Presented by: Joris Jonkers Both & Patrick Spaans
Supervisor: Alexandru Geana

riscure

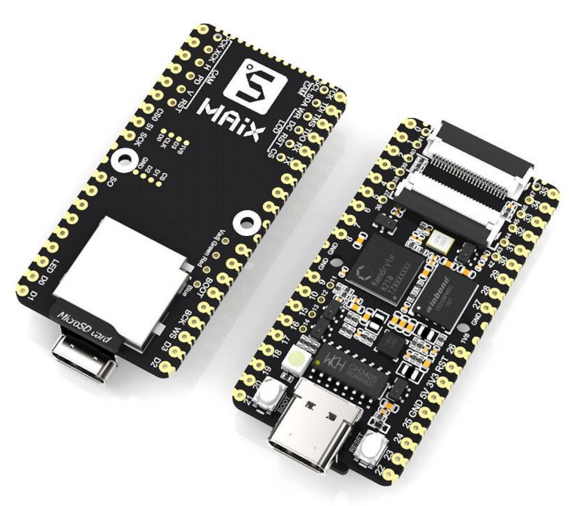
driving your security forward

Analyzing embedded software technologies on RISC-V64 using Ghidra

Introduction

RISC-V64

- Like ARM but open source
- One base image
- Extendable with extensions (e.g. M for multiplications)

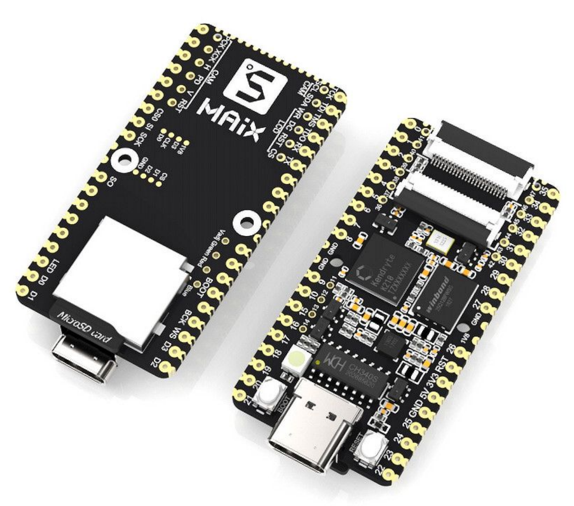


Introduction

RISC-V64

- Like ARM but open source
- One base image
- Extendable with extensions (e.g. M for multiplications)

Security of embedded systems



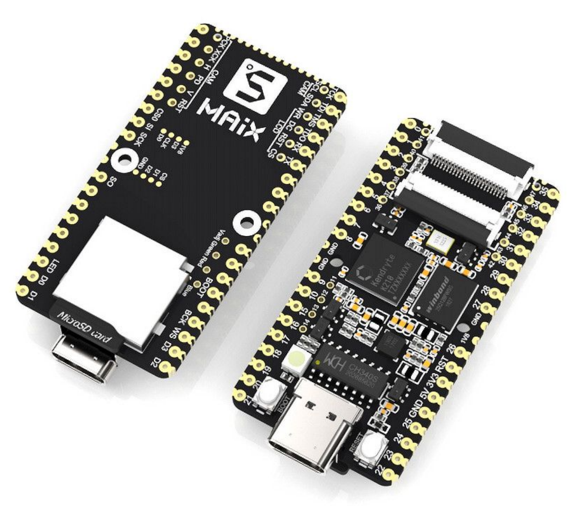
Introduction

RISC-V64

- Like ARM but open source
- One base image
- Extendable with extensions (e.g. M for multiplications)

Security of embedded systems

Ghidra SRE Framework



Introduction

RISC-V64

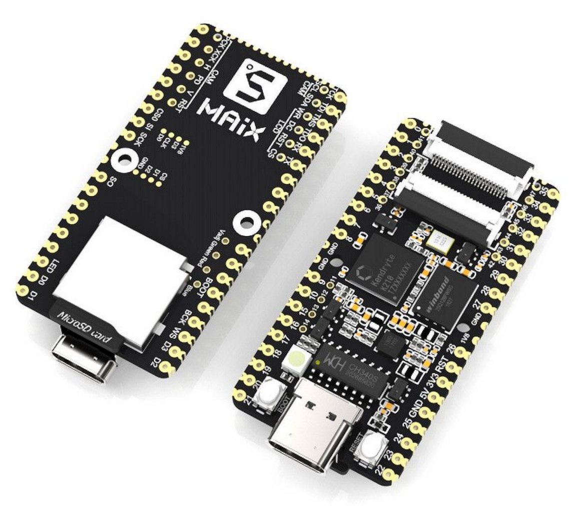
- Like ARM but open source
- One base image
- Extendable with extensions (e.g. M for multiplications)

Security of embedded systems

Ghidra SRE Framework

Kendryte K210 SoC

- System on a Chip
- Maix-bit
- AI capable IoT device



Related Work

Ghidra only recently open source

Analyzing security using reverse engineering is not a new concept

- Udupa et al. in 2005
- Zaddach and Costin in 2013

RISC-V64

Supported extensions

G (groups I, M, A, F, D):

- I → base integer instruction set
- M → standard integer multiplication & division extension
- A → standard atomic instruction extension
- F → single-precision floating-point extension
- D → standard double-precision floating-point extension
- C → standard extension for compressed instructions

- Q → standard extension for quad-precision floating-point

Base	Version	Frozen?
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	**Version**	**Frozen?**
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.2	N
N	1.1	N

RISC-V64

Supported extensions

G (brace grouping I, M, A, F, D):

- I → base integer instruction set
- M → standard integer multiplication & division extension
- A → standard atomic instruction extension
- F → single-precision floating-point extension
- D → standard double-precision floating-point extension
- C → standard extension for compressed instructions

Q → standard extension for quad-precision floating-point

So, Risc-V64GC == Risc-V64IMAFDC...

Base	Version	Frozen?
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	**Version**	**Frozen?**
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.2	N
N	1.1	N

Research Question

In what ways can a disassembly and decompile tool be used to analyze and enhance the working of embedded technologies?

Research Subquestions

- What are the possibilities of implementing a Ghidra plugin for RISC-V?
- What are the possibilities of using reverse-engineering to enable hidden features on the Kendryte K210?

Methodology

Creating a Ghidra Plugin for RISC-V64GC

Reverse engineering the Kendryte K210 bootrom

Research into writing to the Kendryte K210 OTP in order to implement secure boot

Creating a Ghidra Plugin

for RISC-V64GC

- Add support for architectures
- Specifies register layouts and hardware specs
- Must contain all instructions specifications to allow successful decompilation

```
88000288 13 01 01 fe
8800028c 23 3c 11 00
88000290 23 38 81 00
88000294 13 04 01 02
88000298 23 34 a4 fe
8800029c 23 30 b4 fe
880002a0 83 37 84 fe
880002a4 63 de 07 00
880002a8 83 37 84 fe
880002ac b3 07 f0 40
880002b0 23 34 f4 fe
880002b4 83 37 04 fe
880002b8 13 07 d0 02
880002bc 23 86 e7 00
```

→

```
addi    sp,sp,-0x20
sd      ra,0x18(sp)
sd      s0,0x10(sp)
addi    s0,sp,0x20
sd      a0,-0x18=>local_18(s0)
sd      a1,-0x20=>local_20(s0)
ld      a5,-0x18=>local_18(s0)
bge     a5,zero,LAB_880002c0
ld      a5,-0x18=>local_18(s0)
neg     a5,a5
sd      a5,-0x18=>local_18(s0)
ld      a5,-0x20=>local_20(s0)
addi    a4,zero,0x2d
sb      a4,0xc(a5)
```

→

```

void FUN_88000288(longlong param_1,longlong param_2)

{
  longlong local_18;

  local_18 = param_1;
  if (param_1 < 0) {
    local_18 = -param_1;
    *(undefined *)(param_2 + 0xc) = 0x2d;
  }
  FUN_8800013c(local_18,param_2);
  return;
}
```

Creating a Ghidra Plugin

Plugin structure

.ldefs file
(language definition)

```
<language processor="RISCV"
        endian="little"
        size="64"
        variant="RV64GC"
        version="1.0"
        slafile="riscv.lp64d.sla"
        processorspec="RV64GC.pspec"
        id="RISCV:LE:64:RV64GC">
    <description>RISC-V 64 little general purpose compressed</description>
    <compiler name="gcc" spec="riscv64-fp.cspec" id="gcc"/>
    <external_name tool="DWARF.register.mapping.file" name="riscv64-fp.dwarf"/>
</language>
```

Creating a Ghidra Plugin

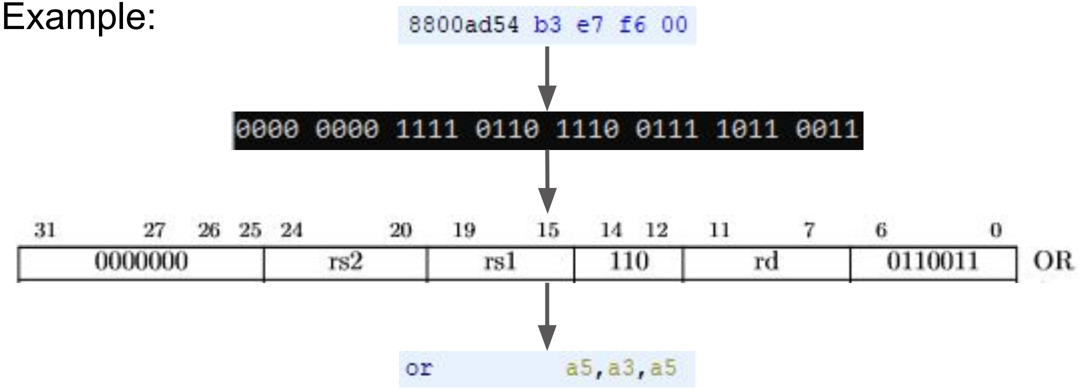
Plugin structure

.ldefs file
(language definition)

.sla file
(Instruction definitions)

```
# or d,s,t 00006033 fe00707f SIMPLE (0, 0)
:or rd,rs1,rs2 is RV32 & RVI & rs1 & rs2 & rd & op0001=0x3 & op0204=0x4 & op0506=0x1 & funct3=0x6 & funct7=0x0
{
    rd = rs1 | rs2;
}
```

Example:



Creating a Ghidra Plugin

Plugin structure

.ldefs file
(language definition)

.sla file
(Instruction definitions)

.pspec file
(Processor specification)

```
<?xml version="1.0" encoding="UTF-8"?>

<processor_spec>
    <programcounter register="pc"/>
    <context_data>
        <context_set space="ram">
            <set name="RV64" val="1"/>
            <set name="RVG" val="0x1F"/>
            <set name="RVC" val="1"/>
        </context_set>
    </context_data>
</processor_spec>
```

Creating a Ghidra Plugin

Plugin structure

.ldefs file
(language definition)

.sla file
(Instruction definitions)

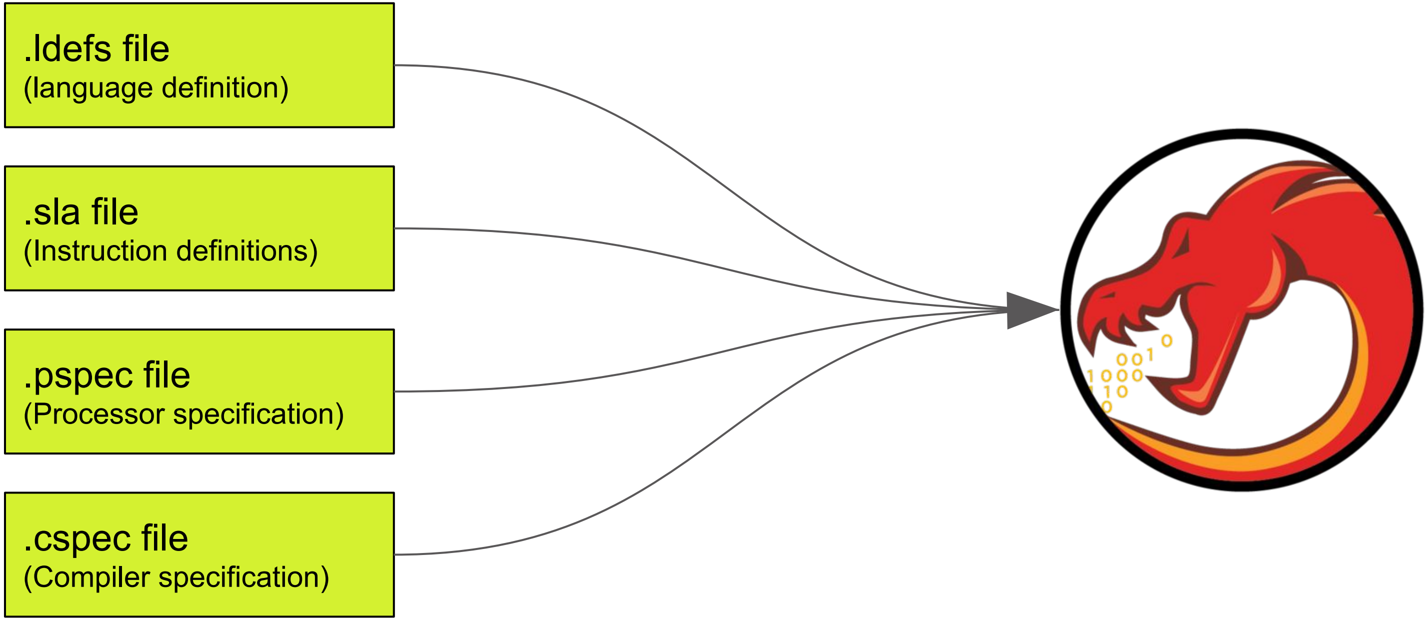
.pspec file
(Processor specification)

.cspec file
(Compiler specification)

```
<compiler_spec>
    <data_organization>
        <absolute_max_alignment value="0" />
        <machine_alignment value="8" />
        <default_alignment value="1" />
        <default_pointer_alignment value="8" />
        <pointer_size value="8" />
        <short_size value="2" />
        <integer_size value="4" />
        <long_size value="4" />
        <long_long_size value="8" />
        <float_size value="4" />
        <double_size value="8" />
        <size_alignment_map>
            <entry size="1" alignment="1" />
            <entry size="2" alignment="2" />
            <entry size="4" alignment="4" />
            <entry size="8" alignment="4" />
        </size_alignment_map>
    </data_organization>
    <spacebase name="gp" register="gp" space="ram"/>
    <global>
        <range space="ram"/>
        <register name="gp"/>
        <register name="tp"/>
    </global>
    <returnaddress>
        <register name="ra"/>
    </returnaddress>
    <stackpointer register="sp" space="ram"/>
    <default_proto>
    <prototype name="__stdcall" extrapop="0" stackshift="0" strategy="register">
        <input>
    <pentry minsize="1" maxsize="8">
        <register name="a0"/>
            </pentry>
    <pentry minsize="1" maxsize="8">
        <register name="a1"/>
            </pentry>
    ...
```

Creating a Ghidra Plugin

Plugin structure



Reverse engineering the Kendryte K210 bootrom

Using the plugin

```
88000000 73 50 30 30     csrrwi      zero,mideleg,0x0
88000004 73 50 20 30     csrrwi      zero,medeleg,0x0
88000008 73 50 40 30     csrrwi      zero,mie,0x0
8800000c 73 50 40 34     csrrwi      zero,mip,0x0
88000010 97 02 00 00     auipc       t0,0x0
88000014 93 82 c2 07     addi        t0,t0,0x7c
88000018 73 90 52 30     csrrw       zero,mtvec,t0
8800001c b7 62 00 00     lui         t0,0x6
88000020 73 a0 02 30     csrrs       zero,mstatus,t0
88000024 97 c1 5f f8     auipc       gp,-0x7a04
88000028 93 81 c1 7d     addi        gp,gp,0x7dc
8800002c 17 c2 5f f8     auipc       tp,-0x7a04
88000030 02 32           c.fldsp     ft4,0x20(sp)
88000032 01 72           c.lui       tp,-0x20
88000034 02 fc           c.sdsp      zero,0x38(sp)
88000036 73 25 40 f1     csrrs       a0,mhartid,zero
8800003a 45 15           c.addi      a0,-0xf
8800003c 00 05           c.addi4spn  s0,sp,0x280
8800003e 15 00           c.nop
88000040 16 d5           c.swsp      t0,0xa8(sp)
  ...
```

Reverse engineering the Kendryte K210 bootrom

Using the plugin

```
...
88000066 73 00 50 10     wfi
8800006a f3 27 40 34     csrrs      a5,mip,zero
8800006e 93 f7 87 00     andi       a5,a5,0x8
88000072 e3 8a 07 fe     beq        a5,zero,LAB_88000066
88000076 1b 03 10 00     addiw      t1,zero,0x1
8800007a f3              ??         F3h
8800007b 01              ??         01h
8800007c e7              ??         E7h
8800007d 00              ??         00h
8800007e 03              ??         03h
8800007f 00              ??         00h
88000080 6f              ??         6Fh    o
88000081 00              ??         00h
...
```

Can be: “f3 01 e7 00” or “f3 01”
Neither are in the documentation

Reverse engineering the Kendryte K210 bootrom

Using an alternative reverse engineering tool

An alternative to Ghidra could be used to find out more about these functions.

Reverse engineering the Kendryte K210 bootrom

Using an alternative reverse engineering tool

An alternative to Ghidra could be used to find out more about these functions.

Ghidra

```
88000066 73 00 50 10      wfi
8800006a f3 27 40 34      csrrs
8800006e 93 f7 87 00      andi
88000072 e3 8a 07 fe      beq
88000076 1b 03 10 00      addiw
8800007a f3               ??
8800007b 01               ??
8800007c e7               ??
8800007d 00               ??
8800007e 03               ??
8800007f 00               ??
88000080 6f               ??
88000081 00               ??
```

Radare2

```
0x88000062    f36744307300     xor byte [ebx], r14b
0x88000068    50               push rax
0x88000069    10f3             adc bl, dh
0x8800006b    27               invalid
0x8800006c    403493           xor al, 0x93            ; 147
0x8800006f    f78700e38a07.    test dword [rdi + 0x78ae300], 0x10031bfe
0x88000079    00f3             add bl, dh
0x8800007b    01e7             add edi, esp
0x8800007d    0003             add byte [rbx], al
0x8800007f    006f00           add byte [rdi], ch
```

Reverse engineering the Kendryte K210 bootrom

Using the complete bootrom

```
8800007c 1b 03 10 00      addiw      t1,zero,0x1
88000080 13 13 f3 01      slli       t1,t1,0x1f
88000084 e7 00 03 00      jalr       ra,t1=>SUB_80000000,0x0
```

Reverse engineering the Kendryte K210 bootrom

Using the complete bootrom

```
8800007c 1b 03 10 00    addiw   t1,zero,0x1
88000080 13 13 f3 01    slli    t1,t1,0x1f
88000084 e7 00 03 00    jalr    ra,t1=>SUB_80000000,0x0
```

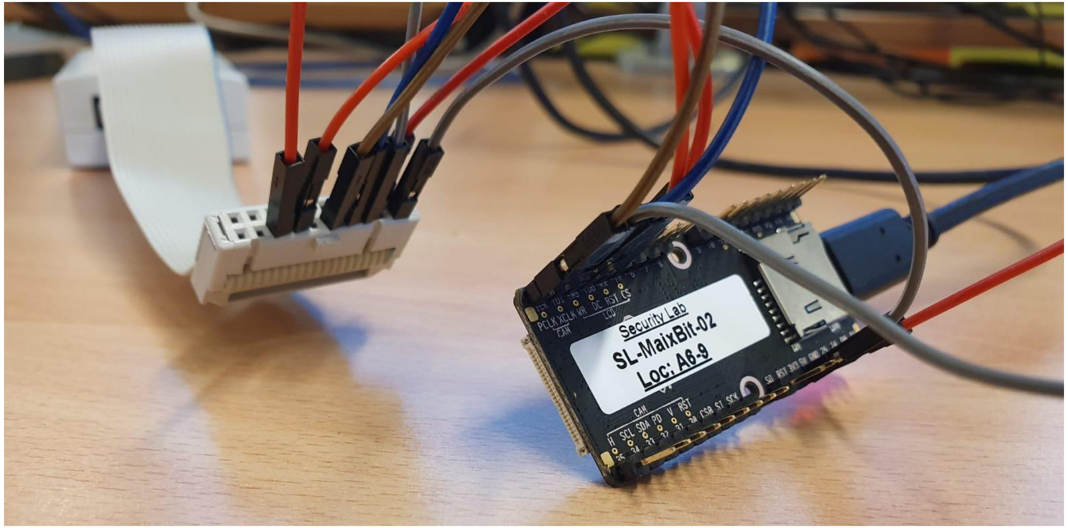
There are still some unrecognized instructions

```
8800b4b8 00 00    c.unimp
8800b4ba 2b       ??      2Bh   +
8800b4bb 50       ??      50h   P
8800b4bc 00       ??      00h
8800b4bd 00       ??      00h
8800b4be 00 00    c.unimp
```

Reverse engineering the Kendryte K210 bootrom

Debugging

Using J-Link and OpenOCD (on-chip-debugger)



Reverse engineering the Kendryte K210 bootrom

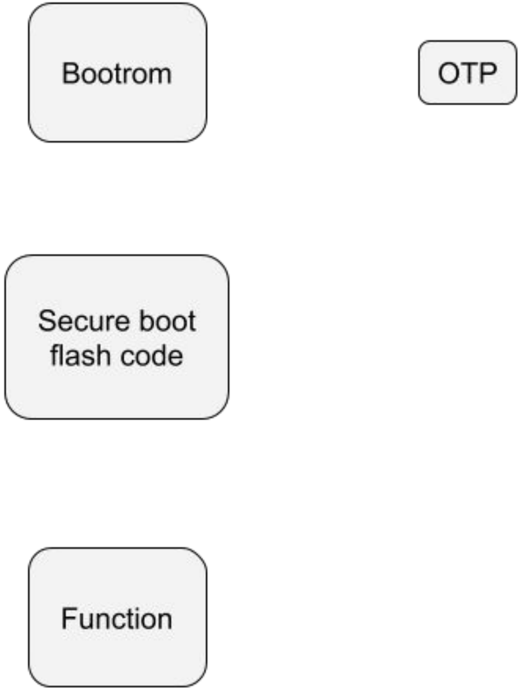
Debugging

It turns out that all instructions left were no actual instructions

```
:kendryte.unimp is RV32 & op0001=0x3 & (op0711=0x0 | op0711=0x1 | op0711=0x1e | op0711=0x1f ) &
 (funct3=0x0 | funct3=0x1 | funct3=0x3 | funct3=0x5) & op1519=0x0 & (op2024=0x0 | op2024=0x9 | op2024=0xb) & op2531=0x0
{
    trap();
}
```

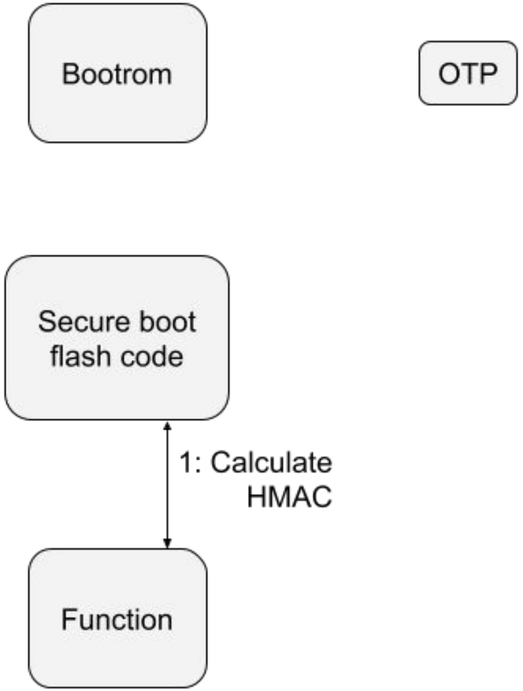
Research into writing to the Kendryte K210 OTP

Implementing secure boot



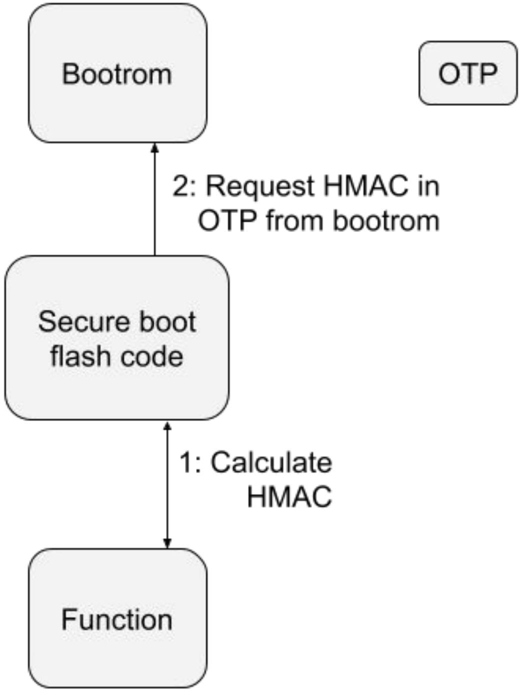
Research into writing to the Kendryte K210 OTP

Implementing secure boot



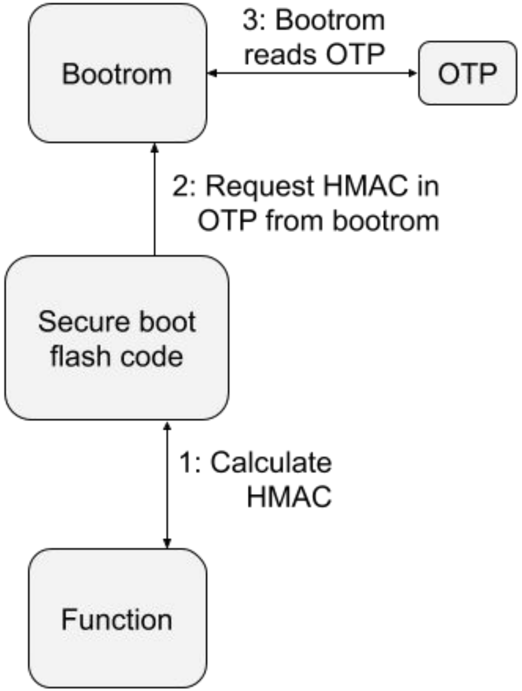
Research into writing to the Kendryte K210 OTP

Implementing secure boot



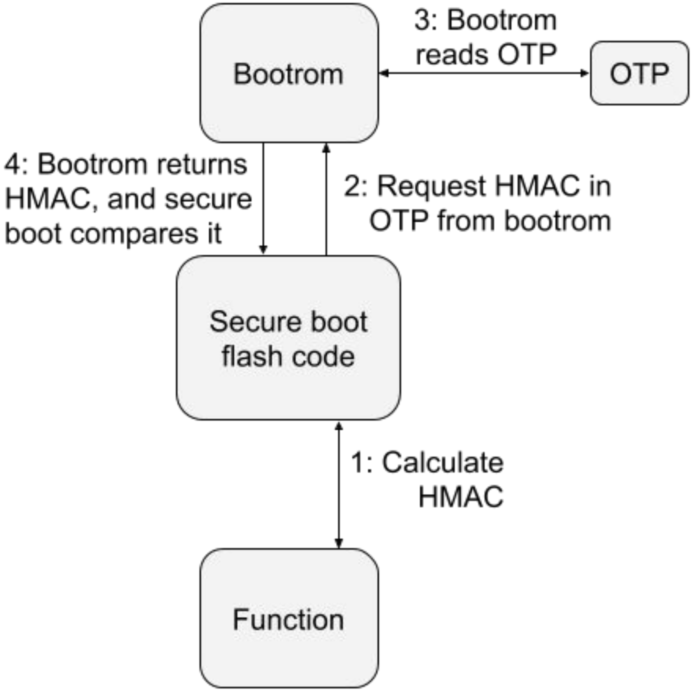
Research into writing to the Kendryte K210 OTP

Implementing secure boot



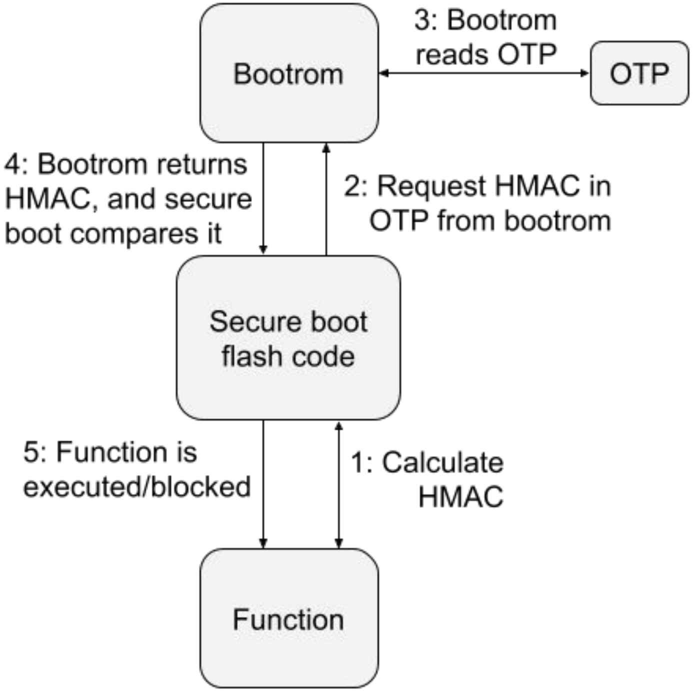
Research into writing to the Kendryte K210 OTP

Implementing secure boot



Research into writing to the Kendryte K210 OTP

Implementing secure boot



Research into writing to the Kendryte K210 OTP

Trying to write to the OTP

We used the Ghidra plugin to find the OTP write function

```
int otp_write(uint32_t offset,uint32_t *data,uint32_t data_length)
{
```

Research into writing to the Kendryte K210 OTP

Trying to write to the OTP

We used the Ghidra plugin to find the OTP write function

```
int otp_write(uint32_t offset,uint32_t *data,uint32_t data_length)
{
```

While being the correct function, it is yet unable to write

Research into writing to the Kendryte K210 OTP

What is this return value?

In the function, the following is specified:

```
if (_DAT_50420060 == 1) {
  return 2;
}
```

Research into writing to the Kendryte K210 OTP

What is this return value?

In the function, the following is specified:

```
if (_DAT_50420060 == 1) {
  return 2;
}
```

So what is this _DAT_50420060?

Conclusion

The Ghidra Plugin works, and is able to completely reverse engineer the Kendryte K210 Bootrom

However, it is not possible to enable any features that require writing to the OTP if the write disabling bit has been set.

Future Work

Test the write function on a Kendryte K210 chip with an unwritten OTP

Use the Ghidra Plugin as a means to analyze the security of embedded SoC's

Enable other features of the Kendryte K210 using reverse engineering

Create a plugin for other RISC-V types or extensions