University of Amsterdam

Report

Analyzing and enhancing embedded software technologies on RISC-V64 using the Ghidra framework

February 9, 2020

Supervisor:
Alexandru Geana

Lecturer:
Cees de Laat

Course:
Research Project 1

Course code:
53841REP6Y

Students:
Joris Jonkers Both
11045116

Patrick Spaans
11268476

Abstract

RISC-V, short for reduced instruction set computer - five, is an instruction set architecture (ISA) commonly used for embedded systems and an alternative to the Acorn RISC Machine (ARM) ISA. For embedded systems, security requires special consideration, and disassembly and decompilation tools for RISC-V would therefore be useful to analyze the state of security of such devices. At the moment of writing, RISC-V lacks such reliable tooling, making it more difficult to analyze the state of security of embedded systems using this architecture. The open-source Ghidra software reverse engineering framework can be used to reverse engineer code, but does not completely support RISC-V. In this paper, we present the development approach for a Ghidra plugin which adds the required features to analyze RISC-V assembly. We also discuss how to use the plugin for analysis of the Kendryte K210, a 64-bit chip based on the RISC-V architecture. Documentation for the chip does not describe how to make use of certain features, such as reading and writing from the internal one-time-programmable (OTP) memory. This functionality can potentially be reverse engineered with the help of the Ghidra plugin, and used to implement secure boot for the chip. We used the Ghidra plugin to reverse engineer and analyze the Kendryte K210 chip, but were unable to fully access the OTP memory. This was due to a restriction set in the OTP memory on the chip, which prevented us from writing to it. We were unable to circumvent this restriction in the limited time for the project.

Acknowledgement

We would like to thank the company Riscure B.V. located in Delft, the Netherlands, and in particular our supervisor Alexandru Geana for their help and support during this research. They helped us come up with new ideas for our project, learned us amazing things about debugging and reverse engineering and gave us great feedback.

1 Introduction

RISC-V, which stands for reduced instruction set computer - five, is an open-source hardware instruction set architecture (ISA) targeted towards hardware designs which require efficient processing [1]. It is the successor of the old RISC ISA and is built with modularity in mind. The RISC-V base ISA is a small and minimalistic ISA that can be expanded with RISC-V extensions. This way, a hardware designer can, for example, compile the ISA that suits the needs of the project without having to deal with an overhead of unused functionalities. RISC-V comes in two main versions: a 32- and a 64-bit version (RV32I and RV64I respectively). One example of hardware running RISC-V64, is the Kendryte K210 System on a Chip (SoC) [2]. This chip is meant for internet of things (IoT) purposes and can therefore be applied in many real world scenarios.

Back in 2014, Asanović and Patterson proposed that the community should rally behind RISC-V to test if a free, open ISA would work, as RISC-V is the only open source ISA that fulfills all their mentioned requirements for a good ISA[3]. They envision RISC-V to become the standard ISA in a time where most electronic products use SoCs. As RISC-V has the potential to become the standard ISA, methods to analyze the security of SoCs using it would be needed. At the moment of writing, there is a lack of reliable tooling for RISC-V, making it hard to reverse engineer RISC-V code and analyze its security. Therefore, one of the goals of this research is to create such tooling.

One method to disassamble and reverse engineer software packages is by the use of the Ghidra software reverse engineer framework[4]. Ghidra is a software package that was originally developed by the United States (US) National Security Agency (NSA). This project was US government classified and not open to the public until it was released and made open-source on April 4, 2019. Until then, the only publicly available disassembly tool were Interactive Disassembler Pro (IDA Pro), Radare and Binary Ninja [5][6][7]. While IDA Pro and Binary Ninja manage to suffice most companies needs, they are closed source and require the purchase of licenses to be used. Radare and Ghidra on the other hand, are free, open-source and can be used for roughly the same purposes. Unlike Radare, Ghidra features a structured approach of adding new ISAs, and additionally contains a decompiler which leads to a more convenient reverse engineering process.

While all of these solutions offer similar features, none of these offer full support for the RISC-V64 platform, with the Ghidra and Radare implementations for RISCV-64 not being completed as of writing. This causes reverse engineering RISC-V64 using architectures to be more difficult than needed, which negatively impacts the ability to analyze the security of embedded systems using this architecture. In addition, a reverse engineering tool could be used to discover what features are contained in a chip, as the Kendryte K210 chip for example theoretically supports features not enabled by default. According to the official documentation, this includes features such as a so-called ”turbo mode” that increases the clockspeed.[8] In addition, the chip features a one-time-programmable memory (OTP memory) that could theoretically be used to implement a secure boot feature. A reverse engineering tool for RISC-V64 could be used to discover how these features are or could be implemented, and might allow for the enhancing of chips such as the Kendryte K210.

This leads us to the main question this research will try to find the answer:
In what ways can a disassembly and decompilation tool be used to analyze and enhance the working of embedded technologies?

Which can be divided into the following subquestions:

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

1.1 Related work

There has not been done any research to how RISC-V64 can be implemented for Ghidra. One reason as to why this may be the case is the fact that Ghidra has only been open source for eight months at the time of writing, and hence there has not been much time to work on an Ghidra extension.

As for the general idea for this research itself, using reverse-engineering of code in order to analyze security is not a new concept. Udupa et al. released a paper back in 2005 in which they talk about using reverse-engineering as an analysis method for security.[9] Their research went indept on a multitude of techniques that could be used to deobfuscate code, in order to simplify the reverse engineering process. Deobfuscation is not directly relevant to this research, as there is no reason to suspect that the Kendryte K210 makes use of deobfuscation, however the paper does bring up the point that not much research has been done regarding reverse engineering back in 2005.

The concept to reverse engineer to analyze security is however even older than that. Back in the year 2000, Devanbu et al. released a paper back in 2000 discussing the history of software engineering for security, where it can be read that the need for reverse engineering tools to analyze security was not a new concept even back then.[10]

More closely related to this research, Zaddach and Costin published a paper in 2013 on how the security of the firmware of embedded devices can be analyzed through reverse engineering.[11] They discuss the current state of security and the challenges they face, as well as the tools available to analyze the security. They also discuss the use of firmware emulation for easier and faster vulnerability discovery. In this paper, RISC-V was not discussed, as it was still a fairly new ISA at that point, with it only having been released in 2010. [3]

Regarding the Kendryte K210, whereas no official research into its workings have been conducted before, GitHub user LaanWJ has conducted his own research into some use cases for the Kendryte K210, and has been creating more documentation about the specifications of the chip.[12] While this research is neither official nor complete, this information could certainly be used as reference material. In addition, LaanWJ documented some of the functions in the bootrom, as well as created a code to read the values stored in the OTP of the chip.

2 Methodology

This project consists of three parts, first a plugin will need to be created for Ghidra, which allows for the disassembly and decompilation of RISC-V code. A concept version of this already exists for Ghidra, but as of writing is not fully operational. This code will be used as basis for the plugin. The second part of this project is to use this plugin to reverse engineer the bootrom of the Kendryte K210, a hardware and programming environment using RISC-V64. Lastly, the reverse engineered bootrom will be used to enable features supported by the Kendryte K210 but not programmed in by default, such as secure boot. The functions that can be used to implement this will be found in the reverse engineered bootrom. In order to access the chip and enable these functions, a Sipeed MAIX-BiT development board was used, which is a board that contains the Kendryte K210 chip.

2.1 Implementing a Ghidra plugin for RISC-V64GC

For the first part of the project, the Ghidra plugin for RISC-V support will be build from the ground up. First, a plugin will be created for RISC-V32, which is the 32-bit version of RISC-V. RISC-V64 is an extension for use with 64 bit registers, and therefore the simplest method to implement it is by creating a plugin for RISC-V32 first, and later alter it to support 64-bit. The plugin will be based around the RISC-V instruction set manual, found on the official RISC-V GitHub page.[**riscv-GitHub**] After creating this plugin, plugins will also be created for the extensions of RISC-V, namely the extensions 'G' and 'C'. Extension 'G' consists of a collection of numerous extentions. In table 1, all extensions and their definitions can be seen.

Extension	**Definition**
I	Integer computational instructions
M	Integer multiplication and division instructions
A	Atomic read/modify/write memory instructions
F	Floating point instructions
D	Double precision floating point instructions
Zicsr	Control and status register instructions
Zifencei	Instruction fetch fence instructions
C	Compressed instructions

Table 1: The RISC-V extensions relevant for this project.

After RISC-V32GC is implemented, the code is altered in order to support 64-bits. First just the RISC-V32 plugin is altered to create the RISC-V64 plugin, and afterwards the extensions are modified as well in order to get the final product, RISC-V64GC. After every of these creation steps, each plugin is tested on its functionality and correctness, by compiling our own code in RISC-V and disassembling it using the plugin. This will also be compared to the disassembly using the current concept version of the Ghidra version from the GitHub repository.

To create a plugin, the preliminary RISC-V plugin which is available on the Ghidra GitHub repository will be used as basis. From there on, the bootrom is decompiled in Ghidra and missing instructions are analyzed and added to the plugin. The analysis of an instruction will be done in three steps: the conversion of hexadecimal values as shown in Ghidra to the binary equivalent, comparison of the instruction data with the official RISC-V documentation, implementation of the instruction in the plugin.

Since the RISC-V architecture is little endian, the conversion of hexadecimal values has to occur accordingly. The hexadecimal bytes are first reversed in order and then converted into binary values. For example, the byte combination 'b3 e7 f6 00' would in this case be converted into '0000 0000 1111 0110 1110 0111 1011 0011'. If we then compare this binary string with the definition that is defined in the RISC-V documentation, we find that this string matches the requirements for an 'OR' instruction. According to the documentation, an instruction is considered an 'OR' instruction if the first seven bits (when reading right to left) are set to '0110011' and the last seven bits are all set to zero. As one can tell from the binary conversion above, this instruction clearly is an 'OR' instruction. Now it is clear that this is an 'OR' instruction, we can start defining this instruction for Ghidra. Ghidra makes use of the Sleigh language that is based on SLED [13]. As such, we need to define the instruction in this language. First, a colon with the name of the instruction behind it will have to be specified, followed by the required opcodes that identify such an instruction. Finally, the C code equivalent of the operation is specified between curly braces. A complete instruction will look like this:

```
:and rd,rs1,rs2 is RV32 & RVI & rs1 & rs2 &
rd & op0001=0x3 & op0204=0x4 & op0506=0x1 &
funct3=0x7 & funct7=0x0
{
 rd = rs1 & rs2;
}
```

In this example, the 'AND' instruction is implemented. First, the name of the instruction is specified, followed by the necessary parameters (rd, rs1 and rs2). Then the instruction characteristics are specified. The bit mode is specified (RV32 for 32 bit mode). Then all opcodes are defined. In this case, the opcode in the bits 0 to 1 needs to have the value 0x3, the opcode in bits 2 to 4 needs to be 0x4 etc., before Ghidra will consider a binary instruction an 'AND' instruction. To complete the plugin, all currently missing instructions in the plugin will have to be implemented using this method.

2.2 Using the Ghidra plugin to reverse engineer the bootrom

After the plugin has been created, it will be used to reverse engineer the entire bootrom. At this point, all instructions used in the bootrom that are also listed in the RISC-V manual should be recognized automatically by Ghidra. All bits not recognized as an instruction are either data, such as a string, or instructions exclusive to the Kendryte K210 and therefore not part of the general manual.

Out of these two, data is the easier of the two to reverse engineer, as Ghidra can recognize certain byte combinations as ASCII code, making it easy for the person using the tool to recognize it as data while parsing through the data file by hand. Another Ghidra functionality that simplifies this process is the 'find next unknown' function, which searches the file for unrecognized bytes. While this tool is quite helpful for finding all unrecognized data bits, it still requires manual labor. In addition, the Ghidra user should remember that Ghidra can recognize strings or subsets of strings as instructions, meaning that not all recognized instructions necessarily are functions. This can easily be solved by going over the file manually.

As for the other type of not recognized bits, the exclusive instructions, one method to gather more information about these instructions is debugging. A debugger such as the Open On-Chip Debugger (OpenOCD) can be used to debug the MAIX-BiT board. While OpenOCD does not support the Kendryte K210 chip by default, an alternative version that does support it can freely be found on the internet [**openocdGitHub**]. OpenOCD requires a hardware debugger that is connected to the pins of the MAIX-BiT board as well as to the USB port of the computer executing OpenOCD to function. While any JTAG device would suffice for this, the recommended hardware debugger is a JLINK device as the OpenOCD version that supports the Kendryte K210 supports the JLINK device by default. In figure 1, the setup of a MAIX-BiT, a JLINK device and their respective pinouts can be seen.

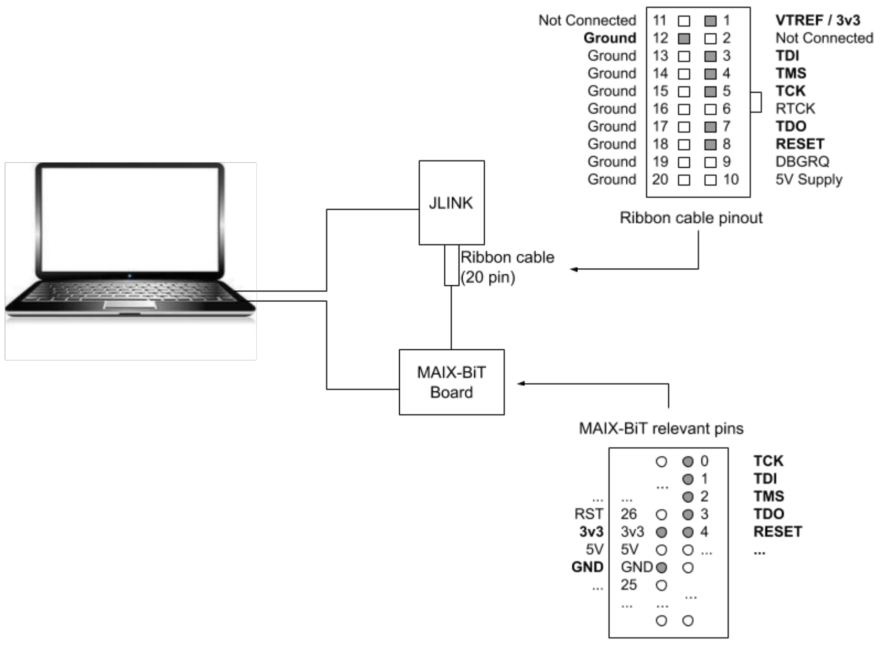

Figure 1: The debugging setup, including the pinout. The pins that need to be connected are listed in bold.

Using the OpenOCD debugger, it is possible to alter the program counter of the MAIX-BiT device such that it is set to the address of one of the unrecognized bits. In addition, all other registers of the device can be set to different values as well. Then, the unrecognized bits will be executed, and the effects of the instructions can be noted. As for what the effects of such an instruction might be, there are three outcomes that can happen. First, one of the registers can have a different value, meaning that the instruction interacted with the registers. For this reason, it is important to execute these unrecognized bits with registers having non-zero values. As an example, an addition between two registers with the value of zero cannot be differentiated from no addition happening at all. The second outcome can be that the function is a jump instruction. If this is the case, the next program counter after executing these bits contains a value that is not equal to the index of the first instruction after the unrecognized bits. Lastly, the unrecognized bits can have no purpose at all, at which point neither of the aforementioned possible outcomes have taken place.

With the purpose of the unrecognized bits discovered, the Ghidra plugin can be completed for the Kendryte K210, allowing it to be completely able to reverse engineer the bootrom of the chip. At this point, a manual inspection of the completeness and correctness of the reverse engineered bootrom will take place. The completeness can be affirmed by searching for unrecognized instructions in the bootrom, while the latter of which done through using the Ghidra tool that transforms the assembly code into C code. If none of the reverse engineered C code contains 'bad data' warnings, it is most likely correct as incorrectly recognized instructions or data would cause such a warning to appear.

2.3 Enabling new features in the Kendryte K210

The reverse engineered bootrom can then be used for the final part of this research, enhancing the MAIX-BiT board and it's Kendryte K210 chip to support features not enabled by default. One such feature is secure boot, which is not implemented by default. It is not officially supported, and therefore there is no bit in the Kendryte K210 that can be set to enable it. However, the chip does include the means necessary to create a custom made secure boot implementation. The Kendryte K210 supports OTP, and an OTP can be used to implement secure boot.[14]

Secure boot can be programmed into the Kendryte K210 due to the fact that the chip contains OTP memory. This is memory that can be read to and written by the bootrom of the chip, and once a bit in the memory has been set it can never be unset again, which explains the name. The secure boot will be implemented as follows: a HMAC hash will be calculated of the code we want to boot securely, and this same hash will be stored into the OTP memory. At every boot of the chip, before this code is executed, the hash will be recalculated and confirmed with the hash in the OTP. Figure 2 shows this implementation.

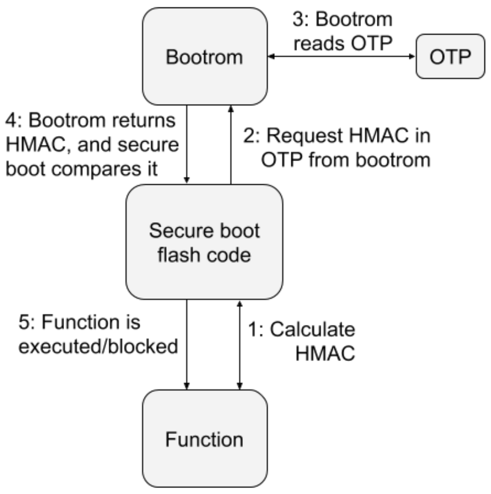
Bootrom
3: Bootrom reads OTP
OTP
4: Bootrom returns HMAC, and secure boot compares it
2: Request HMAC in OTP from bootrom
Secure boot flash code
5: Function is executed/blocked
1: Calculate HMAC
Function

Figure 2: The proposed secure boot implementation on the Kendryte K210.

Multiple components are required to implement this secure boot implementation. For the code that needs to be booted securily, a simple hello world program will be created. This program will be created using PlatformIO, as it is one of the officially supported programming platforms for the MAIX-BiT board [15]. The main code that handles the secure boot will create a HMAC hash of all bits of the hello world program stored in the flash memory of the MAIX-BiT device. The location of these program bits can be found in the .elf or .bin file of the code that will be uploaded to the MAIX-BiT board. One difficulty about generating this HMAC value or obtaining these bits is the fact that the MAIX-BiT device only supports RISC-V64GC instructions, meaning that using prebuild functions to do this might cause difficulty.

In order to compare the calculated hash of the program with the original, the original HMAC value needs to be stored in the OTP memory first. This requires two of the functions listed in the bootrom, namely a function that can read from the OTP and a function that can write to the OTP. The OTP memory is only accessible to the bootrom, and anyone trying to read the OTP memory without using a bootrom function will obtain all ones as a result.

An OTP read function can be found in the reverse engineered bootrom, and prebuild C code on how to use this function to print the OTP can be found on the GitHub repository of LaanWJ.[12] As LaanWJ's code printed out the OTP in Intel hex values, it can only serve as basis and not be used directly. The read function requires three inputs, namely the offset in the OTP of the data that it is trying to read, a pointer to a variable or buffer to store the value in, and the amount of bytes to be read. Using this function for every offset in the OTP, the entire OTP can be obtained.

If space in the OTP is found to store HMAC values in, these hashes can be written to by using a bootrom function that can write to the OTP. This function can be found using the reverse engineered bootrom, and is a function similar to the OTP read function, as it requires the same parameters. Whether such a write function is actually able to be used depends on two factors. First, a write protecting bit must not be set in the OTP, as that would prevent the function from activating. Secondly, the Kendryte K210 chip should not require additional voltage in order to enable OTP writing. Some chips require this as a way to make it harder to write to the OTP[16], however no such information is listed in the documentation of the Kendryte K210.

3 Results

As discussed before, in the first part of this research, the possibility of creating a Ghidra plugin for RISC-V64GC was researched. It turned out that the NSA had already a preliminary version of a RISC-V plugin available on their GitHub repository[**ghidraGitHub**]. This plugin, however, was incomplete and in some cases even incorrect. Some of the plugin functions were incorrect in the sense that the requirements for some of the instructions were not specified according to the official RISC-V documentation. One example of an instruction that turned out to be incorrect is the fence instruction. This instruction was defined on the Ghidra GitHub repository like the code below:

```
:fence pred,succ is RV32 & RVI & pred & succ & op0001=0x3 &
op0204=0x3 & op0506=0x0 &
funct3=0x0 & fm=0x0 & op0711=0x0 & op1519=0x0
{
}
```

Compared to the official RISC-V specification, the above defined definition is more strict than it should. The official RISC-V documentation only specifies that the only requirement for a fence instruction is that the instruction should start with the opcode '0001111'. The Ghidra implementation however, specified that also bits at different positions later in the instruction should be zero. In practice, this showed to be an issue in cases where these positions were not filled with zeros. In those cases, Ghidra was unable to identify the instructions as FENCE instructions and aborted. By removing these unnecessary constraints the problem was solved.

Then, the instructions that were still unrecognizable by Ghidra were inspected and implemented. These instructions turned out to be: c.slli.hint, c.lui.res, c.addi4spn.res, fence.i and kendryte.unimp. This last instruction was added due to decompile the Kendryte K210's specific instructions that were not present in the official RISC-V documentation.

The kendryte.unimp instruction was defined by analyzing all occurrences of missing instructions and noting what characteristics they have in common. The instruction that resulted from this analysis is:

```
: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();
}
```

Once this Kendryte specific instruction was added to the list of supported instructions by the plugin, the bootrom could be decompiled completely.

During this research, it turned out that the bootrom obtained through simply printing all bootrom addresses was in fact incomplete, meaning that a different method of obtaining the bootrom was required. One way to require a fully complete bootrom would be through the use of OpenOCD, which allows for the dumping of data in between certain indexes of the board.

As for implementing a secure boot feature, it was impossible to implement it as the bootrom function that allows for writing to the OTP memory returns an error code. This error code was caused due to a bit set in the OTP memory that prevents writing. While we were unable to write to the OTP memory, unprogrammed space in the memory could be found, as it contains a large section for program data which is mostly unwritten to. In boards using the Kendryte K210 that do not make use of the full program data and do not have a write protection bit set, this space can be used to store HMAC values in.

4 Discussion

During this research, it turned out the bootrom obtained through printing out the values at each bootrom address was incomplete. The bootrom obtained through this method contained nearly two percent less bits than the complete bootrom. All missing bits were located at random locations in the code, and were always entire hexadecimals of bits that went missing. As the print function used printed out the bits as hexadecimals, the issue was most likely caused by the print function. While the cause of this issue is not yet known, it is expected that it is caused by the printbuffer being full while trying to read the bootrom at high speed. This causes it to miss a few hexadecimals every so often. Using OpenOCD to obtain the bootrom is a more effective method, allowing you to obtain the full bootrom. One thing to keep in mind is that the bootrom needs to be of a certain size, as the amount of addresses you try to print should match up with the amount of hexadecimals in the resulting bootrom.

While the reading function to read from the OTP worked as intended, it is uncertain whether the available space inside the OTP is usable or not. While most of the OTP seems unused, the chip might need these values to be set to 0, and could crash when altered. The space we selected for the HMAC seems to be unused however, as it seems to be reversed for program data. In one of the first fields of the OTP, a value is specified which is the exact same length as the used program data that follows it. This suggests that the space directly after this program data is unused, since if it were used for different purposes no length indication would be required for the program data.

In this research, we were unable to write to the OTP due to a write disabling bit that was set in the OTP. Even if this bit would be unset, it might still be the case that additional voltage might be required in order to write to the OTP. If this were to be the case, it would make things more complicated as there is no documentation on what pins on the Kendryte K210 the voltage needs to be applied to, or what voltage it would need to be.

5 Conclusion

In this research a Ghidra plugin to decompile and disassemble RISC-V64GC code was created succesfully. To do this, all incorrect instruction definitions in the already existing RISC-V plugin for Ghidra were fixed. In these incorrectly defined instructions, the unnecessary constraints were removed. On top of that, all missing instruction definitions were added. The missing instructions were first identified as such and their hexadecimal equivalent converted into raw binary code. Then, they were compared with the RISC-V documentation to find out what kind of instruction it was. To turn this match into a definition in the Ghidra plugin, the specification in the RISC-V documentation was rewritten in for Ghidra understandable Sleigh code and compiled. However, there was a number of instructions that did not match any instruction in the RISC-V documentation. To find out what kind of instructions these were, a hardware debugger was connected to the MAIX-BiT board. This showed that the missing instructions all had one thing in common: when executed they resulted in an infinite loop. It became clear that these instructions were causing the same action as a regular unimp instruction would have. A new instruction was then added to the plugin called 'kendryte.unimp', specifying the same action as the unimp instruction, to define this kind of instruction. This way, it was possible to create a RISC-V64GC plugin for Ghidra that was capable of disassembling and decompiling the Kendryte K210's bootrom code. It also showed that using the plugin in Ghidra it is possible to succesfully reverse engineer the Kendryte K210 bootrom code into human readable C code. The plugin could therefore be used for further analysis of such bootrom code.

As for writing the OTP, the functions in this completed bootrom could be used to read and write to the OTP. But it turned out to be impossible to write to the OTP on the default MAIX-BiT board. In the function that seems to be the OTP write function, an if statement is included that checks for a certain bit in the OTP. If this bit were to be set, the code would return with error code two and would not perform the write operation, which makes it impossible to use the write function. As this bit is set inside the OTP memory, there is no possible way to unset it, making it impossible to write to the OTP memory on a MAIX-BiT board.

6 Future work

In this research we tried to write data to the OTP area of the Kendryte K210 chip. While this area should have been writable from the factory, we were not able to do so. This was probably because of the bits that were set by Sipeeed, the manufacturer of the Maix-bit board the chip was soldered on to. These bits caused the software to go into an infinite loop when trying to write to the OTP memory. In a next research, one could try to use a chip from a different manufacturer or at least a chip that does not have these particular bits set.

Even if a board that features the Kendryte K210 chip were found that has not set the OTP write protection bit, it might still be the case that writing to the OTP memory is impossible. No research has been done to confirm or deny that the Kendryte K210 requires additional voltage in order to be written to.

Another thing that could be researched in a future research is how useful the created plugin is in analyzing the security of the so-called SoCs. Now the plugin is ready, it should be possible to analyze the security of all functions that are present on the chip and an overview could be made of how well the security in SoCs is implemented.

While doing this research, it became clear that the researched Kendryte K210 chip offers a lot of features such as a turbo mode that would double the clock frequency of the chip. Most of features were, however, disabled by the manufacturer and enabling them would require us to write to the OTP memory of the chip, which turned out to be impossible. In a future research, one could research what kind of performance or abilities these extra features could unlock. This may also give a better insight in how this chip could be implemented as effective as possible.

Last but not least, the created plugin could be extended to work with all other available RISC-V extensions. Currently the created Ghidra plugin only supports the extensions G and C. A future research could further investigate how the current version of the plugin could be extended to support all other extensions in all supported bit modes (i.e. make the plugin work for RISC-V32, 64 and 128).

References

[1] Andrew Shell Waterman. "Design of the RISC-V instruction set architecture". PhD thesis. UC Berkeley, 2016.

[2] Inc. Canaan. *Kendryte Homepage*. URL: `https://kendryte.com/` (visited on 01/09/2020).

[3] Krste Asanović and David A Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[4] U.S. National Security Agency. *Ghidra Homepage*. URL: `https://www.nsa.gov/resources/everyone/ghidra/` (visited on 01/07/2020).

[5] Hex-Rays. *IDA: About*. URL: `https://www.hex-rays.com/products/ida/` (visited on 01/08/2020).

[6] Radare. *Radare homepage*. URL: `https://www.radare.org/r/` (visited on 01/10/2020).

[7] inc. Vector 35. *Binary Ninja homepage*. URL: `https://binary.ninja/` (visited on 01/10/2020).

[8] Canaan Inc. *K210 Datasheet*. 2018.

[9] Sharath K Udupa, Saumya K Debray, and Matias Madou. "Deobfuscation: Reverse engineering obfuscated code". In: *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE. 2005, 10–pp.

[10] Premkumar T Devanbu and Stuart Stubblebine. "Software engineering for security: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 227–239.

[11] Jonas Zaddach and Andrei Costin. "Embedded devices security and firmware reverse engineering". In: *Black-Hat USA* (2013).

[12] LaanWJ. *Kendryte K210 / MaixGo stuff*. URL: `https://github.com/laanwj/k210-sdk-stuff` (visited on 02/06/2020).

[13] NationalSecurityAgency. *Sleigh documentation*. URL: `https://ghidra.re/courses/languages/html/sleigh.html` (visited on 02/08/2020).

[14] Dong-jin Park, Myung-Hee Kang, and Won-Churl Jang. *Secure Boot Method and Method for Generating a Secure Boot Image*. US Patent App. 13/279,512. August 2012.

[15] Sipeed. *Sipeed MAIX*. URL: `https://www.seeedstudio.com/sipeed-maix.html` (visited on 02/09/2020).

[16] Silicon Laboratories Inc. Evan Schulz. *Using Low-Cost OTP Microcontrollers to Simplify Embedded Applications*.