DATA CENTER SYSTEM LABORATORY
MASTER SEMESTER PROJECT

Mirage : firmware virtualization

Abel VEXINA WILKINSON

Supervised by :
Charly CASTES
Neelu SHIVPRAKASH KALANI

8 June 2022

Contents

List of Figures

Abstract

While Operating Systems have been virtualized for a long time for security purposes, Firmwares have been mostly ignored. Mirage is a prototype that aims to sandbox untrusted firmware for security purposes. This requires running firmware in an unprivileged manner, without restricting its functionality.

This report aims to explore and explain the basic steps taken to virtualize firmware without any code modifications. It outlines important elements such as emulation of privileged registers and instructions. But also unique trapping mechanisms and non-firmware executions. Additionally, the report introduces the concept of security into the Mirage prototype via the use of special registers.

1 Introduction

What do we trust in computers? Where does trust begin? What should we not trust? All of these questions apply to modern day computing. Security takes a more important place than ever before.

One of the elements that we trust in all cases is firmware. Firmware takes care of configuring the hardware for it to be available for other software. Firmware runs in Machine mode, which is a privileged operating mode in RISC-V architecture, providing the highest level of control over the processor. It allows access to all system resources and is responsible for managing lower privilege modes (user and supervisor mode), handling interrupts, and configuring hardware settings. This makes firmware ubiquitous to all machines.

Firmware is usually written by the manufacturer of the board, making it a black box. Trusting the firmware implies trusting the manufacturer. Manufacturers use open-source firmware such as OpenSBI, a standard open-source for the RISC-V architecture, and modify them with proprietary components to suit their needs. The real issue lies in manufacturers not releasing the source code for the firmware they use, leaving potential security concerns unaddressed. OpenSBI serves more as a library for building custom firmware tailored to specific hardware, rather than a universal, ready-to-use firmware solution, due to the unique proprietary elements of each board.

Virtualization is a technique that has been used with Operating Systems (OS) and other software to protect from their potentially malicious intents. Sandboxing is a type of virtualization. If it can be done for an OS, then it can probably be done with firmware.

That is the idea behind Mirage.
Instead of replacing the whole firmware with something custom, Mirage aims to virtualize any existing firmware. This can be done by replacing the firmware's place in the structure by Mirage and running the firmware with lower privileges. By virtualizing and

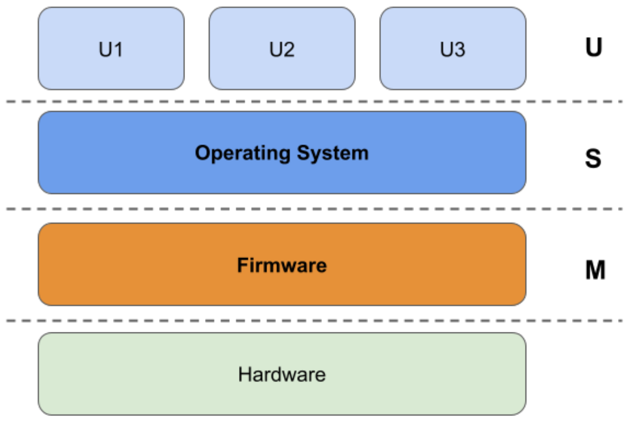

Figure 1: Classic 3 levels privileges structure

emulating registers and instructions of the original software, Mirage can also control its execution. This makes sure that the firmware does not affect the execution aside of its intended purpose.

This project focuses on establishing the basics of firmware virtualization and security. The first objective was to be able to make a general purpose firmware, such as OpenSBI, boot up a system while being emulated. The idea of this is to expand the emulation capabilities later on. The second objective of the project was to make a operating system without emulation possible on top of Mirage and alongside the virtual firmware. The third and last objective was to make sure some security assurance was available to Mirage. All done to start to provide a safe environment to software without the need to trust a firmware.

1.1 Contributions

These are the contributions of the project.

- Finalized M-mode emulation with M-mode CSRs and operations.
- Extended M-mode emulation with support for traps.
- Allowed for S-mode execution.
- Added security for Mirage with PMPs.

1.2 Overview

This report is structured as follows. Firstly, we will explore some necessary concepts and elements for understanding the report and the overall project. Secondly, we will focus on what is needed to emulate a Machine mode software, a "firmware", and what contributions these programs have. Thirdly, we will see what it takes to run a Supervisor mode "payload" that depends on the previously mentioned M-mode "firmware". Fourthly, we will look at what it takes to configure Physical Memory Protection (PMP) registers (and their security capabilities) in this particular context. We will finish the report with a round-up of the main characteristics of the project and a discussion of the work. This report also includes a quick deep dive into the implementations details, this is meant for the people working on the project further down the line.

2 Context

2.1 Theoretical background

A virtual machine (VM) is a software emulation of a physical computer. It runs an operating system and applications just like a physical computer, but it operates in a virtual environment.

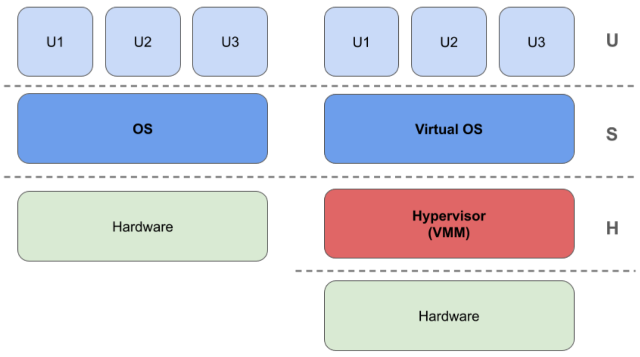

Figure 2: Differences in 2-level and virtual privileges structures

The Popek and Goldberg theorem, formulated in 1974 by Gerald J. Popek and Robert P. Goldberg, provides a set of conditions that a computer architecture must satisfy to support efficient and secure virtualization. A "virtualizable" system would be a system which can run equivalently whether it is running directly in hardware or in a virtual machine. For a system to be considered "virtualizable", all sensitive instructions must be privileged, ensuring that any attempt to execute them in user mode will trap to the VMM, allowing the VMM to manage them appropriately.

In our situation, the virtualized software would be the Firmware running in a lower-privilege mode.

Architectures have different privilege levels of execution. These allow to configure different sections of the execution, but also to abstract away unnecessary elements. Most importantly, they serve a security purpose, software in lower privilege levels can be untrusted. Layering allows those pieces of software to not harm the rest of the system.

In a classic system, three (sometimes more) privilege levels or modes exist. The lowest privilege level is User mode, or U-mode, which can only configure its own execution and not affect other programs.

The next would be Supervisor mode, which can configure elements such as page tables to be used by the rest of the software. It also can configure how U-mode software runs. This is usually the mode in which an Operating System runs.

The last and most privileged is Machine mode, or M-mode. This mode has everything allowed to it. And allows configuring the hardware directly. Firmware usually runs at this level, since it has to configure the machine to run correctly.

Firmware is a specialized piece of software, usually running with the highest privileges. Firmware is usually loaded at boot, and stored in a specific part of memory included with the board. Firmware acts as the intermediary between the device's hardware and its higher-level software, such as the operating system or applications.

2.2 Practical background

Some practical background is also in order. This project focused on some particular combination of architecture, firmware, and platform to keep its scope limited.

The architecture of choice this time has been the RISC-V [1] architecture. As its name implies, this architecture belongs to the "Reduced Instruction Set Computer" (RISC) family of "Instruction Set Architecture" (ISA). The fact that it is a RISC, helps when considering the emulation, because it means that there are fewer instructions to consider and emulate. This reduces the complexity and difficulty of the implementation.

Another nice perk of the RISC-V architecture is that it is open-standard, which makes obtaining information about it much, much easier, and allows the project to move forward faster.

The last, but certainly not least, aspect of this architecture that is crucial to the project, is that it allows for multiple privilege modes of execution. Mainly it has the capacity of running software in Machine, Supervisor, and User mode (as explained before), and has an interesting set of security features such as PMPs (explained in section 5)

The platform of choice is (a bit ironically to my taste) the RISC-V Qemu emulator [3]. We have chosen this option for three main-reasons. Firstly, it allows having access to the platform extremely easy as it makes part of the project installation. This makes testing and running the early stages of Mirage very practical.

Secondly, the configuration of the platform can be changed with ease to test different elements of the firmware (whose execution depends on the platform's features).

Thirdly, the emulator is open-source, allowing to explore its source-code with ease in case of problems or abnormal situations.

The choice of firmware is the easiest to modify. For this project, we have opted for the OpenSBI firmware. As its name indicates, it is an open-source firmware for RISC-V . The open-source characteristic of OpenSBI helps in the testing and debugging of the project. It allows knowing which steps the program will take during its emulation, making it easier to debug and understand during the project. It also becomes a judge of progress for the project, as the further we go into the boot process, the more features are available.

3 M-mode virtualization

3.1 Firmware Emulation

As we have previously stated, the objective of this project is to virtualize the firmware. To achieve this, we need to emulate the behaviour of the firmware. To emulate a behaviour is to copy it with all of its characteristics.

In the project, the basic emulation loop works as follows. For starters, Mirage jumps into the firmware code to execute it. This will start making the operations that the firmware needs to run to configure the system. Except that the firmware will be running in U-mode. As such, privileges to handle the hardware are not given to the firmware execution.

This, causes traps when the firmware needs to execute a privileged instruction (we will see some of these in the next section). This trap is then relayed to the trap handler of Mirage. This makes the execution return to Mirage, and most importantly under our control. At this point, we can emulate the instruction and jump back to the firmware to continue.

You can see in Figure 3 how the new privileges structure works. Mirage is the only software running in M-mode. Making it have exclusive access to the needed privileges.

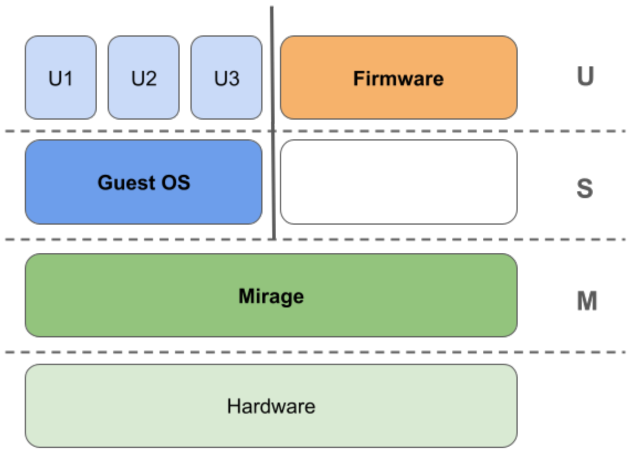

Figure 3: Mirage's privileges structure

One of the key factors of emulation is that only the privileged instructions and operations are emulated. To achieve this in Mirage, we use a "Virtual Context".

This "Virtual Context", contains all the hardware information (and some metadata to help execution) that the firmware should have access to during normal operation. And most importantly, that should otherwise be in hardware.

For instance, there are 32 basic RISC-V registers (x0 to x31) in the virtual context, which are the ones that the firmware operates on. These values must be in hardware while the firmware executes, since they can be accessed without any privileges. In section 3.2 we will see an abundance of example of elements belonging to the virtual context.

To achieve this, jumping and returning from the firmware is not only a simple jump. But also serve as points for context switch logic (in which we will dive deeper later on). During this context switch, the values that are needed in hardware during firmware execution can be copied from the virtual context, and then copied back from hardware into the virtual context when returning to mirage.

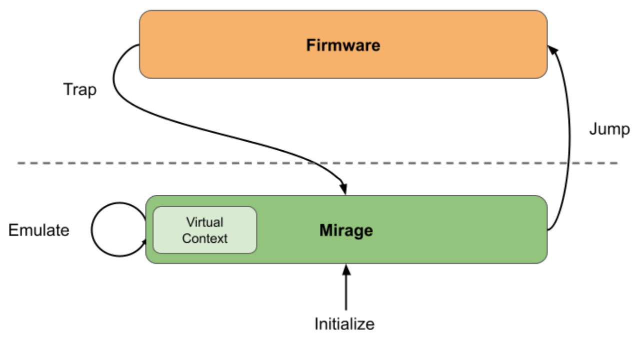

Figure 4: Quick view of firmware emulation

Figure 4 shows a little summary of how the basic firmware emulation works. We first initialize Mirage to configure everything and get the execution of the firmware ready. Then, we jump into the firmware to execute it. Then an instruction will trap because the firmware does not have the necessary privileges. At that point, we emulate the instruction modifying the previously mentioned virtual context. And we finally go back into the firmware to continue execution.

3.2 M-mode CSRs

A Control and Status Register (CSR) is a register that allows to configure and control a certain part of the hardware. Some can also serve to inform of the status of the hardware or the execution. CSRs are divided by privilege mode. Some CSRs are exclusive to M-mode, others can be used by S-mode.

We will now focus on the CSRs that belong to M-mode in RISC-V . We will start by doing a quick round-up of the most common characteristics for CSRs.

Firstly, a CSR can belong to a specific extension. This means that if the extension is not available in the platform, the CSRs that belong to it are not available either. For example, there is the Debug extension, which makes a bunch of debugging CSRs available. Another example is the Hypervisor extension, which enable fields on some CSRs, requires operations to do additional things, but also unlocks some CSRs.

Secondly, a CSR can be Read-only, Write Anything Read Legal (WARL) or Write Anything Read Anything (WARA). A Read-only CSR cannot be written with any instruction, these are usually registers that contain information about the manufacturer of the board, such as the Machine Vendor Id (mvendorid) CSR. If a value is written, the register is not modified. A Write Anything Read Anything CSR is what we would consider a "normal" register. In these, the last value written is the value that will be read. The most complex behaviour comes from the Write Anything Read Legal CSRs. In these registers, the values written get filtered to accommodate only "legal" values. What is considered legal depends on each register. Some require having a value of 0 in some fields or bits, and some can only have a certain range of values.

Thirdly, CSRs change depending on the bit-length of registers (32 or 64 bits). We will not dive into this topic, but know that some CSRs are split into two different registers when we are in a 32-bit platform.

There are many CSRs in RISC-V serving vastly different purposes. We will quickly see some of the most important and interesting ones. Others will be explained as we move on in the implementation details.

If you were wondering from 3 paragraphs above, "How can we know which extensions are available?", then the "Machine ISA and extensions" (misa) CSR is your answer. This CSR shows which extensions are available in the platform and can be modified. Nevertheless, in Qemu it is Read-Only.

Another example can be the Machine Status (mstatus). This CSR serves to obtain and modify the status of the execution. For instance, it can indicate the previous mode of execution via its Machine Previous Privilege (MPP).

One last example is Machine Scratch (mscratch) which can be used to save scratch values during execution. The crucial feature of this register is that it is protected by privileges, making it a permanent scratch for the M-mode software (in this project, Mirage).

We can operate on CSRs with some special instructions[4]. The important aspect of the instructions that modify CSRs is that they are all privileged (depending on the CSR). This means that they trap when executed without the correct privileges, allowing us to emulate them.

CSRs are operated on with basic Read and Write operations. There also exist instructions to swap values between a CSR and a regular register. Some examples are the "csrrw" (read-write) or "csrrs" (read-set) instructions.

For the purposes of emulation (remember that this is our ultimate objective), Mirage's "Virtual Context" of the firmware contains all the CSRs of the architecture.

As such, when a CSR specific instruction traps from the firmware, it has to be first be decoded. This is to know what instruction we need to emulate, but also which virtual CSRs need to be modified.

Once we obtain the instruction, we emulate it accordingly and modify the virtual CSRs with their new values. Always keeping into account that the CSR in question may be RO or WARL.

Another important element to keep in mind during emulation is unwanted behaviour. Some CSR values may be in direct contradiction with what Mirage is capable of. For instance, the Machine Interrupt Delegate (mideleg) CSR can delegate interrupts, but Mirage does not have the capabilities (for now) to handle interrupts of any kind. As such, this CSR needs to be limited to a "no delegation" policy, which would be equivalent to a Read-Only zero register. Another CSR to control related to interrupts is the Machine Interrupt Enable (mie), which allows enabling certain interrupts. Again, since Mirage does not support interrupts, this register must be maintained at 0.

3.3 Traps

As we have seen until now, traps are the way to come back to Mirage from the firmware execution. As such, supporting traps is a crucial step of the project that must be accomplished.

Traps are the combination of exceptions and interrupts, that allow to change the control flow of the program in exceptional situations. In this project, we will only support exceptions, which are needed by the firmware during normal execution. As such, interrupts are not available. This implies that when "traps" are mentioned, it refers to exceptions. Nevertheless, the following explanations can also be applied to interrupts further down the line.

Traps are an essential part of a firmware's work. Traps are the main way to change privilege modes during the execution, as such to move from and to the firmware, they are used constantly. For example, the family of exceptions Exception Call (ecall), are used to jump into other modes at will when it is necessary.

Traps in RISC-V work in a specific way. To indicate where the hardware should trap to, i.e. where the trap handler is located, the CSR Machine Trap Vector (mtvec) must be written with the trap handler's address. The execution will jump to this address when trapping. When trapping, some M-mode CSRs get modified automatically. These are: Machine Status (mstatus), Machine Cause (mcause), Machine Exception Pointer (mepc), Machine Trap Value (mtval). The mepc CSR contains the address of the trapping instruction. The mcause CSR contains the code of the cause of the trap, you can find the list of trap codes in the privilege specification [2]. The mtval CSR contains extra information about the instruction that trapped. For example, when the trap is a load fault, mtval will contain the address of the memory section which tried to be loaded. The mstatus CSR will not change on its entirety, but only the Machine Previous Privilege (MPP) field. This field will contain the previous privilege mode (U, S or M) in which the execution was at the time of the trap. This is useful because the trap may cause the privilege level to change, by going into a higher-privilege mode. In our situation, it changes from U to M-mode.

Mirage handles all incoming traps from the firmware. Nevertheless, it must also handle traps coming from its own code. This means that we must be able to distinguish traps coming from the firmware and Mirage. The important difference between Mirage and the firmware is that the firmware executes in U-mode. As such, we can identify its traps by evaluating the MPP field of the "mstatus" CSR in hardware at the moment of the trap. If the MPP value corresponds to the M-mode value, then we know that the trap belongs to Mirage. IF the MPP value corresponds to U-mode, then it comes from the firmware.

Aside from distinguishing traps by origin, during the project we did not have to handle traps coming from Mirage, so we will focus on firmware traps. As such, how do we handle firmware traps?

It is first important to know why traps can occur during the virtualized firmware execution. Exceptions can occur for a multitude of reasons.

Firstly, firmware can use exceptions to do some hardware exploration and detection. In RISC-V , hardware detection is done in the following manner. First a CSR is accessed with an instruction, then, an Illegal Instruction is raised by the hardware to make the firmware know that the component is not available. If it is available, then no exception is raised.

Secondly, exceptions occur naturally during execution to handle errors, unexpected behaviours, or demands from other level of privileges. For the sake of example, OpenSBI has two trap handlers. The first one, called Expected Traps Handler, handles "expected" exceptions, the ones it expects to occur due to accessing unknown hardware. The second is the general trap handler that is used for all other kinds of exceptions.

Thirdly, privileged instructions that the firmware should not be able to execute and that Mirage needs to emulate also trap. These instructions need to be then emulated by Mirage. One thing to note is that these instructions also raise an Illegal Instruction exception.

Now that we know why traps occur, we can handle them accordingly. The traps caused by privileged instructions can occur in the case where a normal CSR is used or when hardware is explored. Since Mirage is running on the same hardware as the firmware, we have access to the hardware's capabilities when handling the trap. As such, we can detect when the instruction needs to be emulated (normal execution) or when the trap needs to be forwarded to the firmware's trap handler.

With this information, we can now separate traps. All exceptions that are not of the kind "Illegal Instruction" should be forwarded to the firmware's trap handler. All exceptions raised due to hardware exploration need to be forwarded as well. To forward an exception, we have to emulate a jump to the firmware's trap handler. To emulate the jump, we have to modify the mstatus MPP to correspond to M-mode, since our objective is virtualization. This implies making the firmware believe nothing has changed. But also to set the virtual context to all the values that were present in hardware at the moment of the trap (mcause, mepc ...). Once that is done, Mirage has to jump to the mtvec value of the virtual context (which corresponds to the trap handler of the firmware).

The rest of exceptions are "normal" "Illegal Instruction" exceptions, which simply are the instructions we have to emulate. This implies emulating the instruction with the virtual context and then jumping back to the firmware. The instruction to jump is the one following the one that raised the exception. Or else the same instruction would be executed and raise another exception, and an infinite loop would be created.

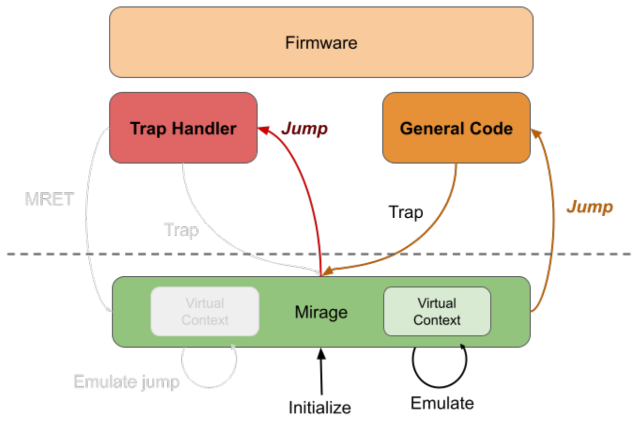

Figure 5: Simplified view of trap handling and emulation

At this point, the trap has been handled. Either, Mirage has emulated the behaviour of the instruction and jumped back to the firmware, in this situation Mirage's job is done. Or, Mirage has emulated the jump to the trap handler of the firmware, and it has executed. In this situation, there is one last step to finishing the trap handling, the Machine Return (MRET) instruction. The MRET instruction is used to return from a trap handler in M-mode back to the normal execution. This instruction applies some changes to some CSRs. As such, this instruction has to be emulated by Mirage. The most important part is that the instructions makes the execution jump to the mepc of the original trap, in our situation this corresponds to the mepc in the virtual context.

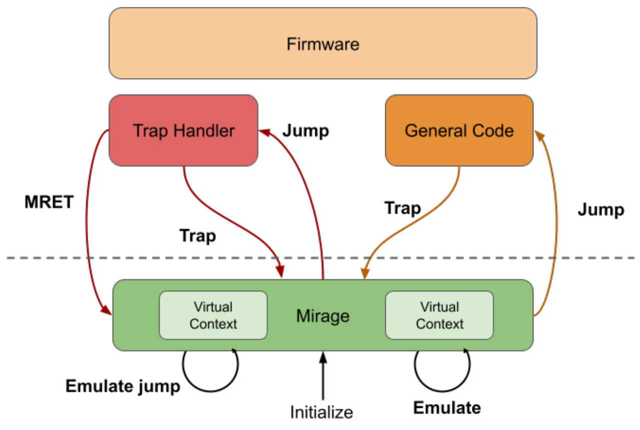

Figure 6: Complete trap emulation

4 S-mode support and execution

Our objective to virtualize the firmware is at this point partially complete. A firmware is nevertheless not very useful if its only task is to configure hardware without allowing to run anything else. That is why it is equally important to support a payload to be run after the firmware initial configuration.

One of the possible extensions in the RISC-V architecture is S-mode. This extension enables the platform to be configured to run software in S-mode, such as Operating Systems. This is exactly what we need to run a payload after OpenSBI finishes its initial configuration.

4.1 S-mode CSRs

One of the most important parts in supporting any extension, is for the CSRs related to that extension to be accessible. Supporting CSRs in our situation is being able to emulate them without raising any exception to the firmware (as we have seen before). S-mode CSRs are related to multiple elements. For example, the Supervisor Address Translation and Protection (satp) CSR allows configuring how the address translation works. If there is any. For our project, address translation is non-existent. This is done to limit the scope of the implementation. This implies keeping the CSR to a Read-Only 0 register. This is similar to the interrupt-related CSRs.

Some other CSRs, which were already available, only make sense if some other program is running on top of the firmware. For instance, the Machine Exception Delegate (medeleg) CSR allows delegating exceptions to a lower-privilege mode. In our case, it would be the S-mode payload. This means that the exceptions selected with medeleg do not get raised at the Mirage level but directly in the payload.

Supervisor CSRs have a particularity compared to other CSRs. They can be accessed without any restriction by software in M-mode, but also S-mode. This makes it that the firmware can kickstart the S-mode configuration, and it can be continued by the payload later down the line. Aside from this particularity, they behave in the same manner as M-mode CSRs.

Let's recall that our objective is to virtualize the firmware, as such the S-mode payload should not be emulated in any way. We may wonder now : "If the S-mode payload can modify the CSRs, but it's not emulated, how will the firmware obtain the information?".

4.2 Guest execution

And that would be a very good question. Since we do not emulate the S-mode payload, we do not trap on each modification of the Supervisor CSRs, but those are accessible by the firmware. As such, the correct information must be in the virtual context of the firmware when running the firmware.

That is why Mirage needs to ensure a smooth transition between the S-mode execution and the firmware execution.

To execute our S-mode payload, we have to jump to its code. The firmware takes care of this jump at the end of its initial configuration. To change the flow of an execution and at the same time change the privilege level in RISC-V the MRET instruction must be used. By indicating the next privilege level in the MPP field of mstatus, the hardware knows what mode it should be executing in. We will call this MRET jump to the S-mode payload the "Exit Point" of the firmware.

Once we know that the next mode will be S-mode, Mirage can take care of setting everything up. For this, Mirage does a "Context Switch". When the execution goes from the firmware to the payload, some registers must be in hardware. Those should be the ones that either the payload can access without extra privileges, or ones that impact the execution. In the first category, we can find all the common registers and all the S-mode CSRs. In the second category, we can find CSRs such as medeleg or mstatus.

The "Context Switch" then needs to transfer some information from the firmware's "Virtual Context" to the hardware. Remember, that the Virtual Context contains all the information that the firmware sees and modifies, as such it also includes the S-mode CSRs.

The same logic, but inverted, must be used when entering the firmware from the S-mode payload. At this point, all the information we put initially in the hardware may have been updated. As such, it must be loaded from hardware and used to update the Virtual Context. This is to make sure the firmware always sees the up-to-date information, and can be virtualized correctly.

If there is an "Exit Point" then there must be an "Entry Point". The Entry Point is when the execution comes back from the S-mode payload to the firmware. This occurs in the event of a trap. As explained before, some traps will be delegated to S-mode directly by the medeleg and mideleg CSRs. Nevertheless, all other traps will be forwarded to M-mode. In our situation, they will be received by Mirage.

Before, we have discussed trap handling in the case of a firmware trap. Now, the situation changes because these traps are coming from the payload. These do not have to be handed in a special way, but need to be forwarded to the firmware to be managed by its trap handler. It is the same mechanism as before, we will have to emulate the jump to the firmware's trap handler. The only important part is now to distinguish these traps from all the others.

To differentiate these, we can use the Entry and Exit points. These points must be used to go from the payload to the firmware and the other way around. This allows us to adapt our response to traps depending on the last point traversed by the execution. To know this, added to the Virtual Context information regarding in which mode we are running : the firmware or the payload.

If we pass through an Exit point, the execution now is transferred to the payload. This implies that all incoming traps can be forwarded to the firmware.

If we pass through an Entry point, the firmware is executing next. As such, the traps received must be handled as explained in section 3.3.

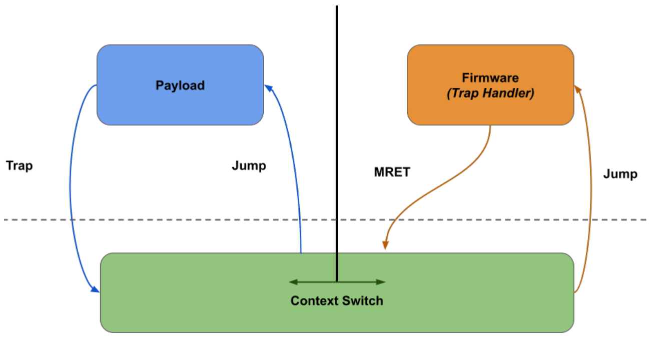

Figure 7: Simplified view of payload execution and trap handling

5 Security with PMPs

As we have mentioned in the introduction, our main objective for the project was to virtualize the firmware. After everything we have seen, the bare-bones for firmware virtualization are in place. Nevertheless, we have also mentioned the security aspect of the virtualization.

Our last objective in this project is to provide some basic security assurance to Mirage. The idea is to protect Mirage from the firmware. We will see how we successfully protected Mirage with the use of the Physical Memory Protection (PMPs) CSRs.

5.1 Deep dive into PMPs

Physical Memory Protection CSRs allow protecting regions of memory by using pattern matching on memory addresses.

Figure 8 shows the overall structure of these CSRs. Each PMP entry has a configuration and an address. Addresses are stored in their own register, and configurations are bundled together into a single register. This figure represents the structure of a 64-bit platform. 64-bit platforms in RISC-V only have the even numbered registers (0, 2, 4, 6, 8, 10, 12 and 14), the odd ones are unavailable. But they contain 8 PMP configurations per register. In 32-bit platforms, all registers are available, but each contains 4 PMP configurations.

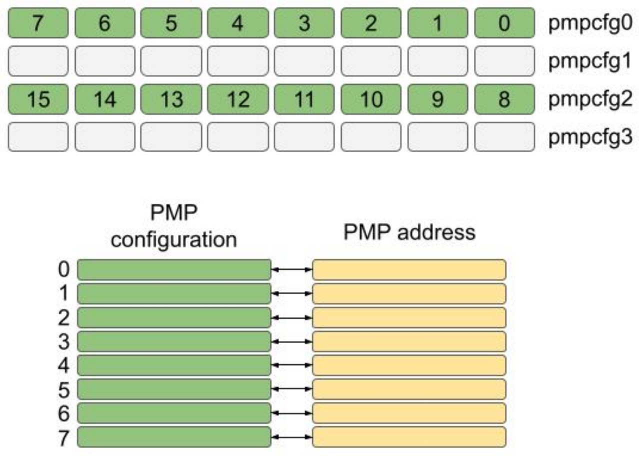

Figure 8: pmpconfig structure for a 64-bit machine with 16 PMPs, and 8 PMP entries example

One PMP entry is then the association of a configuration and an address.

The configuration allows specifying three elements. Firstly, whether the entry is locked or not. This is not available in Mirage but deserves a mention. Locking an entry makes it impossible to be modified. This can be useful to prevent any modification by untrusted parties. Secondly, the permissions related to the entry can be changed. The three basic permissions are read, write and execute. Any combination of these can be achieved in the configuration. Thirdly, the address matching mode can be changed. There are 3 matching modes to define the boundaries of a memory region using the entry's address. These modes are : Top of Range (TOR), Naturally Aligned 4-byte region (NA4) and Naturally Aligned Power of Two.

These modes change how the address impacts the matching of addresses during execution. The first thing to know is that addresses are matched in order of entries. Entry 0 has priority over entry 1, and so on. Once an entry matches using the address, the permissions are decided. The remaining addresses are ignored. If no address matches, then an error is returned. The way the address of an entry is matched by the hardware changes radically with the mode. The simplest one is TOR. In this mode, the matching region is between the address of the previous PMP entry and the current entry. This basically defines an interval of addresses which match with the entry. Each mode requires addresses to be encoded in a special way. For example, TOR requires the address to be shifter by 2 to the right in the PMP entry.

PMPs work differently in M-mode than they do in U or S mode. Most importantly, in M-mode, no PMPs restrictions apply to the running program. This allows M-mode to configure memory without running into any restrictions.

Note that in our case, the M-mode program is Mirage. The virtualized firmware should believe that it is running in M-mode. As such, we will explore later a solution to making the virtual firmware have the bare minimum of restrictions.

5.2 Mirage's PMPs

So, what do we need to protect?

Mirage's PMPs should firstly focus on protecting Mirage from the firmware and the OS. To do this, we first need to know where Mirage resides in memory. Once we have this information (which may vary depending on the configuration), we can set up the PMPs. In our situation, Mirage resides in between the addresses 0x80000000 and 0x80100000. As such, we need to protect this memory region from every action, read, writes and execution.

Figure 9 shows an example of a PMP structure that Mirage uses. This example contains a total of 16 PMPs. In this example, the first 2 PMPs are used to protect Mirage's memory region with a TOR configuration.

The other thing that Mirage's PMPs need to take care of is giving to the firmware the illusion that it has all PMPs for itself. For that, we require two elements to be present. As mentioned before, the root PMP has address and configuration set at 0. This needs to be emulated for the firmware. As such, the PMP before the ones allocated for the firmware needs to contain the value 0 all the time. Figure 9 shows an example of where that 0 value would be placed in the whole PMP structure.

There is another element we need to virtualize with Mirage's PMPs. That is, the firmware needs to be allowed to access the rest of memory without any restrictions. For this, we can set the last PMP to allow all actions on all possible addresses. This ALL allowed PMP is shown in Figure 9 as well.

To allow Mirage's PMPs to cohabitate in hardware with the firmware's, we introduce a PMP offset into the Virtual Context. This offset allows the firmware to read and write PMPs without leaking or overwriting Mirage's information.

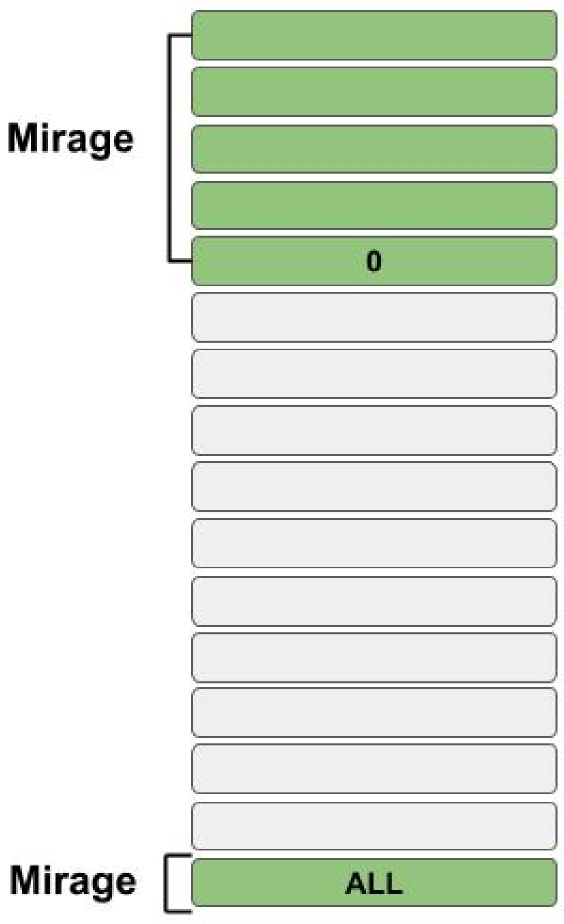

Figure 9: Mirage's PMP configuration structure

5.3 Firmware's PMPs

The offset is crucial to the virtualization of the firmware's PMPs. As we explained before, multiple PMP configurations are present in the same register. Because Mirage's configuration can be present in one of these registers, the offset allows filtering out Mirage's values for the firmware.

Now that PMPs are available to the firmware, they need to be emulated correctly. Aside from the offset, we have to consider another value. That value would be the amount of PMPs that are allocated to the firmware. This value is given to the firmware by Mirage during its initial configuration.

There are two possible operations on PMPs, read and write. In the Virtual Context in which these operations operate, PMP values are as if in hardware. This means that the first PMP of the firmware corresponds to the first PMP entry in the Virtual Context. To read a PMP we have to filter out the values that do not belong to the firmware, those exceeding the allocated number. The same happens when writing. The written value is filtered to not allow changes outside the allocated entries.

Aside from the number of entries, there is one other element to consider. Since we operate on an 64-bit environment, all reads and writes to an odd PMP configuration CSR should not work.

To achieve full PMP emulation, the CSRs need to be placed in hardware during the execution of the S-mode payload. This makes it important that during the Context Switch explained in section 4.2, PMPs are taken into account.

If the U-mode firmware emulation is executing, the firmware's PMPs do should not be in hardware, or else they would impede the firmware itself from accessing memory. Since the firmware is in U-mode, PMPs in hardware affect it, but since we are virtualizing it, only the minimum should affect it (Mirage's protection). The last PMP should be configured to allow all memory to the firmware.

If the S-mode payload is executing, the PMPs configured by the firmware should be active. Since this is the configuration, the payload should be restrained to. In this case, the last PMP should not permit the payload to access all memory unrestricted. To do this, we can simply modify the configuration to have no permissions.

It is important to note that the firmware's PMPs have no offset in the Virtual Context. As such, they have to be moved before entering the hardware to not overwrite Mirage's protection.

Figure 10 shows an idea of what should change. The arrows represent the context switch.

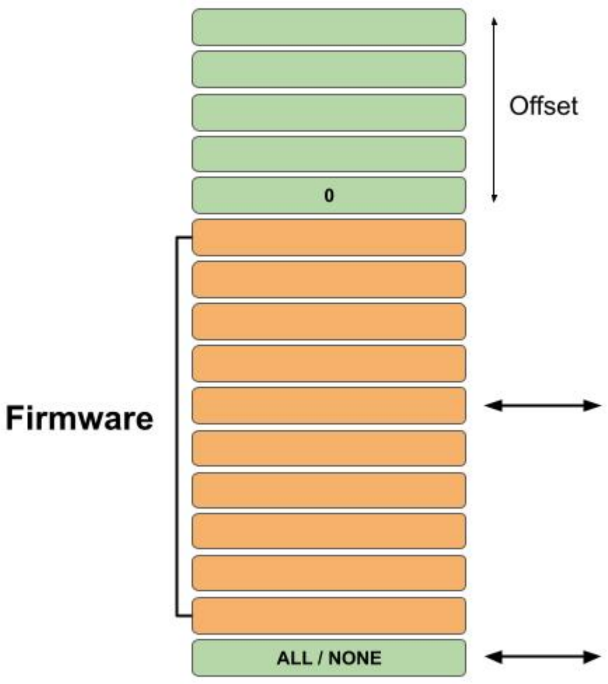

Figure 10: Firmware's PMP configuration structure

One last element to tie the whole PMP implementation together is the emulation of the "vfence.vma" instruction. This instruction is used to flush all caches for memory location. These caches contain information about the PMPs (among others). As such, when a PMP is modified, the instruction needs to be called.

Since OpenSBI is at this point aware of its PMPs it uses this instruction after its initial PMP configuration.

6 Conclusion

In the first section, we have seen how we managed to make emulation work. We saw how basic emulation works in the context of firmware virtualization. But also how M-mode CSRs can be adapted and controlled for the sake of virtualization. And finally, how traps are an all-important tool for the firmware and as such for its emulation. We saw how to differentiate traps coming from multiple places and how to emulate their effects.

In the second section, we explored how to adapt the execution and virtualization to make a non-virtualized software work. Even when its execution directly depends on the outcome of the firmware execution and emulation. For this, we explored how S-mode CSRs are different from M-mode CSRs in some aspects and what we need to take into account when emulating them. More importantly, we saw how to switch from one execution context to another thanks to entry and exit points.

In the third and last section, we looked at the security capabilities of PMPs, and how they allow protecting Mirage from the firmware. During this, we looked at how PMPs work in more details. But also how Mirage can protect itself during the execution of the firmware without sacrificing any of the capabilities of the firmware.

With all of this said, we can safely conclude that the objectives of the project have been attained. Mirage is now capable of executing and controlling a complete firmware such as OpenSBI. But Mirage can also offer some protection to itself via PMPs.

6.1 Discussion : Limitations and Expansion

Nevertheless, there are some limitations and expansions to be made to the current state of Mirage. We will mention some of them here.

Firstly, from a security point of view, PMPs are only used to protect Mirage itself. But the firmware can also affect other programs such as an Operating System without any issues. As such, it would be ideal for Mirage to offer some protection to other parts of the system as well.

Secondly, as mentioned before, interrupts are not support in Mirage. Interrupts are used by the firmware to wait for certain actions, as such, they should be supported.

Thirdly, page tables are related to the S-mode CSR satp, which during current emulation is kept at 0. This makes page tables unavailable. But page tables are used in almost all executions. Making them an useful addition.

Fourthly, some major features are still lacking. Automatic hardware detection to know which elements are available on the board Mirage is executing would be useful. As to have a single version of Mirage capable of configuring different boards without any issue. This relates also to supporting 32-bit platforms, not only 64-bit, and common extensions such as the Debug and Hypervisor RISC-V extensions.

Fifth, and the most complex, would be to support other architectures aside from RISC-V such as x86. This would of course come with its own set of challenges, unique to each architecture.

References

[1] RISC-V International. *RISC_V International.* URL: `https://riscv.org/`.

[2] RISC-V International. *Specifications - RISC_V International.* URL: `https : / / riscv . org / technical/specifications/`.

[3] QEMU. *QEMU.* URL: `https://www.qemu.org/`.

[4] riscv-isa-pages. *RV32I, RV64I Instructions.* URL: `https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html#`.