driving your security forward

Analyzing embedded software technologies on RISC-V64 using Ghidra

Presented by: Joris Jonkers Both & Patrick Spaans
Supervisor: Alexandru Geana
RISC-V64
- Like ARM but open source
- One base image
- Extendable with extensions (e.g. M for multiplications)
RISC-V64
- Like ARM but open source
- One base image
- Extendable with extensions (e.g. M for multiplications)

Security of embedded systems
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
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

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

<table>
<thead>
<tr>
<th>Base</th>
<th>Version</th>
<th>Frozen?</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32I</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>RV32E</td>
<td>1.9</td>
<td>N</td>
</tr>
<tr>
<td>RV64I</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>RV128I</td>
<td>1.7</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extension</th>
<th>Version</th>
<th>Frozen?</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>A</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>F</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>D</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>Q</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>L</td>
<td>0.0</td>
<td>N</td>
</tr>
<tr>
<td>C</td>
<td>2.0</td>
<td>Y</td>
</tr>
<tr>
<td>B</td>
<td>0.0</td>
<td>N</td>
</tr>
<tr>
<td>J</td>
<td>0.0</td>
<td>N</td>
</tr>
<tr>
<td>T</td>
<td>0.0</td>
<td>N</td>
</tr>
<tr>
<td>P</td>
<td>0.1</td>
<td>N</td>
</tr>
<tr>
<td>V</td>
<td>0.2</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>1.1</td>
<td>N</td>
</tr>
</tbody>
</table>
RISC-V64

Supported extensions

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...
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

```c
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

```xml
<ldefs file="riscv.lp64d.sla" language="RISCV"
      version="1.0"
      endian="little"
      size="64"
      variant="RV64GC"
      processor_spec="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"/>
</ldefs>
```
Creating a Ghidra Plugin

Plugin structure

.ldefs file
(language definition)

.sla file
(Instruction definitions)

Example:
Creating a Ghidra Plugin

Plugin structure

.ldefs file
(language definition)

.sla file
(Instruction definitions)

.pspec file
(Processor specification)

```xml
<?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)
Creating a Ghidra Plugin

Plugin structure

- **.ldefs file**
  (language definition)

- **.sla file**
  (Instruction definitions)

- **.pspec file**
  (Processor specification)

- **.cspec file**
  (Compiler specification)
Reverse engineering the Kendryte K210 bootrom

Using the plugin
Reverse engineering the Kendryte K210 bootrom
Using the plugin

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.

<table>
<thead>
<tr>
<th>Ghidra</th>
<th>Radare2</th>
</tr>
</thead>
<tbody>
<tr>
<td>880000066 73 00 50 10 wfi</td>
<td>0x88000062 f36744307300 xor byte [ebx], r14b</td>
</tr>
<tr>
<td>88000006a f3 27 40 34 csrrs</td>
<td>0x88000068 50 push rax</td>
</tr>
<tr>
<td>88000006e 93 f7 87 00 andi</td>
<td>0x88000069 10f3 adc bl, dh</td>
</tr>
<tr>
<td>880000072 e3 8a 07 fe beq</td>
<td>0x8800006b 27 invalid</td>
</tr>
<tr>
<td>880000076 1b 03 10 00 addiw</td>
<td>0x8800006c 403493 xor al, 0x93</td>
</tr>
<tr>
<td>88000007a f3</td>
<td>0x8800006f f78700e38a07. test dword [rdi + 0x78ae300], 0x10031bfe</td>
</tr>
<tr>
<td>88000007b 01</td>
<td>0x88000079 00f3 add bl, dh</td>
</tr>
<tr>
<td>88000007c e7</td>
<td>0x8800007b 01e7 add edi, esp</td>
</tr>
<tr>
<td>88000007d 00</td>
<td>0x8800007d 0003 add byte [rbx], al</td>
</tr>
<tr>
<td>88000007e 03</td>
<td>0x8800007f 006f00 add byte [rdi], ch</td>
</tr>
<tr>
<td>88000007f 00</td>
<td></td>
</tr>
<tr>
<td>880000080 6f</td>
<td></td>
</tr>
<tr>
<td>880000081 00</td>
<td></td>
</tr>
</tbody>
</table>
Reverse engineering the Kendryte K210 bootrom

Using the complete bootrom

8800007c 1b 03 10 00  addiw  tl, zero, 0xl
88000080 13 13 f3 01  slli  tl, tl, 0x1f
88000084 e7 00 03 00  jalr  ra, tl=>SUB_80000000, 0x0
Reverse engineering the Kendryte K210 bootrom

Using the complete bootrom

There are still some unrecognized instructions
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

```c
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

Diagram:

- Bootrom
- OTP
- Secure boot flash code
- Function
  - 1: Calculate HMAC
Research into writing to the Kendryte K210 OTP
Implementing secure boot

1: Calculate HMAC

2: Request HMAC in OTP from bootrom

Function

Secure boot flash code

Bootrom

OTP
Research into writing to the Kendryte K210 OTP

Implementing secure boot

1: Calculate HMAC

2: Request HMAC in OTP from bootrom

3: Bootrom reads OTP

Bootrom

Secure boot flash code

Function

OTP
Research into writing to the Kendryte K210 OTP

Implementing secure boot

1: Calculate HMAC

2: Request HMAC in OTP from bootrom

3: Bootrom reads OTP

4: Bootrom returns HMAC, and secure boot compares it

Function

Secure boot flash code

Bootrom

OTP
Research into writing to the Kendryte K210 OTP

Implementing secure boot

Diagram:

1: Calculate HMAC

2: Request HMAC in OTP from bootrom

3: Bootrom reads OTP

4: Bootrom returns HMAC, and secure boot compares it

5: Function is executed/blocked

Function

Secure boot flash code

Bootrom
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

```c
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

```c
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:

```c
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:

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

So what is this _DAT_50420060?
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.
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