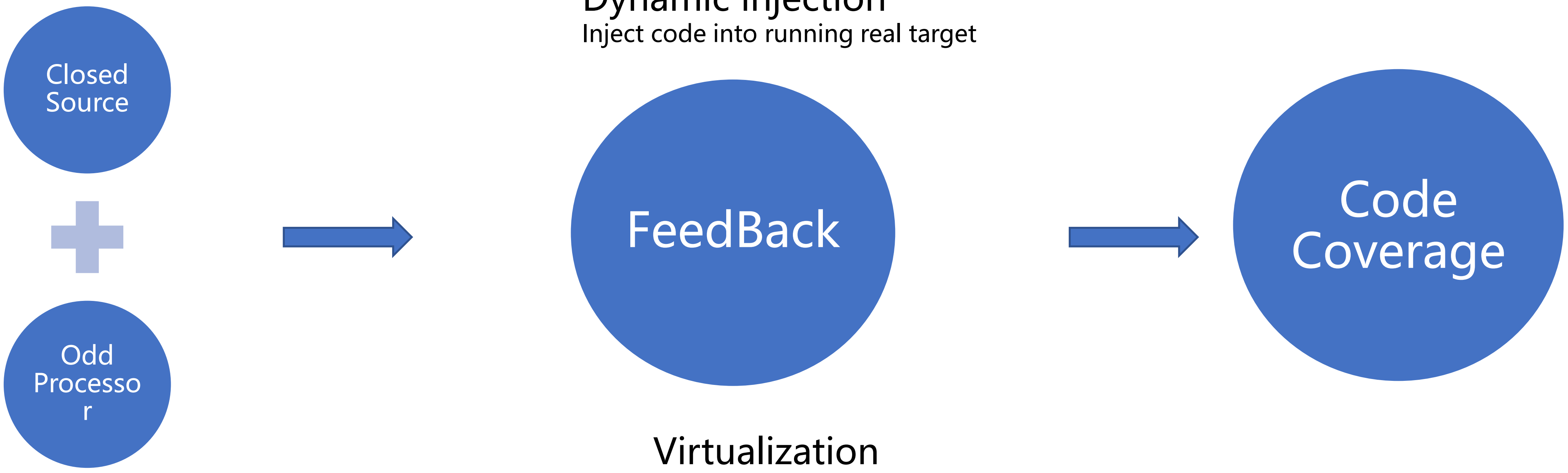
- 1 Dynamic Injection
  - Inject code into the REAL running target
- 2 Virtualization
  - Emulate the target

- ~~3 Symbolize Execution~~
- ~~4 AI…~~
- ~~5 Blackbox fuzzer~~

Possible Solutions Of Hexagon Fuzzer



Possible Solutions For Hexagon Fuzzer



Closed Source

Odd Processo
r

Dynamic injection
Inject code into running real target

FeedBack

Virtualization
Emulate the target

Code Coverage

Tradeoff (No Silver Bullet)

Dynamic Injection

- Cons
  - High cost
  - Low stability
  - Low flexibility
  - Low Performance
  - Low Scalability
  - Target should be debuggable
- Pros
  - Real running status
  - Real hardware
  - Deeper Code Coverage

Virtualization

- Cons
  - High cost (if no emulator available)
  - Hardware dependency
  - Fake running status
- Pros
  - High stability
  - High flexibility
  - High performance
  - High Scalability

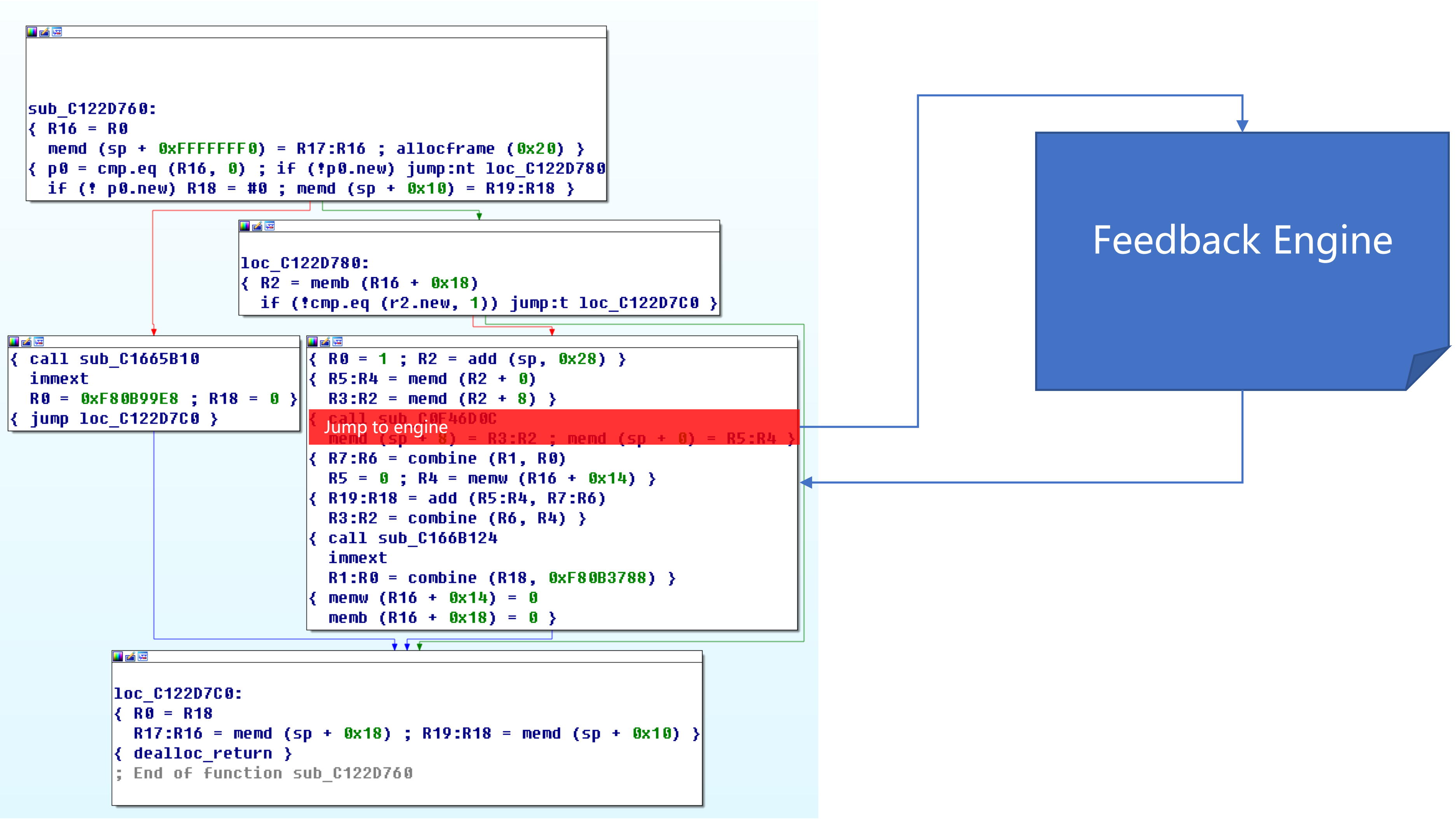
WE CHOOSE

DYNAMIC INJECTION!

Why We Choose Dynamic Injection

- No Hexagon emulator when we start the work
  - QEMU-Hexagon: Automatic Translation of the ISA Manual Pseudcode to Tiny Code Instructions, 2019, Niccolò Izzo, rev.ng & Taylor Simpson, Qualcomm Innovation
- Qualcomm Baseband is difficult to emulator
  - Heavily rely on hardware and running environment
  - Looks like infeasible
  - Even if feasible, it’s hard to improve the code coverage
- The first challenge of dynamic injection is INJECTION
  - However, we have a sophisticate debugger allow us to inject code into Hexagon processor

Dynamic Injection (Simple explanation)



```
sub_C122D760:
{ R16 = R0
  memd (sp + 0xFFFFFFF0) = R17:R16 ; allocframe (0x20) }
{ p0 = cmp.eq (R16, 0) ; if (!p0.new) jump:nt loc_C122D780
  if (! p0.new) R18 = #0 ; memd (sp + 0x10) = R19:R18 }
```

```
loc_C122D780:
{ R2 = memb (R16 + 0x18)
  if (!cmp.eq (r2.new, 1)) jump:t loc_C122D7C0 }
```

```
{ call sub_C1665B10
  immext
  R0 = 0xF80B99E8 ; R18 = 0 }
{ jump loc_C122D7C0 }
```

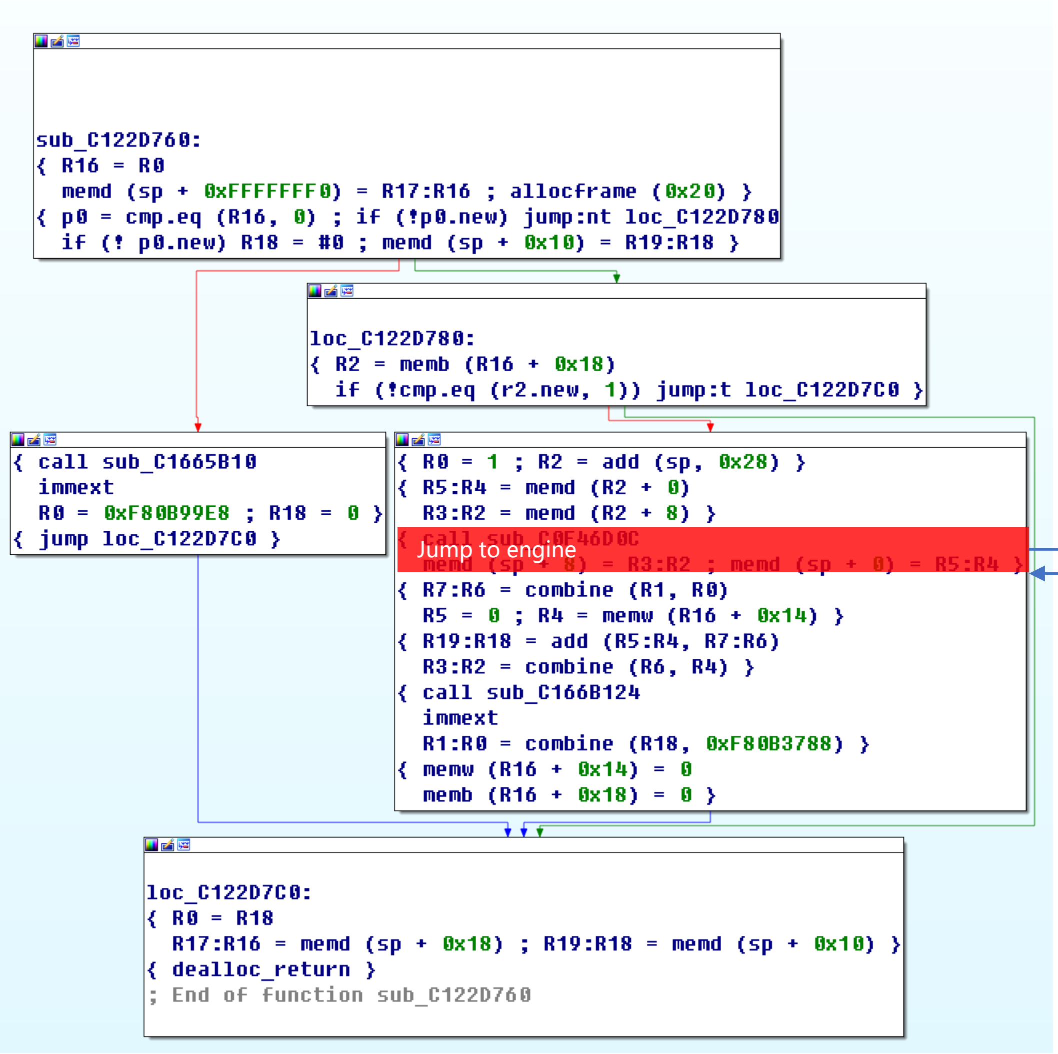
```
{ R0 = 1 ; R2 = add (sp, 0x28) }
{ R5:R4 = memd (R2 + 0)
  R3:R2 = memd (R2 + 8) }
{ call sub_C0F46D0C
  memd (sp + 8) = R3:R2 ; memd (sp + 0) = R5:R4 }
{ R7:R6 = combine (R1, R0)
  R5 = 0 ; R4 = memw (R16 + 0x14) }
{ R19:R18 = add (R5:R4, R7:R6)
  R3:R2 = combine (R6, R4) }
{ call sub_C166B124
  immext
  R1:R0 = combine (R18, 0xF80B3788) }
{ memw (R16 + 0x14) = 0
  memb (R16 + 0x18) = 0 }
```

Jump to engine

```
loc_C122D7C0:
{ R0 = R18
  R17:R16 = memd (sp + 0x18) ; R19:R18 = memd (sp + 0x10) }
{ dealloc_return }
; End of function sub_C122D760
```

Feedback Engine

Dynamic Injection (Simple explanation)



```
sub_C122D760:
{ R16 = R0
  memd (sp + 0xFFFFFFF0) = R17:R16 ; allocframe (0x20) }
{ p0 = cmp.eq (R16, 0) ; if (!p0.new) jump:nt loc_C122D780
  if (! p0.new) R18 = #0 ; memd (sp + 0x10) = R19:R18 }
```

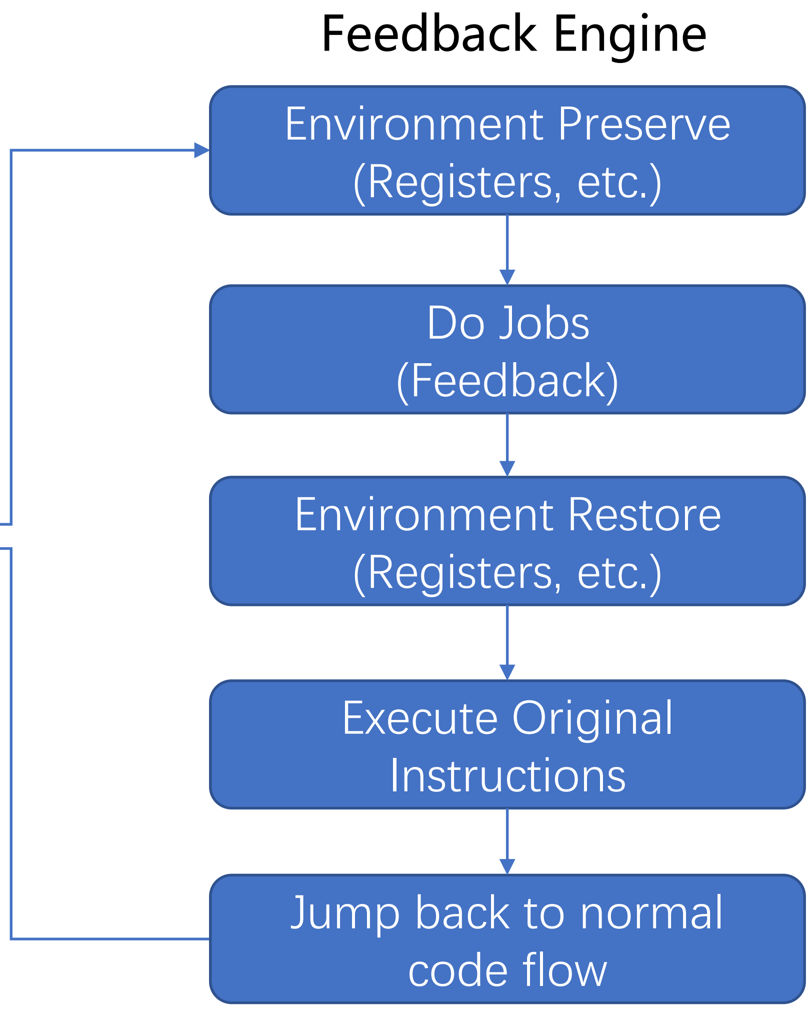
```
loc_C122D780:
{ R2 = memb (R16 + 0x18)
  if (!cmp.eq (r2.new, 1)) jump:t loc_C122D7C0 }
```

```
{ call sub_C1665B10
  immext
  R0 = 0xF80B99E8 ; R18 = 0 }
{ jump loc_C122D7C0 }
```

```
{ R0 = 1 ; R2 = add (sp, 0x28) }
{ R5:R4 = memd (R2 + 0)
  R3:R2 = memd (R2 + 8) }
{ call sub_C0E46D0C
  memd (sp + 8) = R3:R2 ; memd (sp + 0) = R5:R4 }
{ R7:R6 = combine (R1, R0)
  R5 = 0 ; R4 = memw (R16 + 0x14) }
{ R19:R18 = add (R5:R4, R7:R6)
  R3:R2 = combine (R6, R4) }
{ call sub_C166B124
  immext
  R1:R0 = combine (R18, 0xF80B3788) }
{ memw (R16 + 0x14) = 0
  memb (R16 + 0x18) = 0 }
```

Jump to engine

```
loc_C122D7C0:
{ R0 = R18
  R17:R16 = memd (sp + 0x18) ; R19:R18 = memd (sp + 0x10) }
{ dealloc_return }
; End of function sub_C122D760
```



Feedback Engine

Environment Preserve (Registers, etc.)

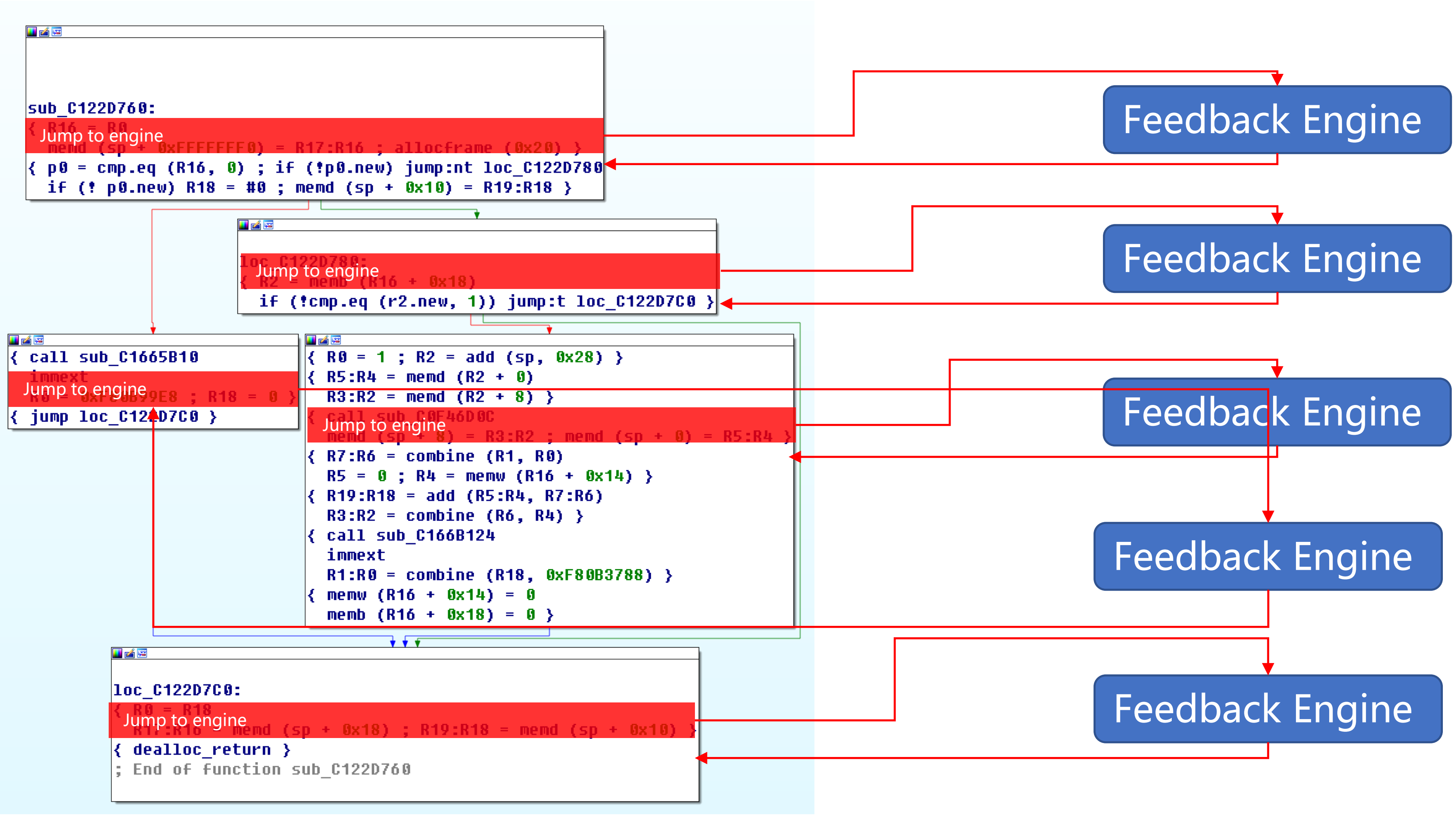
Do Jobs (Feedback)

Environment Restore (Registers, etc.)

Execute Original Instructions

Jump back to normal code flow

Dynamic Injection (Simple explanation)



```
sub_C122D760:
Jump to engine
{ p0 = cmp.eq (R16, 0) ; if (!p0.new) jump:nt loc_C122D780
  if (! p0.new) R18 = #0 ; memd (sp + 0x10) = R19:R18 }
```

```
Jump to engine
  if (!cmp.eq (r2.new, 1)) jump:t loc_C122D7C0 }
```

```
{ call sub_C1665B10
Jump to engine
{ jump loc_C122D7C0 }
```

```
{ R0 = 1 ; R2 = add (sp, 0x28) }
{ R5:R4 = memd (R2 + 0)
  R3:R2 = memd (R2 + 8) }
Jump to engine
{ R7:R6 = combine (R1, R0)
  R5 = 0 ; R4 = memw (R16 + 0x14) }
{ R19:R18 = add (R5:R4, R7:R6)
  R3:R2 = combine (R6, R4) }
{ call sub_C166B124
  immext
  R1:R0 = combine (R18, 0xF80B3788) }
{ memw (R16 + 0x14) = 0
  memb (R16 + 0x18) = 0 }
```

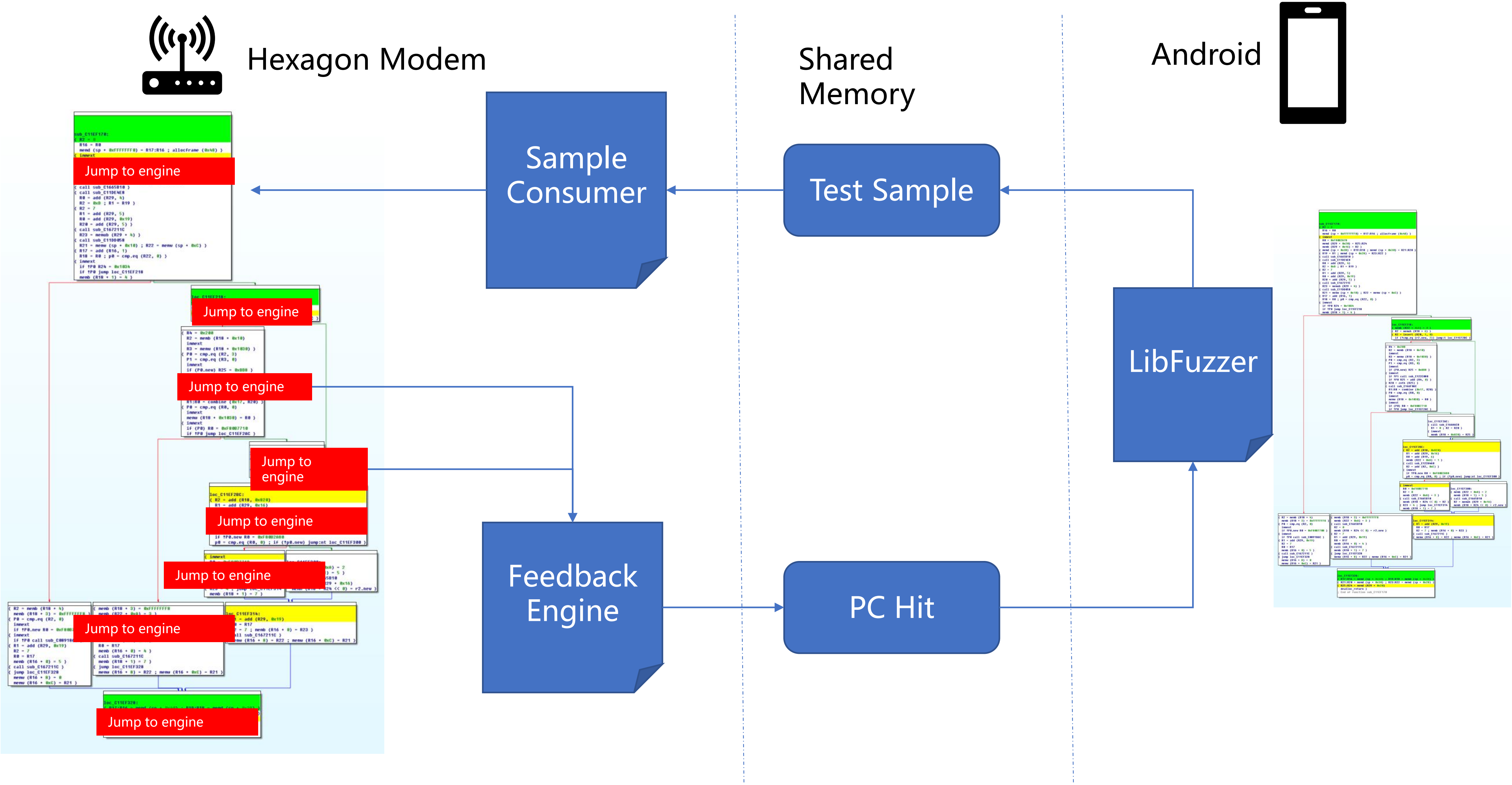
```
loc_C122D7C0:
Jump to engine
{ dealloc_return }
; End of function sub_C122D760
```

Overall Architecture

Hexagon
Debugger Engine
Feedback Engine

Android
Debugger
Libfuzzer

PC
Analyzer
Patch Generator



Hexagon Modem

Shared Memory

Android

Sample Consumer

Test Sample

LibFuzzer

Feedback Engine

PC Hit

Jump to engine

Jump to engine

Jump to engine

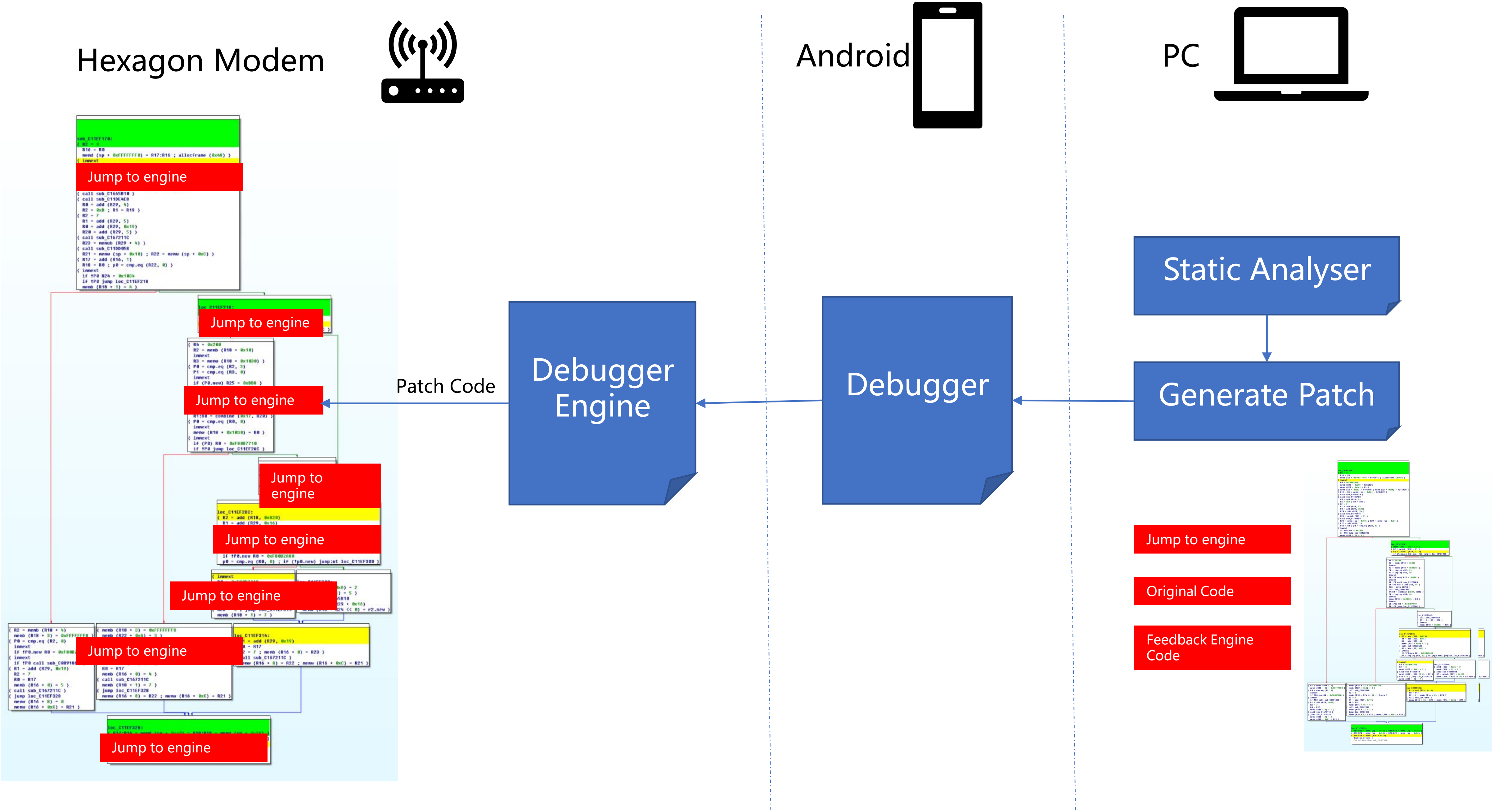
Jump to engine

Jump to engine

Jump to engine

Jump to engine

Jump to engine



Hexagon Modem

Android

PC

Static Analyser

Generate Patch

Debugger

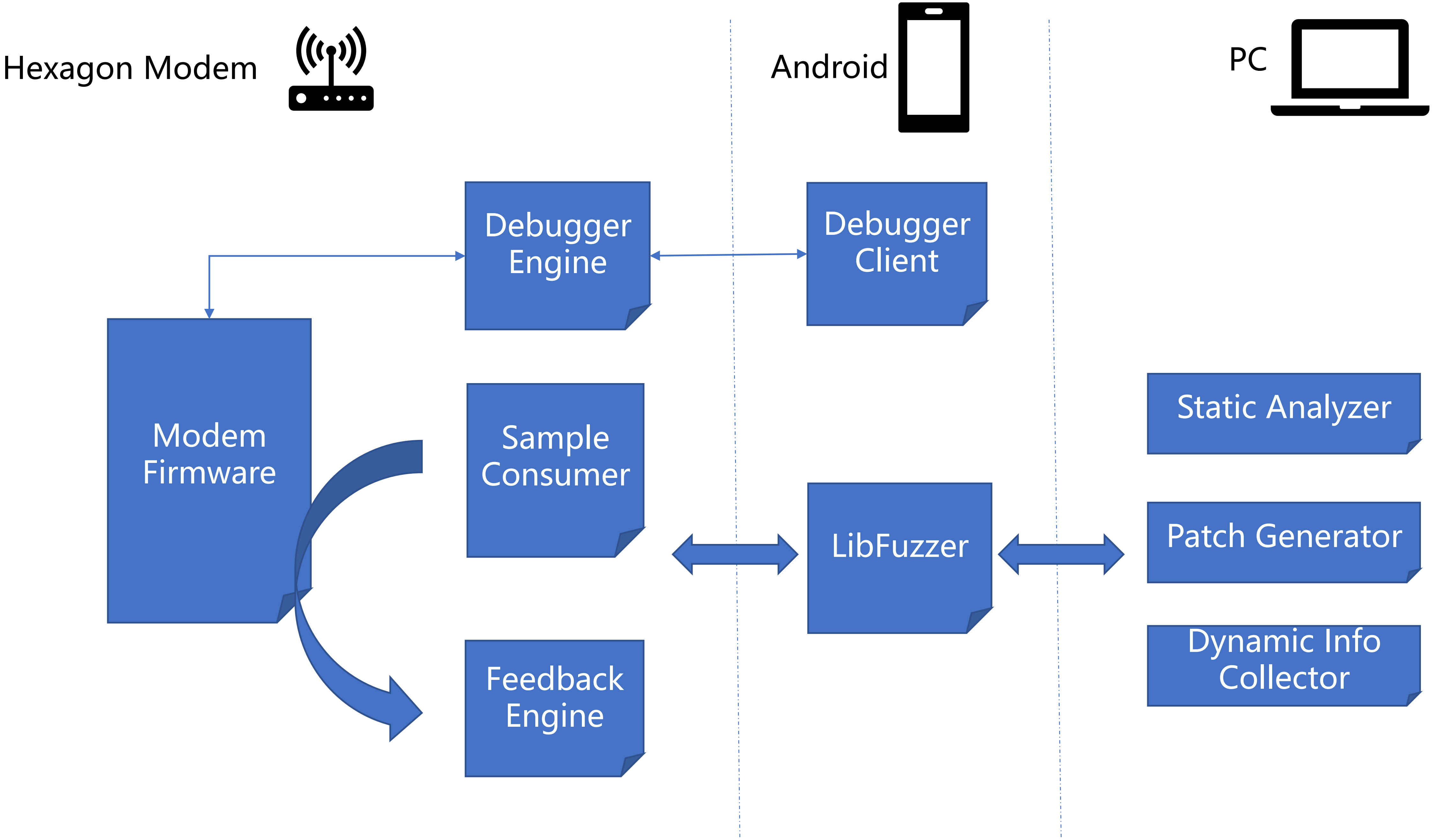
Debugger Engine

Patch Code

Jump to engine

Original Code

Feedback Engine Code


Hexagon Modem
Android
PC
Debugger Engine
Debugger Client
Modem Firmware
Sample Consumer
Feedback Engine
LibFuzzer
Static Analyzer
Patch Generator
Dynamic Info Collector

Trouble Shooting

- Stability of the debugger and feedback engine
  - Fix bug, fix bug…
  - (Stack depth, General register and condition register preserve, make sure the original instruction is execute correctly, etc.)
  - Good news is that you can eventually find and solve all the bugs

- Cost & Scalability & Performance
  - Using development board instead of phone
  - So you can deploy lots of fuzzers simultaneously
  - Also be aware of reduce the overhead of the fuzzer

Fruits

- 5+ Vulnerabilities
  - Fuzzer is still running
  - Will fuzze more components
- Lots of crashes
- Lots of asserts…

Related Works (Fuzzing)

- BaseSAFE: Baseband SAnitized Fuzzing through Emulation, 2020, Dominik Maier, Lukas Seidel, Shinjo Park
- Emulating Samsung's Baseband for Security Testing, 2020, Grant Hernandez, Marius Muench
- Attacking Hexagon: Security Analysis of Qualcomm’s aDSP, 2019, Dimitrios Tatsis

Related Works(Qualcomm Baseband)

- Reverse engineering a Qualcomm baseband, 2011, Guillaume Delugré
- All your baseband belongs to us, 2016, Ralf Weinmann
- A Journey into Hexagon: Dissecting Qualcomm Basebands, 2018, Seamus Burke
- Exploring Qualcomm Baseband via ModKit, 2018, Tencent Blade Team
- Exploiting Qualcomm WLAN and Modem Over The Air, 2019, Tencent Blade Team
- Advanced Hexagon Diag and getting started with baseband vulnerability research, 2020, Alisa Esage

THANK YOU!

BLADE

Tencent
Blade

Xiling Gong of Google
Bo Zhang of Tencent Blade Team

https://blade.tencent.com