Philipp Jantscher

Counteracting Code-Reuse Attacks using Tagged Memory on a RISC-V Architecture

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Mario Werner

Assessor

Prof. Stefan Mangard

Institute of Applied Information Processing and Communications

Graz, September 2016

AFFIDAVIT

Acknowledgements

I would like to thank my supervisor Mario Werner, for his technical and organisational knowledge and for the countless productive discussions. Without his help, this thesis would not have been possible. Same applies also to Prof. Stefan Mangard for supporting the idea of this thesis and giving me the opportunity to finish my master studies at the Institute of Applied Information Processing and Communications (IAIK). I also would like to thank the lowRISC team at the University of Cambridge for giving me technical support and constructive feedback during the chip implementation. Especially Wei Song provided me with detailed information of the lowRISC SoC throughout this thesis. A part of my thesis will be contributed to this project and I feel truly honored about this fact.

Throughout my studies, I had a lot support from my parents and study colleges, whenever I needed it. Especially, I would like to thank Richard Fellner, for our good teamwork and study groups we formed during our time at the Graz University of Technology. The list of further people I would like to thank, is too long to include in this thesis, which is why I direct this note to all my friends and colleagues, who followed and supported me for years on my path, till the present day.

Kurzfassung

Moderne Computer Systeme werden immer häufiger Ziel von komplexen Attacken. Programmcodes enthalten meist viele Schwachstellen, welche benutzt werden können, um Daten auf dem Stack oder im Heap zu überschreiben. Dies kann durch die Anwendung von Speicherüberlauf Techniken erzielt werden. Diese Überläufe ermöglichen es dem Angreifer, so genannte Code-Reuse Attacken zu initiieren. Diese Angriffsklasse is nicht davon abhängig, böswilligen Code einzuschleusen. Stattdessen wird bei Code-Reuse Attacken bereits im Programm existierender Code verwendet, um den Programmablauf zu ändern und die Attacke auszuführen. Dies kann durch das Überschreiben der Rücksprungadressen am Stack für Return-Oriented Programming Attacken, oder von Funktionszeigern für Jump-Oriented Programming Attacken erreicht werden. Generell ist es schwierig, erlaubte Programm Sequenzen von den Angreifer verursachten Sequenzen zu unterscheiden. Einige Forscher haben bereits eine Vielfalt von Software und Hardware basierten Gegenmaßnahmen für Code-Reuse Attacken entwickelt. Allerdings enthalten diese Lösungen immer noch einige Probleme und Schwachstellen. Die meisten Gegenmaßnahmen erfordern Änderungen der Software oder des Compilers und verursachen Leistungseinbußen. Vor allem das Unterscheiden von validen und böswilligen Funktionszeigern stellt sich als besonders schwierig heraus.

In dieser Masterarbeit stellen wir eine effektive Gegenmaßnahme gegen verschiedene Typen von Code-Reuse Attacken vor, welche komplett von der Hardware umgesetzt wird. Dadurch ist das System transparent gegenüber dem Betriebssystem und dessen Software. Wir stellen eine Prozessor Architektur vor, welche Tagged Memory verwendet, um Rücksprungadressen und Funktionszeiger vor Angreifern zu schützen. Durch so genannte Tagged Security Policies wird es möglich, Return-Oriented Programming und Jump-Oriented Programming Attacken zu verhindern. Die Architektur ist basierend auf dem lowRISC System-on-Chip Design implementiert, welches wiederum eine Modifikation des Rocket Chip darstellt. Der Rocket Chip wurde an der Universität von Berkley entwickelt. Das Prozessor Design basiert auf der RISC-V Instruction Set Architektur und wird häufig von Forschungsteams verwendet. Weiters wurde das Design auf einem Kintex-7 FPGA, auf welchem Linux ausgeführt wurde, getestet. Das System erfordert nur 6.25% des gesamten Hauptspeichers. Die notwendige zusätzliche Ausnutzung für Flip-Flops und Lookup-Tabellen des FPGA's ist nur um ungefähr 10% erhöht. Diese Erhöhung ist hauptsächlich durch die vergrößerten Caches, welche für das Speichern von Tags benötigt werden und durch den zusätzlichen Tag Cache begründet. Wir erwarten, dass der neue Ansatz, Tagged Memory Prozessor Architekturen zur Abwehr von Code-Reuse Attacken einzusetzen, die generelle Sicherheit von modernen Computer Systemen verbessern kann. Weiters wird die Effizienz der Systeme sogar für Low-End Anwendungen sicher gestellt.

Stichwörter: Code-Reuse Attacken, Tagged Memory, ROP, JOP, RISC-V

Abstract

Modern computing systems are getting more and more a target of advanced attack strategies. Most code still includes a lot of vulnerabilities, which can be exploited in order to overwrite data on the heap or stack. This can be archived by applying buffer overflow techniques. Buffer overflows allow attackers to mount so called code-reuse attacks. This kind of attacks do not inject code, but instead alter the control flow and execute already existing code within programs. This is done, by either modifying the return addresses on the stack for return-oriented programming or by modifying function pointers for jump-oriented programming. It is a hard problem to distinguish good code sequences from malicious sequences, which the attacker is initiating. Researchers developed a wide range of software- and hardware-based countermeasures for these attacks, but they still include drawbacks and flaws. Most existing techniques require software or compiler modifications and introduce a significant performance overhead. Especially, distinguishing valid from malicious function pointers is a problem for hardware-based solutions.

In this thesis, we propose an effective countermeasure against different types of code-reuse attacks, which is entirely enforced within hardware and is therefore transparent to the operating system and applications. We introduce a tagged architecture, which enforces protection of return addresses and function pointers. This is achieved by tagged security policies and as a result, the system is able to prevent return-oriented and jump-oriented programming attacks. The architecture is implemented on top of the lowRISC System-on-Chip, which is a variant of the Rocket chip that is based on the RISC-V instruction set architecture. This chip is currently used for various research topics. Furthermore, we tested the design on a Kintex-7 FPGA while running Linux on the modified architecture. Our design only requires 6.25% of the total main memory for tags. The overall utilisation of lookup tables and the flip flops of the FPGA is increased by roughly 10% each. This increase is mainly issued by the additionally required cache memory for tags and the newly introduced tag cache. We expect that this new approach of using tagged memory architectures as a countermeasure for code-reuse attacks, could enhance the security aspects of future processing systems while remaining efficient, even for low-end use cases.

Keywords: Code-Reuse Attacks, Tagged Memory, ROP, JOP, RISC-V

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Modern computing systems are getting more and more a target of advanced attack strategies. With the growing interconnection of services, based on server applications and the Internet of Things, the potential damage an attacker can cause is increasing accordingly. Although both software and hardware designs are evolving to support advanced security mechanisms and most programmers are aware of the problems, the systems remain vulnerable. Attackers are always able to find new exploits. Most code includes vulnerabilities, which can be used to overwrite data in the heap or stack. This can be archived by applying buffer overflow techniques. The problem occurs, when buffer range checks are missing or if the attacker is able to exploit a so called format string vulnerability. This allows the attacker to inject arbitrary data and code, in order to cause malicious behaviour. However, classic code injection, where the return address is overwritten in order to point to the injected code, is impossible in modern operating systems due to Data Execution Prevention (DEP) [16]. To overcome this limitation, attackers developed a new class of attacks, called code-reuse attacks [13][27]. This kind of attacks do not inject code, but instead alter the control flow and execute already existing code within programs. This is done, by altering return addresses on the stack or by modification of function pointers. Therefore, it is necessary, to reuse existing code snippets, called gadgets, which represent the attack code. These gadgets are mostly only consisting out of a few instructions, which can be chained together. Thus, the good code can cause malicious behaviour, when attackers reuse library functions in order to get access and take control over the machine.

The difficulty of developing countermeasures against code-reuse attacks is, that it is a hard problem to distinguish code sequences that are intended to execute from malicious sequences, which the attacker is initiating. Another issue occurs, because software-based countermeasures often infer a significant performance overhead and the solutions are not transparent to application code or operating systems. A promising countermeasure is Control Flow Integrity [30]. This technique is ensuring valid transitions of the control flow, by comparing the target addresses of control flow transitions with a statically generated Control Flow Graph (CFG). Only paths, which are existing in this graph, are allowed to be performed. Whenever an attacker modifies a target address, the operating system detects the malicious behaviour. The downside of this approach is the introduced performance overhead and that dynamic address calculations within the program can not be secured. Static analysis is not capable of generating the CFG for these situations. Also, return addresses can not be protected with CFI, since return operations can have multiple allowed targets, and a CFG for this situations would be too complex. Thus, return-oriented programming attacks can not be prevented.

Recently, also hardware-based countermeasures for code-reuse attacks have emerged [7][10][22][25]. In general, this can be archived by dynamic flow tracking and enforcement of security policies within the core. As within our thesis, tagged memory is used to store meta information for every data word. This allows the system, to distinguish malicious and trusted data within the program. However, at this point in time, no countermeasure exists, which ensures the validity of return addresses and function pointers, without the need of additional software or compiler support. Especially, the efficient identification of valid and invalid function pointers remains unsolved with previous solutions.

In this thesis, we aim to implement an effective countermeasure against code-reuse attacks, which is entirely enforced within hardware and is therefore transparent for the operating system and the application programmers. We propose a tagged architecture, based on the lowRISC System-on-Chip (SoC) [14]. This SoC incorporates and extends the Rocket Chip [32], which is a core that supports the RISC-V [23] instruction set architecture and is currently widely used for research purposes. RISC-V offers a highly extensible and flexible instruction set and is developed by the University of Berkley. We introduced tagged memory on the untethered lowRISC SoC, which is supported by a specially designed tag cache. Our tagged memory implementation is based on a previous tethered lowRISC release, and had to be redesigned due to major architecture modifications for the untethered lowRISC release. Furthermore, we performed modifications to the core pipeline, in order to implement tagged security policies. This policies require a processor core, which is aware of tags, mapped to the corresponding core registers. In order to run the system, minimal modifications to the Linux kernel are performed, in order to ensure full compatibility.

In general, tagged memory adds meta information to the data words. In our case, we introduce four tag bits per 64-bit data. Two bits are used for the tagged security policies, which protect return addresses and function pointers. The remaining two bits can be used for user defined policies. In order to protect the return address, which is located within the stack, we set a tag bit for the corresponding address in the memory. This tag bit is called return address tag and indicates a valid return address. The bit is set, whenever a call instruction is executed and the return address is saved in a temporary register. The tag bit propagates into memory and throughout operations. However, if a modification in form of an arithmetical or logical or even partly memory stores/loads of this register is performed, the tag bit is cleared. This leads to an invalid return address value, and the processor detects the security violation. In order to protect function pointers, which are later used for indirect call instructions, a dynamic flow tracking technique is introduced. Trusted and untrusted data is distinguished by assuming all data, loaded from peripheral devices as malicious. Therefore, all data, which is loaded form any of these devices is tagged as invalid for a call instruction, using the second tag bit. This tag, other than the return address tag, is not cleared when operations are performed with the register. Instead, the input tags are propagated through the output of the execution. This allows a dynamic propagation of this tag bit, since the result of a operation, which would modify any valid data with invalid data, also leads to an invalid output. This kind of data can not be used as jump target for indirect call instructions anymore.

The tags are stored within an own section in the main memory. This means, that data and tags need to be loaded and stored separately, while correctly assigning the tags to each data field. In our architecture, this is handled by a tag cache, which also increases the performance of the system, since it caches the tags. The core receives tags and assigns them to the core registers, by storing them in a separate tag register file. The enforcement of the policies requires a tag propagation unit and a unit, which recognizes malicious return addresses and function pointers within the core. The tag propagation unit is denoted as tag ALU and applies rules on how to propagate tags for given instructions and policies. The tag check unit compares the current tag value of the input registers and validates, if the indirect call instructions are allowed to be performed. This is done, by comparing the tag values to the given rules for each tagged security policy.

We implemented the tagged architecture and enforced the tagged security policies on a Kintex-7 FPGA from Xilinx [12], running a Linux kernel. The system remains fully operational, while preventing code-reuse attacks from being executed. This fact is proved with various test applications, which apply the typical attack initiation strategies for return-oriented and jump-oriented programming attacks. Our tagged security policies require four tag bits per 64-bit data and as a result, introduce a minimal memory overhead of 6.25% of the total memory size. The flip flop utilization of the FPGA is increased by 9.45% and the lookup table utilization is increased by only 7.22%. This results from the added control logic and the tag cache. This shows, that tagged memory and tagged security policies do not significantly increase the chip complexity and size.

The remainder of this thesis is structured as follows. In Chapter 2 we evaluate buffer overflows in general and provide background information of code-reuse attacks and its sub classes. Chapter 4 briefly summarizes the principles and the evolution of tagged processor architectures. The RISC-V Instruction Set Architecture (ISA) is described within Chapter 3. We continue with a detailed description of our proposed tagged security policies against return-oriented and jump-oriented programming in Chapter 5. Chapter 6 summarizes related work and compares previous solutions to our tagged security policies. Details on the implementation of the architecture are presented in Chapter 7. In Chapter 8, we continue with the evaluation of the results and future work. Finally, we provide a conclusion in Chapter 9.

Chapter 2

Common Software Vulnerabilities

Current state of the art software is often flawed by potential security vulnerabilities, that allow attackers to cause malicious behaviour. Modern software design flows and countermeasures in highly secured architectures are existing, but still some attackers always find exploits. Software vulnerabilities are mostly caused by not paying attention on range checks, or by using insecure user IO functions. Although it is widely known, that this kind of programming errors can cause a lot of damage, they are still existing in modern systems. The attacker will search for some kind of entry point, like an unchecked length of the input, in order to overflow the stack or function pointers. The goal of attackers is, to redirect the control flow in order to execute arbitrary code and take over control of the machine. The usual way to accomplish this behaviour, is to start a shell by use of the attacked program and execute arbitrary scripts within that shell. The interesting thing about this strategy is, that the executed shell gathers the same access rights as the attacked program. If a program with root credentials is attacked, pretty much everything can be executed within the shell scripts. In this thesis, we provide countermeasures for attacks, that use these kinds of buffer overflow vulnerabilities of victim programs. More specifically, the class of code-reuse attacks is the target of the countermeasures. The following sections provide background knowledge for buffer overflows in general, followed by details of code-reuse attacks.

2.1 Buffer Overflows

Buffer overflow vulnerabilities within programs are often the entry point for an attacker. For example, attackers can take control over network servers, running software that includes these leaks. This section describes buffer overflows in detail and points out its causes.

The usual cause for a buffer overflow vulnerability is a mistake by the programmer. Often, functions like *fgets()*, *scanf()* or format strings [29] can be used to exploit a buffer overflow vulnerability. The problem occurs, if array bounds are not checked and user input can overwrite bigger memory areas than expected by the programmer. In this case, the attacker can modify the value at an arbitrary location in memory. The code snippet, presented in Listing 1 is, for example, vulnerable to buffer overflows.

```
void main(void)
{
  byte buffer[4]
  scanf("%s", array);
}
```

Listing 1: An example of code, vulnerable to buffer overflows.

In this simple example code, *scanf* is used to read input from the user and to store it in the array buffer. If the user inputs a string bigger than 4 bytes, other variables on the stack will be overwritten. The stack is overwritten, since the array is instantiated within the functions scope, rather then being global or in the heap.

In the case of a stack buffer overflow, the likely goal for the attacker is to manipulate the control flow, by overwriting the return address, stored on the stack. This works, because the stack grows downwards for every variable, declared within the function scope. On x86 for example, at a function call, first the previous stack pointer is pushed onto the stack, followed by the function arguments. Afterwards, the return address, which points to the next instruction after the call, is pushed onto the stack, in order to be able to continue execution after the function returns. After all this necessary steps, the variables of the functions will be instantiated on the stack. If the attacker is able to write more data to the variables on the stack, the write address increases until it reaches the address of the return address. If this value is exchanged with an arbitrary value, the control flow is changed according to the attacker, once the function returns. Figure 1 illustrates this situation by showing the contents of the stack before and after a buffer overflow.

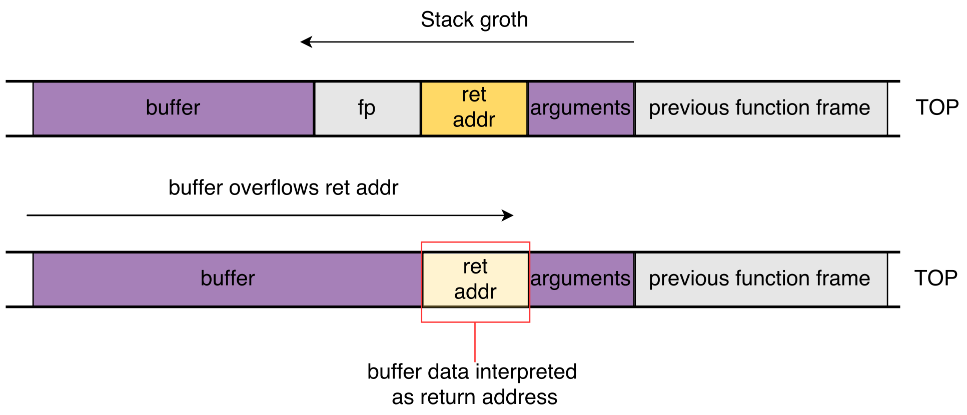

Figure 1: Stack buffer overflow. The return address is overwritten by the injected content.

In the case of a heap buffer overflow, the attacker is not targeting return addresses, but function pointers. Again, an unchecked input is used to overwrite arbitrary memory locations. If the attacked code is well known, attackers can search for function pointers residing on the heap or global address space, in order to change the control flow. The next time, the application will call a function by referencing the modified function pointer, the attacker chosen code is executed. Without compiler supported security measures, this change of the control flow is not recognized. Figure 2 shows the principle of overwriting the function pointer, located somewhere in the heap.

The simplest attack is, to inject malicious code on the stack or heap and then change a return address or a function pointer to the start of the injected code. However, on modern systems this approach is pretty much impossible since the implementation of the Data Execution Prevention policy, which is also denoted as X-xor-W policy. This policy prevents data form being executed, since pages can never be both writeable and executable, when this policy is enforced by either software or hardware mechanisms.

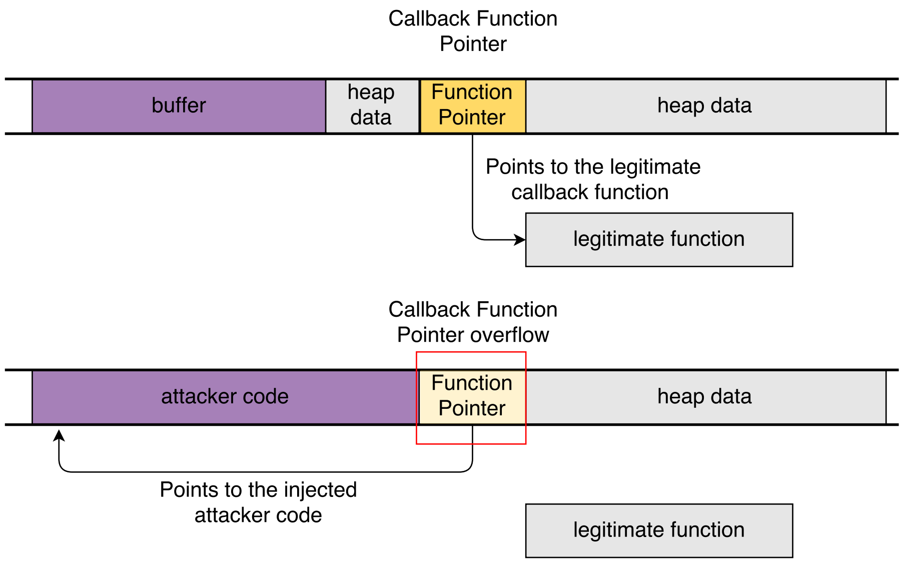
Callback Function Pointer
buffer
heap data
Function Pointer
heap data
Points to the legitimate callback function
legitimate function
Callback Function Pointer overflow
attacker code
Function Pointer
heap data
Points to the injected attacker code
legitimate function

Figure 2: Heap buffer overflow. The function pointer is overwritten with an arbitrary address, which points to the attackers function.

2.2 Code-Reuse Attacks

To overcome the limitation of the forbidden code injection, a new class of attacks, called code-reuse attacks, is introduced. They are used to perform arbitrary code execution by applying different techniques or even deactivate the DEP during the attack process. Code-reuse attacks can be initiated using standard buffer overflow techniques as already described. As evaluated already, the usual buffer overflow attacks, where attackers inject code onto the stack are almost impossible since the introduction of various security mechanisms. In the case of DEP, all pages, which are writeable or were once modified, are not executable anymore, restricting injected code to be a target of the control flow mechanisms. In other words, only not modified program code can be executed and not data.

However, with code-reuse attacks, code injection is not necessary anymore, due to reuse of existing code within the binary or shared libraries. Therefore, the argument, that good code always results in good behaviour and malicious code results in bad behaviour, is not valid anymore. Attackers are now able, to induce bad behaviour, by using instructions, which are already located within the good code. This reuse of instructions within the binary requires a big code base, to apply complex control flow changes and achieve the desired bad behaviour. Since most programs use shared libraries, the code base already includes a lot of useful instructions for the attacker, such as system calls and memory altering instructions.

The code snippets or single instructions, which are executed within a code-reuse attack are called gadgets. These gadgets only include a couple of useful instructions. Usually, code-reuse attacks implement techniques to chain multiple gadgets and therefore create a Turing complete language. The gadgets have to be found beforehand with various algorithms in order to design the attacks. Countermeasures for these attacks exist and are more or less effective. A lot rely on randomizing the location of the program and library code. This prevents the attacker from knowing the location of gadgets beforehand, and as a result makes it harder to detect them. An example for this system is the Address Space Layout Randomization (ASLR) [5], which is commonly used in modern operation systems. The location of the code differs every time the program is executed and the prediction of where gadgets are located is harder. However, if the attacker manages to locate a gadget within this randomized space, all other locations can be inferred. Another approach is the the monitoring of control flows. Code-reuse attacks mostly infer a pattern of execution, which uses call and return instruction more frequently as in normal program executions. This anomalies can be detected by the operating system, but infers a significant performance overhead to the whole system. In this thesis we propose a countermeasure using tagged memory, which also senses anomalies in the control flow in an efficient manner. The proposed mechanisms are introduced in Chapter 5.

In the following sections we introduce two types of code-reuse attacks, operating with different principles on how the attack is mounted and executed afterwards. While return-oriented programming relies on stack corruption and return instructions, the jump-oriented programming archives the same goals, using function pointers. Both use chained gadgets, to accomplish the attacker's goals.

2.2.1 Return-Oriented Programming

The first code-reuse attack to be introduced, is Return-Oriented Programming (ROP) [27]. The name implies, that the program flow is not proceeded using ordinary function calls and linear execution, but instead uses return instructions to execute arbitrary programs. It can be seen as an alternate type of programming. However, its main field of usage are attacking strategies. As already mentioned, ROP uses buffer overflow techniques to mount the attack initially. In this particular case, a buffer on the stack needs to be overflowed. The basic principle is using a standard vulnerability like format strings to overwrite contents of the stack. But since injected code is not a solution because of the mentioned countermeasures like DEP, no code is injected for this type of attack. Instead, supporting data contents are injected and also the return address is overwritten. The stack has to be overwritten in an area around the current stack pointer. As the stack both contains return addresses and function arguments, the contents can be rewritten to accomplish full control for the attacker. So the entry point for the attacker is redirecting the control flow to the first selected gadget, by overwriting the return address in order to point to the correct gadget location. Once the return of the ordinary function within the attacked application is performed, the control flow is redirected to an attacker chosen instruction. This instruction is part of the first gadget to be executed. Return-oriented programming does not perform function calls at all. The control flow is entirely altered by chaining gadgets, which also end in a return instruction. Multiple variants of return-oriented programming exists, like for example return-into-libc. In this case, a return instruction is used to call a libc function. For example, an attack could call the *exec* function, which could launch a new shell. This is not a traditional return-oriented programming scheme, since only one function is called, in contrast to chaining multiple gadgets.

Most classic return-oriented programs use instructions, located within the libc rather than using whole functions of the library. This means, that the altered return addresses on the stack point to instructions within libc. This opens a lot of possibilities for the attacker, since the library has a large code base and is available in most programs. Gadgets can be found beforehand, and then simply executed in the attacked program. Also the library includes system call instructions, which can be used to take control of the machine and launch a shell with administrator rights. Once the attacker gathered all necessary gadgets, to perform a meaningful attack, the exploit process is started. As already mentioned, ROP does not perform function calls, but instead enters all gadgets by execution of a return instruction. This requires a new computing model, in order to allow one gadget, to transfer the control flow to the next gadget within the chain. Therefore, a new artificial instruction pointer is introduced, which always points to the next gadget within the chain. In general, return instructions always perform a jump, to the return address, which is located in the current stack frame. It is assumed, that before a return instruction is performed, the next return value is automatically loaded by the processor, from the location of the return address within the stack. This means, that the return address is popped from the stack and the stack pointer is advanced. This is true for x86 systems. A gadget only needs to perform the return instruction and it is assured, that the control flow is redirected to the next return address within the stack, since the stack pointer will point on the next memory location. In normal program flow, this ensures, that the callee gains control over its stack variables again, after the called function returns. ROP exploits this fact, and uses the stack pointer as a artificial instruction pointer. Imagine, that after the return of the gadget, the stack pointer is positioned on an attacker modified location. Then the attacker knows, where the next return address has to be located on the stack. Thus, attackers are able to inject all return addresses, which point to gadgets, on the stack. When a gadget returns, the processor simply advances the stack pointer and reads the return address given in the current stack frame. Using this technique, gadgets can be chained and the instruction set supports the exploit. On top of the possibility to arbitrarily advance the control flow within the attacker's program, also temporary data storage can be achieved. By using gadgets, which execute push or pop instructions, data on the stack can be modified. If this is the case, the gadget needs to ensure, that the stack pointer stops again at a valid gadget address after data is popped off the stack. Figure 3 visualizes this basic operation principle of ROP.

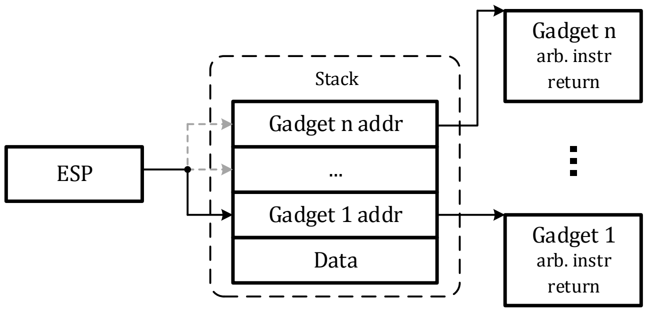

Figure 3: ROP design principle. The stack is overflowed with the gadget addresses. Also data can be injected, which serves as temproary storage for the attack. The stack pointer (ESP) always points to the next gadget address on the stack. The functional gadgets 1 to n are entered by a return instruction, after the new gadget address is loaded from the stack. In the end of each gadget, a return instruction is performed, which starts the execution of the following gadget.

However, before this attack can be mounted, appropriate gadgets have to be found within the library first. Gadgets, which perform useful work, mostly contain at least one instruction and end with the return instruction. Different types of gadgets are needed, to create a Turing complete language. This gadgets have to include types that perform arithmetical and logical register to register operations, as well as control flow altering and memory access procedures. It turns out, that even with this high demand on complex and specific gadgets, all required types can be found within the library, using specifically designed algorithms. Also, the gadgets need to end in an return instruction, which means that finding an appropriate gadget is even harder, since it has to include a useful instruction and the return instruction in addition. However, on x86 systems, this process can be executed more efficient, since the instruction word does not have to be correctly aligned in memory. Therefore, the algorithm can shift byte wise through the code base and create appropriate gadgets out of shifted original code words. The result addresses are then injected as the gadget start address and processed by the ROP mechanism.

The previously described operation principle and and the gadget search phase is not applicable for RISC architectures. However, researchers state, that ROP is possible on any system architecture, but the details of the execution needs to be altered [26]. Within the RISC instruction set, the return instruction does not perform an automatic relocation of the stack pointer. In addition, the return address is not loaded by the return instruction, but needs to be read from the stack and stored in the return address register with separate instructions. The gadget discovery process is significantly more complicated because of this facts. The gadgets do not only have to end in an return instruction, but also prepare the next return address before this instruction is executed. It is also possible, that the gadgets jump to a separate control gadget, which prepares this return address and then continues with execution of the next gadget. A similar strategy is used within jump-oriented programming. What needs to be kept in mind is, that instructions on RISC architectures always need to be correctly aligned in memory. The byte shift strategy to create new instructions out of existing code is therefore not applicable on this architectures.

2.2.2 Jump-Oriented Programming

Since ROP is relying on the stack and the extensive use of return instructions, it can be detected by some countermeasures. These countermeasures mostly operate on a software basis and include a lot of overhead in terms of performance. However, the problem that ROP attacks are not possible in every situation still remains. Therefore, a new type of attack has been developed, to overcome this issues. Jump-Oriented Programming (JOP) [13], builds on the main idea behind ROP and applies different mechanisms for initiating and the execution. Instead of using gadgets, which end in return addresses, it uses solely jump instructions to execute one gadget after the other. This means, that the vulnerability of the attacked program now lies in the function pointers, which are existing within the code base. The attacker overwrites any function pointer and when the function pointer is called the next time, the attacker's function or instruction is executed. This way, the attacker does not need to overwrite the stack, but any arbitrary location within the heap or global space. This is also performed, by using a buffer overflow vulnerability, where the buffer is located in this memory areas. In case of ROP, the control flow is altered by execution of return instructions. This instruction also provides the increment of the stack pointer, which is used as instruction pointer within the stack and always points to the next gadget. The problem with jumps is, that this feature is not included anymore and the execution of a jump is only working in one direction. This means, that gadgets wouldn't be able to jump to the next gadget. To overcome this issue, a so called dispatcher gadget is introduced. Basically this gadget is the control unit of the newly created computer and manages the chained execution of gadgets. Therefore, the dispatcher handles the advancing of the instruction pointer, which is also only a pointer to the memory location, where the address of the next gadget is stored. Besides that, the dispatcher gadget does not perform any useful operations. The useful operations are performed by the so called functional gadgets, which are analogue to the gadgets used in return-oriented programming. Functional gadgets are required to end in another jump instruction. Also, these jumps need to ensure that they return back to the dispatcher gadget. There, the address of the next gadget is loaded and executed. Because of the dispatcher gadget, the data, which includes all gadget addresses, can be located anywhere in memory. It might as well again use the stack for data storage and gadget list. The crucial point of the execution of a JOP attack is, to find an appropriate dispatcher gadget and functional gadgets, which end in indirect register based jumps. Within libc, this gadgets can be found quite easily and it is therefore possible to create a touring complete language, comparable to ROP. An additional convenience of JOP is, that also call instructions can be executed. Therefore, normal functions can be reused, since the modification of the stack on function calls does not alter the attack code. In ROP, a function call would alter the values on the stack and therefore overwrite the injected addresses and data of the attacker.

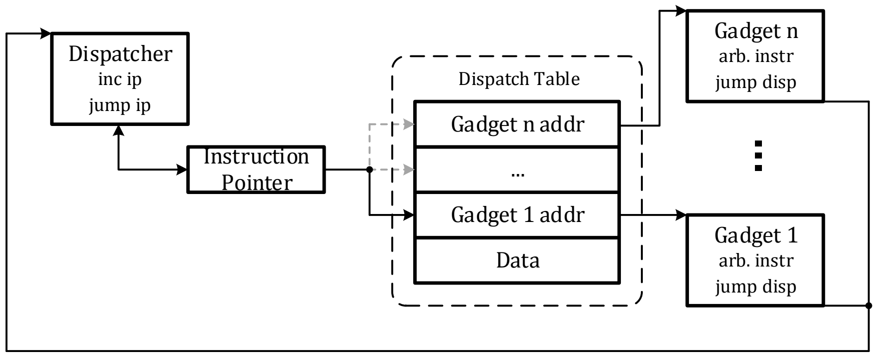

Figure 4: JOP design principle. The dispatch table holds all adresses of gadgets to be executed. Also, this table can include data or serve as temporary data storage. The dispather always increments the JOP instruction pointer with a given rule. The functional gadgets 1 to n are entered by a jump instruction. In the end of each gadget, a jump to the dispatcher gadget is performed.

Figure 4 gives an overview and an example of a jump-oriented program. As it can be seen, the so called dispatch table holds all data and gadget addresses, which shall be executed. This table can be located anywhere in memory and represents the attackers program code. The addresses within this table point to the functional gadgets, which perform the attack program. On the other hand, there is also the dispatcher gadget. Its pointer always points to the address of the gadget within the dispatcher table, which shall be executed next. When the attacker injects the table, he also overwrites a function pointer with the address of the dispatcher instruction. When the function pointer is used in a jump instruction, the dispatcher dereferences the address in order to gather the first functional gadget instruction. Then it increases the pointer, which then points to next functional gadget address in the table. This pointer represents the instruction pointer as within the processor. Then a jump is performed to that address. In the end of the functional gadget, the address of the dispatcher is loaded. If the jump at the end of the gadget is performed, the dispatcher is executed again. The correct execution of a JOP relies on the fact, that the functional gadgets always return to the dispatcher gadget or a gadget, which then returns to it and that the dispatcher itself always increments the instruction pointer to the next gadget address. The functional gadgets can modify the instruction pointer of the dispatcher to perform conditional branches or load and store data in memory. Also arithmetical and logical functional gadgets can be constructed. The data, which is used as temporary storage of the attack, can also be located on the stack. This adds the convenience, to be able to use push and pop instructions, to easily store and read data from the stack. System calls can be performed, by aligning function argument data on the stack and then execute a gadget, which includes the system call instruction, or directly call the library function, responsible for the chosen system call. After the function returns, the control simply is redirected to the dispatcher, by putting the correct address of the dispatcher on the stack, so that the return instruction can return to the dispatcher.

Chapter 3

RISC-V

Since this thesis is based on the hardware implementation of the RISC-V Instruction Set Architecture (ISA) on a Kintex-7 FPGA, this chapter introduces the main concepts of the ISA.

The RISC-V ISA is a new, fully open source ISA, developed by the University of Berkley [23]. The motivation is to create a new standard ISA, which is highly extensible and transparent. RISC-V is used within multiple research topics and further industrial usage is the goal. The ISA is already in use for education of undergraduate and graduate students at the University of Berkley. The ISA was not only designed for emulations or simulations, but also to be used within real hardware implementations in a highly efficient way. Both 32- and 64-bit implementations are supported. The instruction set is divided into various extensions, including the small base integer instruction set and multiple optional extensions, to allow general purpose software development. The base integer instructions have to be included in every design, while the optional extensions are not necessary. Therefore, custom educational and highly optimized versions of RISC-V can be realized. RISC-V is also designed to support multicore or manycore implementations.

The RISC-V organization provides all documentation regarding the user-level instruction set [1], as well as the privileged ISA specification [2], open source implementation to use on FPGA or ASIC, compiler support and even a port of the Linux kernel to operate on it. Teaching material is also available. In addition to hardware implementation source code, software simulation environments are available too, in order to accelerate the development process.

3.1 Instruction Set Architecture

As mentioned above, the RISC-V ISA is separated in the base integer instruction set and several standard extensions. The extensions have a naming convention that shall be kept. For example, the minimal standard integer instruction set is named RV32I/RV64I. This "I" instruction set is the main instruction set, needed as pre-knowledge for this thesis, so only this one will be described in detail. It is mandatory to be included in every RISC-V implementation. It includes integer computation instructions, as well as load/store and control-flow instructions.

The RV32I and the RV64I architecture uses 32-bit instructions and 32 core registers. However, the number of registers and size of instructions are not fixed, since there are different configurations of the base instruction set existing. For example, the instruction set denoted as RV32E only operates on 16 core registers.

This mandatory base instruction set can be combined with various standard extensions, to enable more functionality of the architecture. The standard extensions can be summarized as follows:

- "M" is the integer division and multiplication extension. With these instructions, values in integer operand registers can be multiplied and divided.

- The extension “A” adds several atomic instructions. They include atomic read/modify and write memory instructions in order to perform inter-processor-synchronisation.
- “F” is the single precision float extension. So it adds float load/stores as well as computational instructions.
- The next possible extension is denoted as “D”. It adds double precision floating point computational instructions as well as load/store instructions.
- Finally, there is also a compressed instruction set extension “C”. This extension operates with an reduced instruction size of 16-bit. Most of the basic instruction set instructions can be exchanged with this reduced instructions in order to take up less memory for program code.

The combination of all standard extensions (IMAFD) is simplified to the identifier “G”. Compilers currently target this default RV64G implementations.

3.2 Base Integer Instruction Set

The base standard integer extension instruction set (IS) includes all basic instructions necessary, to perform simple computations and control-flow operations. This section summarizes the most important RV64I instructions, since we used the 64-bit variant of RISC-V within this thesis. Every RISC-V implementation has to support this basic instruction set.

All RISC-V instructions have a fixed length of 32 bit. This also means, that they are always 4 byte aligned in the memory. Any other non-aligned value of the program counter (pc) would cause an exception.

The three basic classes of instructions are computational instructions, control transfer instructions and load/store instructions. The following sections describe the mentioned instruction types in detail.

3.2.1 Computational Instructions

These instructions perform operations within the Arithmetical Logical Unit (ALU). Only a few basic operations are supported by the RISC-V ISA. All instructions except LUI and AUIPC are available as either register-register or register-immediate operations. The instructions always take either 3 registers or two registers and one immediate as arguments. In the case of three registers, two registers denote the operand values and the third identifies the destination register. In the other case, one register and the immediate value are the operands of the computation, while the second register is used as destination register.

- ADD: Adds two values in operand registers and saves the result in the destination register.
- SLT, SLTU: Performs a comparison and sets the destination register to 1, if the first operand register is less than the second operand register value. There is also an unsigned version of this instruction available.
- AND, OR, XOR: Logical operations, which perform the logical operation on the two operand register values and store the result in the destination register.
- SRL, SLL, SRA: Logical right/left shift and arithmetical right shift.

– SUB: Subtracts two values in operand registers and saves the result in the destination register.

– LUI: Load upper immediate stores an immediate in the upper 20 bits of the destination register and sets the 12 lower bits to zero.

– AUIPC: Adds an upper 20 bit immediate to the current program counter.

There is no explicit MV (move) instruction declared. Therefore, the move instruction is realized as an addition of the source register and zero (ADD rd, rs, 0). A similar rule applies for the NOP instruction. This instruction is assembled as ADD x0, x0, 0. Since the register x0 is hard wired to zero, no state change occurs within processor registers.

3.2.2 Control Transfer Instructions

This instructions are separated into unconditional jumps and conditional branches. The jump instructions are used to perform function calls, while the branch instructions change the program flow with given dependencies.

– JAL (Jump and Link): Used for unconditional jump to a given offset, based on the program counter. The pc is changed to this given value and the return address (pc + 4) is saved in the destination register

– JALR (Jump and Link Register): Indirect unconditional jump that sets the program counter to a value loaded from a chosen register. An immediate value, coded in the instruction, can be added to this value.

– BEQ/BNE: Branch equal and branch not equal take the branch if both given registers are equal/not equal, respectively.

– BLT[U]: Branch less than takes the branch if register 1 is less than register 2. U indicates the unsigned comparison of the two registers.

– BGE[U]: Takes the branch if register 1 is greater or equal than register 2.

Again, some branch instructions need to be realized using other instructions. Therefore BGT, BGTU, BLE and BLEU are available by reversing the operands of BLT, BLTU, BGE and BGEU, respectively.

3.2.3 Load and Store Instructions

Within the RISC-V ISA, only the load and store instructions have access to memory. All other instructions are only operating on registers. Loads and stores work best, if the addresses are correctly aligned for the corresponding data type. However, misaligned loads and stores can also be performed with this basic instruction set, but the designers propose a longer execution time for this types of memory operations. Therefore, double word accesses need to be 8 byte aligned, while a word or half word has to be aligned by 4, or 2 bytes, respectively.

– LD: Loads 64 bit data from the memory into the destination register.

– LW: Loads 32 bit data from the memory, sign extends it to 64 bit and stores the result in the destination register.

– LWU: Loads 32 bit data from the memory, extends it with zero to 64 bit and stores the result in the destination register.

- LH/LHU: Defined in the same way as LW/LWU but for 16 bit data loads.
- LB/LBU: Defined in the same way as LW/LWU but for 8 bit data loads.
- SD/ SW/ SH/ SB: Stores 64, 32, 16, 8 bit of data in the memory.

3.3 Registers and Calling Convention

The RISC-V ISA defines the usage of 32, 64-bit registers. The register x0 is hard wired to zero and therefore can not be assigned with any other value. Another user-visible register is the pc register. The list of registers and their general usage within the Application Binary Interface (ABI) is shown in Table 1.

The compiler uses the ABI names of the registers according to this table. The reason, why the temporary registers, named si and ti, are not grouped together, is that there exists a ISA version (E) with a reduced number of the registers. In this version, only 16 registers are available. The registers are grouped in a way that the registers x0 to x15 include all necessary types of registers. Therefore, no second definition for the reduced version is needed.

The calling convention of the RISCV-ISA specifies the usage of the described registers during program execution. This includes the stack operations, function calling routines and the registers involved during this process. The registers are divided in a class of caller saved registers and a class callee saved registers. Registers that are saved by the caller, are volatile across calls and therefore have to be saved, if later used by the callee. On the other hand, callee saved registers are persistent across calls and have to be saved by the callee, if used. In this architecture, the stack grows downward. Unlike most systems, the RISC-V ISA does not include push or pop instructions for stack operations. Instead, the compiler has to take care about the stack manipulation. The addresses are always calculated as an offset, relative to the current stack pointer. Then the registers to save/restore are stored/loaded, using the provided load and store instructions.

RISC-V provides two different instructions in order to perform function calls. These instructions are JAL and JALR. As previously described, JAL is the direct jump and JALR is the register indirect jump. Therefore, JALR is only used in the case of function pointers, v-tables and so on. Also long jumps have to be done with this instruction. In that case, a program counter relative address is generated with the AUIPC instruction and stored in a temporary register. Then another offset can be added to this value of the register, given within the JALR instruction. Both jump instructions store the return address, which is the current program counter added by 4, in the register ra. At this moment, the return address is only stored in this register and has to be copied on the stack, if later needed. This is the case, if the called function calls again another function, so the return address must be restored.

Whenever possible, function arguments are passed entirely in registers. Integer function arguments are passed in registers a0-a7. On the other hand, floating point arguments are passed in the registers fa0-fa7. The return values are returned in the registers a0 and a1.

3.4 Privilege Levels

The RISC-V ISA provides four different privilege levels. The privilege levels control the accessibility of CSRs (Control and Status Registers) and also which instructions can be executed in the corresponding level. The four privilege levels are machine-, hypervisor-, supervisor- and user-mode. Table 2 lists all modes together with the corresponding encoding for the privilege levels. This encoding is used system wide, if any reference to the privilege levels is given.

In machine mode, the processor executes the highest privilege level. This is also the only privilege level, a RISC-V implementation has to provide in order to perform all necessary operations. Usually, the hardware access layer (HAL) is running in this privilege level. All control registers and instructions are accessible and available in this mode. The hypervisor mode is a placeholder for future virtualization operations. In this case, the host machine is running in hypervisor mode, while the virtualized machine is running in supervisor mode. Currently, operating systems are operating in supervisor mode, in the example of the Rocket core. User applications are executed in user mode. In order to access hardware modules, system calls need to be performed, to increase the privilege level. In user mode, privileged instructions and most CSRs are not accessible.

A processor with an implementation according to the RISC-V ISA, has to implement at least one privilege level. This mandatory level is the machine mode. If two privilege modes are implemented, both machine and user mode is mandatory. With three levels the machine, user, and supervisor mode is required. When four privilege levels are implemented, all introduced modes has to be supported.

3.5 Control and Status Registers

RISC-V provides instructions and rules for Control and Status Registers (CSR). These registers are used, to control the processor state, store context switch information, as well as global timers. The instructions, which read, write or modify the CSRs, are within the privileged class. As a result, these instructions are only allowed to be executed, if the processor runs within the correct privilege level. The instructions themselves are not associated with a privilege level. This means, that it depends on the type of the control and status register, which is about to be accessed with the instruction. If the instruction attempts to access a register, which requires a higher privilege level, the processor will trigger a trap. However, a higher privilege level can access registers, which require a lower privilege level. Also, the registers themselves define, whether they are writeable or readable and again the processor triggers a trap, if the operation is not allowed for the accessed register.

The RISC-V ISA offers a few instructions, which are accessing the CSRs, which allow convenient modification or reading of values.

- CSRRW (Atomic read/write). This instruction type exchanges the values in the CSR with the values in integer registers. This means, that the original value of the CRS is read and stored in register rd. The value given in rs1 is then written to the selected CSR.
- CSRRS (Atomic read and set bit) also reads the current value of the CSR and stores it in register rd. Other than the CSRRW instruction, the rs1 register defines the bits to be set afterwards in the CSR. This means, that only the bits which are high in the register rs1 are set, while the other bits in that CSR are unaffected.
- CSRRC (Atomic read and clear bit) is similar to CSRRS. The current value is stored in the destination register rd. However, the register rs1 defines the bits to be reset

in the CRS, after the read is performed. Every bit that is high in register rs1 will be cleared (set to zero), while the other bits stay unaffected.

– The instructions CSRRWI, CSRRSI and CSRRCI are behaving similar to the previously listed instructions. The difference is, that the value written to the CSR is not obtained from an integer register rs1, but from a zero-extended 5 bit immediate, coded in the field, where the number of rs1 is specified in the other instructions. This means, that there is no need for an additional register, if only a 5 bit value needs to be modified within the register.

Also important to mention is, that if rs1 within the instructions CSRRW, CSRRS, CSRRC is equal to x0, no write is performed at all. The assembler pseudo instruction CSRR, to only read values from a CSR, is constructed from CSRRS, by coding rs1 = x0. The pseudo instruction to only write values to the CSR, is constructed from CSRRW by coding rd = x0.

As already mentioned, the registers themselves specify the access permissions. This is archived by using the top 4 bits, within the 12-bit address space of the CRSs. The bits csr[11:10] specify whether the register is writeable and readable (00,01,10), or only readable (11). The bits csr[9:8] define the minimum privilege level that is allowed to access the CSR. The value 00 encodes the user mode, 01 the supervisor, 10 the hypervisor and finally 11 encodes the machine mode. If a read only CSR is attempted to be written, the processor issues an illegal instruction trap. The same applies, if a CSR is accessed from a too low privilege level.

Table 3 shows a list of the currently allocated CSRs for the different privilege levels. They are divided between standard CSRs and non-standard CSRs. If more information about the specific allocations of the CSRs is required, please refer to the official privileged ISA documentation [2]. The official RISC-V privilege specification delivers a list of standard CSRs, which shall be existing in every implementation of the RISC-V ISA.

3.6 Implementations

There are a couple of cores, which are already implementing the RISC-V ISA. The University of Berkley developed RISC-V core generators, which allow to construct different types of RISC-V compatible cores [32]. The smallest architecture is the Z-scale [21] core. It is intended for embedded systems and follows a similar strategy as ARM M0/M0+/M3/M4 processors. The Rocket core is the main multi-purpose processor. The generators are written in Chisel [3], which is a new hardware description language, based on Scala. The vscale [20] architecture is entirely developed with Verilog and is equivalent to the Z-scale implementation, without the use of Chisel. The lowRISC [14] team aims to develop an open source commercial development board. The SoC is based on the previously mentioned Rocket core. The work of the this team is described in detail in Chapter 7.1.

RISC-V Name	ABI Name	Usage Description	Saver
x0	x0	Always zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	Callee
x5	t0	Temporary	Caller
x6	t1	Temporary	Caller
x7	t2	Temporary	Caller
x8	s0	Temporary	Callee
x9	s1	Temporary	Callee
x10	a0	Function argument/ Temporary	Caller
x11	a1	Function argument/ Temporary	Caller
x12	a2	Function argument/ Temporary	Caller
x13	a3	Function argument/ Temporary	Caller
x14	a4	Function argument/ Temporary	Caller
x15	a5	Function argument/ Temporary	Caller
x16	a6	Function argument/ Temporary	Caller
x17	a7	Function argument/ Temporary	Caller
x18	s2	Temporary	Callee
x19	s3	Temporary	Callee
x20	s4	Temporary	Callee
x21	s5	Temporary	Callee
x22	s6	Temporary	Callee
x23	s7	Temporary	Callee
x24	s8	Temporary	Callee
x25	s9	Temporary	Callee
x26	s10	Temporary	Callee
x27	s11	Temporary	Callee
x28	t3	Temporary	Caller
x29	t4	Temporary	Caller
x30	t5	Temporary	Caller
x31	t6	Temporary	Caller
pc	pc	Program counter	-

Table 1: RISC-V registers, defined in both RISC-V and ABI formats ([1] Table 20.2).

Level	Encoding	Name
0	00	User Mode
1	01	Supervisor Mode
2	10	Hypervisor Mode
3	11	Machine Mode

Table 2: Encoding and naming of RISC-V privilege levels ([2] Table 1.1).

Read/Write [11:10]	Privilege [9:8]	Address Range	Use and Accessability
00	00 (U)	0x000-0x0FF	Standard read/write
01	00 (U)	0x400-0x4FF	Standard read/write
10	00 (U)	0x800-0x8FF	Non-standard read/write
11	00 (U)	0xC00-0xCBF	Standard read only
11	00 (U)	0xCC0-0xCFF	Non-standard read only
00	01 (S)	0x100-0x1FF	Standard read/write
01	01 (S)	0x500-0x57F	Standard read/write
01	01 (S)	0x580-0x5FF	Non-standard read/write
10	01 (S)	0x900-0x9BF	Standard read/write shadows
10	01 (S)	0x9C0-0x9FF	Non-standard read/write shadows
11	01 (S)	0xD00-0xDBF	Standard read only
11	01 (S)	0xDC0-0xDFF	Non-standard read only
00	10 (H)	0x200-0x2FF	Standard read/write
01	10 (H)	0x600-0x67F	Standard read/write
01	10 (H)	0x680-0x6FF	Non-standard read/write
10	10 (H)	0xA00-0xABF	Standard read/write shadows
10	10 (H)	0xAC0-0xAFF	Non-standard read/write shadows
11	10 (H)	0xE00-0xEBF	Standard read only
11	10 (H)	0xEC0-0xEFF	Non-standard read only
00	11 (M)	0x300-0x3FF	Standard read/write
01	11 (M)	0x700-0x77F	Standard read/write
01	11 (M)	0x780-0x7FF	Non-standard read/write
10	11 (M)	0xB00-0xBBF	Standard read/write shadows
10	11 (M)	0xBC0-0xBFF	Non-standard read/write shadows
11	11 (M)	0xF00-0xFBF	Standard read only
11	11 (M)	0xFC0-0xFFF	Non-standard read only

Table 3: Allocation of standard and non-standard CSR within the RISC-V ISA ([2] Table 2.1). The address ranges are given for the various privilege levels and accessing modes.

Chapter 4

Tagged Memory

Previously we described, how code-reuse attacks are executed and that the buffer overflow vulnerability is exploited. We developed new countermeasures for this kind of attacks. This countermeasures are called tagged memory policies. Tagged memory is the basis for the implemented security policies, in order to enforce protection against code-reuse attacks in general. Tagged memory is a method, to perform fine grained memory protection, by adding tags to existing data values within the main memory. In this section, we introduce the concept of tagged memory, pinpoint various existing implementations and the evolution of the use case of tagged architectures. A detailed description of the mentioned security policies will follow in Chapter 5.

4.1 Overview

All architectures supporting tagged memory, have in common, that the data words in the memory will be extended with additional meta information, called tags. Note, that there is a difference to tags, which are bound to the address space. For example the X-xor-W policy denotes, which pages can be executed or written to. This also applies some form of tags, but this tags are only defined by the address. The tagged architecture, which we use in this thesis, uses tags, that are bound to the data within the memory. As a consequence, tags also reside in the main memory and are assigned to given data words by a defined rule. By default, tagged memory architectures do not have a defined purpose. Therefore, rules, which we also call policies in this thesis, are necessary to perform useful tasks. These can be either defined in software, or within the hardware. The usage of tags and their size is varying a lot for the given purpose. The use of tagged memory opens a great variety of possible use cases, with different kind of goals. Most of this use cases can not be solved by software, or only with a significant performance and complexity overhead.

The two basic possibilities to arrange data and tags in memory are shown in Figure 5 and Figure 6. It is either possible to extend the word size of the processor with the size of a tag, or to store the tags in a separate region in the main memory. The first strategy would require to fetch data plus tag with one single memory access and handle the first part of the loaded word as data and the second as tag. The second approach does require an additional load operation on the tag segment in order to fetch data plus tag, but does not require changing the word size in the processor which makes the implementation more easy in most cases. Although this approach requires significantly more memory accesses, the overhead can be compensated with a tag caching strategy.

Another side effect of tagged memory is, that it naturally requires additional space in the main memory. In the worst case, the tag can be as big as the data word size, so the effective memory for data and code will be the half of the available RAM size. The goal of tagged architectures is, to achieve additional, powerful functionality by using the smallest possible tags sizes.

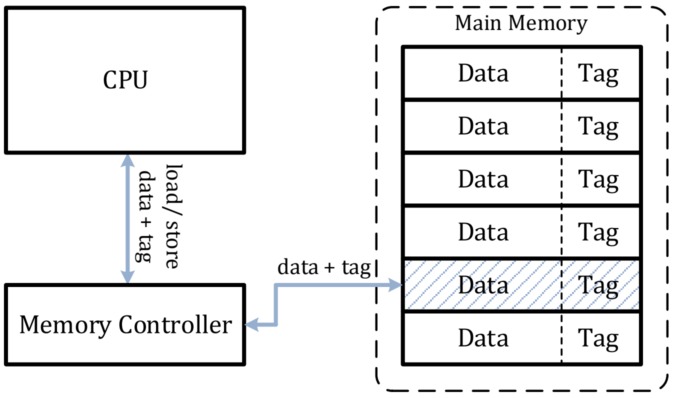

Figure 5: Tagged architecture with tags and data at the same data line.

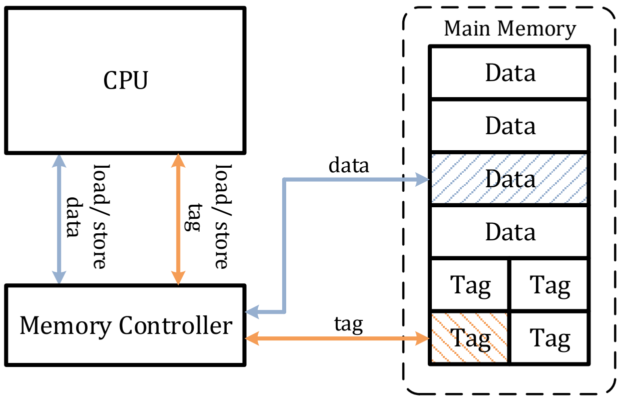

Figure 6: Tagged architecture with tags and data located in seperated areas in the memory.

4.2 Tagged Architectures in History

The idea of tagged processor architectures is existing since the early days of computer research. Back then, they where used to ease hardware debugging or increase the efficiency of the assembly code, but did not have a focus on security at all. The main idea was, to bring more functionality to the hardware itself, in order to decrease the complexity of software. Also, computations in hardware are much more efficient than within software.

One example for a tagged architecture, that was used for debugging purpose is the Rice Computer R-1, developed in 1959 at the Rice University, Houston, Texas [31]. They added two tag bits to each data word and allowed programmers to set this tag bits. The computer was able to trap on this tag bits when set. Also, other architectures where used for similar purposes. For example, some programmers found that the parity bit of the IBM 7040 could be set by software [33]. This feature was then used in order to identify uninitialized memory. The next architecture, that used tagged memory, was the Boroughs B5000 machine, developed in 1960 [4]. This machine had one tag bit, that was used to identify memory regions. For example it was possible to distinguish between code and data, so that data was not allowed to be executed and code to be arguments of computations. Therefore a program could not modify itself at runtime.

Later, tags were used to define data types at runtime. Therefore, every byte in memory was assigned with a tag, which held information which data type or structure it consisted of. Using this technique, the complexity of the assembly could be reduced significantly, since no type dependent instructions had to be programmed. The hardware could decide how to handle arithmetic operations by simply evaluating the given tag for the data. One of the first computers to use this technique was the Rice Computer R-2 [8] and the Boroughs B 6500/7500.

In 1970 MIT developed the LISP machine. As the name suggests, it was highly optimized to execute applications, written in the programming language LISP. The machine also incorporated dynamic runtime typing and thus, generalized the assembly. These operations, as well as checks and conversions, were performed in parallel, so it speeded up the entire execution. E. Feustel summarizes tagged architectures and their advantages and also elaborates the topic of hardware level typing even further in his paper "On The Advantages of Tagged Architecture" [9].

While tagged architectures were not used in commercial products since then, they nowadays get more popular again for security topics. We cover recent work which aim to prevent code-reuse attacks in Chapter 6.

Chapter 5

Tagged Memory Policies

This thesis proposes a countermeasure against the previously described code-reuse attacks. The attacks are based on buffer overflow vulnerabilities. The goal is, not to allow the attacker the initiation of one of these attacks without changing code of applications. Also, there are no compiler changes involved as with previous software- and hardware-based solutions. All protections are done entirely in hardware, so no bigger modifications have to be performed on the software side. Tagged memory allows to implement a fine granular protection within the core, by applying tagged memory security policies. The following chapter introduces the theoretical principles of our proposed countermeasures. The implementation aspect is covered in Chapter 7. Note, that all elaborations within this chapter are done in the context of the RISC-V ISA. This is necessary, to allow a good description of the behaviour of instructions, when tagged memory policies are applied. First, the memory usage for the introduced countermeasures is elaborated, followed by the architectural design principle. Afterwards the security policies, its rules and protection schemes are explained in detail.

5.1 Memory Overhead

In order to support the tagged memory policies, the hardware has to support various features, as the tagged memory support in general. The policies can only be implemented on tagged architectures. The architecture, suggested in this thesis, operates on separated data and tag areas within the memory. The biggest drawback of tagged memory architectures is, that they require a significant amount of additional space within the main memory, in order to store the tags. Therefore, this section elaborates the needed additional memory for our architecture. Even though the policies have a big impact on the security, the memory overhead was kept low with our policies. The introduced policies require only two tag bits per 64-bit data. Therefore, the ideal implementation of the policies only requires 3.125% of the given available main memory size.

The exact amount of memory, required for the tag partition, is calculated with Formula 1. M_t is the tag partition size in bytes, M_d is the effective memory, used for data in bytes and $Tagbits$ denotes the count of tag bits per 64-bit double word.

$$M_t = \frac{M_d}{8} * \frac{Tagbits}{8} \tag{1}$$

Since the available physical memory is split into separate data and tag partitions, an extra factor needs to be taken into account, when the ratio between the partitions is calculated. The less data is available, the less tags are needed in the system. The usage of Formula 1 with M_d as the size of the available RAM would result in a tag partition size, that is required if the whole RAM is used for data. But since the data and tags have to fit into the same available memory, the addition of both would lead to a amount of memory, which is bigger than the RAM. However, the correct data partition size can be calculated with Formula 2. $Memsize$ denotes the total available physical RAM size in bytes. The result leads to sizes of the data and tag partition, that are ideally fitting for the given available main memory size.

$$M_d = \frac{Memsize}{1 + \frac{Tagbits}{64}} \tag{2}$$

However, in this work we implemented four instead of the two necessary tag bits per 64-bit data. The reasons for this result from the architectural constraints of the used chip architecture. It would have been an implementation overhead to support tag sizes smaller than four bits within the processor. Therefore, the total size of the tag partition is 6.25%. The two additional tag bits can be used by the user, to perform, for example, dynamic taint tracking, as described later in Section 5.5.

5.2 Tag Operation

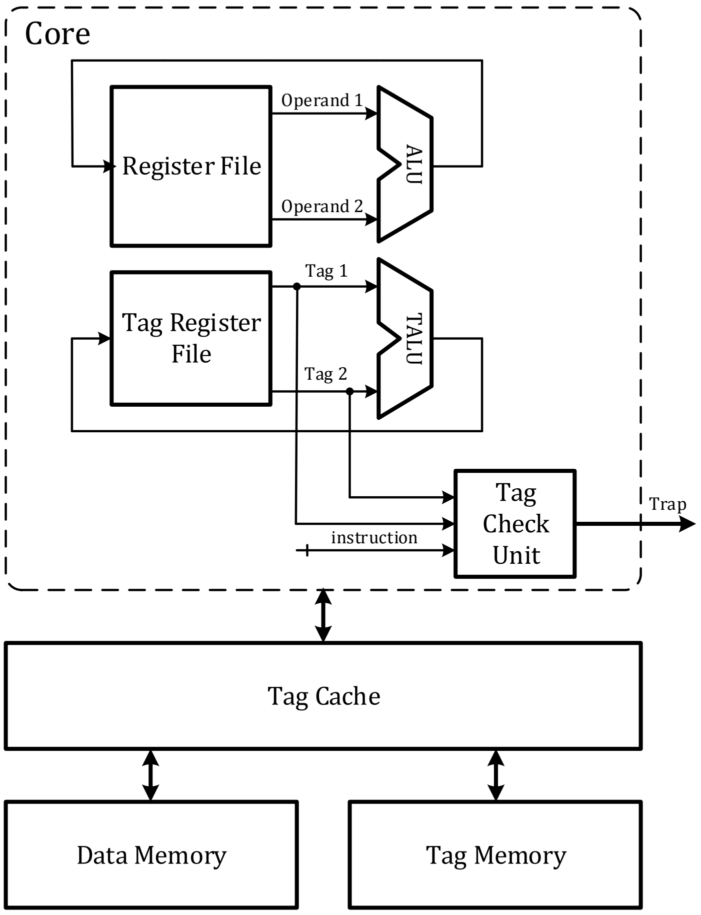

Figure 7: The overview of the tagged architecture, including the tag cache to improve efficiency.

This section describes the architectural constraints, in order to implement effective countermeasures against code-reuse attacks with tagged memory. This constraints apply to any system, designed with the RISC-V ISA and therefore are not implementation specific.

All instructions executed within the processor, need to alter tags in order to apply the tagged memory policies. This means, that tags have to be propagated not only in the memory, but also directly within the core registers. The overall system design is shown in Figure 7. The system is operating using a memory holding tags and another memory containing the data. As described before, multiple configurations and alignments of tags in the memory are possible. In order to describe the tagged security policies, only the main concept is explained.

On every data load or store operation, not only the data has to be transferred, but also the corresponding tags. This requires a manager for this matter. The manager is called tag cache within this thesis. It coordinates the memory access as the last instance within the processor. Tags and data are sequentially transferred, when the processor requests the memory access. The tag cache combines those two memory accesses into one response when data is read from memory, or splits them into the separate memory accesses on data write operations. This is done by calculating the location of tags to the corresponding data at the given address. Also, the tag cache reduces the amount of memory accesses, caused by tag misses. Not every memory access has to perform a second memory access, since the tag cache separately stores the tag values for a wide range of addresses.

The core itself receives the data words including tags on every memory load operation. This tags are stored in a separate tag register file, which is aligned with the core data registers. They are accessed simultaneously, when a register load or store is performed, using the same address bus. The tags need to be propagated to the destination registers. This is done by applying given propagation rules, defined for each instruction. While the ALU performs well defined logical or arithmetical operations on given operands, the tags are calculated in a different way for the same instructions. Therefore, a tag ALU is introduced. It operates in parallel to the ALU, but takes the tags of the operand registers as input. The tag propagation rules are defined by the tagged security policies. The result is stored in the tag register file after each operation.

The enforcement of the proposed policies, is performed by the new introduced tag check unit. Its inputs are the tags of the operand registers and the current instruction. Based on the enforced policy, the correctness of the input tags with the given instruction is checked. If the tag value is not matching to the required value for the instruction, defined by the policy, a trap is triggered. The trap results in an immediate jump to the exception handling routine (written in RISC-V assembly). Therefore, the program counter is set to the according value. This routine can then check the cause of the trap, using the *mcause* register, defined within RISC-V. The software is then able to decide, what actions to perform, if the tag trap occurred. This could be either a debug tag trap or a trap caused by a breach of the security policies.

The system also requires the possibility to modify only the tag portion within a data field by the user. The reason for this is, that some policies, which will be introduced in this chapter, require support by the Linux kernel. Also, the user gets the option, to implement her own tagged policies and tests. This is accomplished by adding two instructions to the instruction set, which modify the tag portion of chosen memory locations. These instructions are storetag and loadtag and are derived from the general load and store instructions within the RISC-V instruction set.

5.3 Return Address Protection

The return address protection serves as countermeasure against return-oriented programming attacks. In the most common scenario, the attacker overwrites a buffer on the stack and changes the return address to an arbitrary value. As a result, the control flow is altered. In order to prevent the attacker to apply this technique, the return addresses, which reside on the stack, have to be protected. This implies, that every modification of the memory locations, where a return address is stored, has to be assumed malicious, since this usually does not happen during normal program execution. With tagged memory, the meta information given by the tag, is used to identify valid and invalid return addresses.

The proposed countermeasure protects return addresses within the memory and registers from being modified. After modification, return addresses are not valid anymore. This not only applies to modifications caused by the attacker, but also when the program itself overwrites its own return addresses by accident. Therefore, wrong usage of buffers within development is recognized, respectively. The return address policy detects potential changes of the control flow and triggers a trap, if this is the case. A benefit of this policy is a high compatibility with existing programs. However, special handling is required within the Linux kernel, since it is performing modifications of return addresses within context switches. In this cases, incorrect traps can occur. In Section 7.4, this incorrect traps are described in detail and the fix is performed.

In order to detect potential attacks, a couple of techniques are applied. Therefore, the return address protection introduces the RA (return address) tag bit. As mentioned before, this bit is propagated within the core and the memory with a set of given rules. The RA bit is only set to “1” by instructions, which perform function calls. The value “1” denotes a valid return address. At the time, the processor executes the JAL or JALR instruction, as defined in the RISC-V ISA, the return address is saved in the ra(x1) register. Only if this instruction is executed, the RA tag bit is automatically set to “1” by the core for this register. This ensures, that no other instruction is able to set this bit too. The basic idea is, that a return on any return address is only possible, if this tag bit is set. Functions always return with the RET instruction. This instruction is only an alias for the JALR instruction within the RISC-V ISA. As a result of this fact, the hardware must ensure, that this type of instruction is identified and the check of the RA tag bit is performed only in this case. The JALR instruction, that is used for function return operations, always has the ra register as argument of the indirect addressing. That introduces the possibility to distinguish an ordinary function call, with indirect addressing, from the function return. Therefore, the tag check unit receives the instruction as input, which includes the chosen registers. If the ra register is used and the instruction is JALR, the RA tag bit is checked. If its value is still “1”, the return address is valid and no trap is triggered. The processor continues with the normal execution. Otherwise it will trap and let the software decide the further steps. It could result in a kernel panic or a termination of the potentially attacked user space application. The traps occur within every privilege level of the RISC-V ISA. This leads to a protection of the whole system, including machine mode handlers within the bootloader code, Linux kernel as supervisor and user applications in user mode. The reset of the RA tag is performed by the tag ALU and the data cache, with given rules. Every modification or computation with a register, that holds the RA tag is malicious and therefore causes the RA tag to reset to “0”. However, there are non trivial cases, in that this tag bit has to be reset and handled in a different way.

With only basic propagation rules within the core, the attacker is still able to copy return addresses as a whole. This is caused because of the fact, that a move instruction has to copy the data and tag as it is to a different register, in order to keep the compatibility. If this scenario occurs, also the source register still contains the tag value after the operation. Same applies to memory load and stores. An attacker could copy a valid return address to a different location in memory. However, in this thesis we decided to not propose counter measures against this issue. The reason is, that a full double word copy of the return address always results in a valid return address, which was already existing within the application. Therefore, an attacker could only execute functions within the application if he is able to find an entry point and perform the copy. The benefit, this mechanism might give the system in a security perspective of view, is not worth the disadvantages, which it causes. The disadvantage would be the creation of an inconsistency within the memory and core registers. Data, that is moved or loaded from the memory would alter its source register or memory location after the operation. Not only the target register would be altered, but also the source register. A second load of this data would result in a different value of the tag. All tags can only be assumed valid after the first load of the value. Also, it would require an additional store operation, that resets the RA tag on every load instruction. As a consequence, the performance of the system, as well as the compatibility is decreased.

Another security issue this policy is targeting, is the partial modification of the return address. The attacker could perform partial stores on memory locations with a valid return address. A partial store operation is every operation, which stores less than 64-bit of data at once. There are 32, 16 and 8-bit stores existing within the RISC-V ISA. By performing a partial store on the return address, the RA tag bit is set within the memory, but not within the register the store is performed with. Therefore, a straightforward implementation of the memory behaviour for partial stores, would still output another valid return address. The problem is caused by the fact, that tags are only stored per 64-bit double words. If the mechanism combines the stored tag value of the data with the new tag of the partial store operation, the tag bit is preserved. Of course, this drawback only occurs, because there is no tag field for every byte in memory within our architecture. As a result, the attacker could perform a byte wise store of the return address and alter it to an arbitrary value, without losing the return address tag. This always happens, if the value of the input buffer is byte aligned. We provide a solution for this issue, by expanding the propagation rules to memory operations. At a partial store instruction, the return address tag bit of the previously saved data is always cleared. Partly modifications of a return address in memory with arbitrary values are therefore forbidden and result in invalid return addresses.

A similar problem occurs for partial copies of return addresses. This means, that the attacker is still able to perform partial loads on any return address, available in the memory, and directly perform a partial store of this data to the attacked return address. This way, a return address can be constructed out of copied byte values of other return addresses. If the attacker can find appropriate values within the memory, this is be a potential threat. Therefore, partial load operations on return addresses shall not result in a valid return tag bit. With this strategy, it is not possible to perform this type of return address modifications. No valid return address can be constructed out of parts of several valid return addresses. Only full loads are allowed to return a valid return address tag bit.

5.3.1 Propagation Rules

The following section describes the necessary propagation rules for the return address protection policies. This rules only apply to the return address tag bit. Therefore, the rules for the other available tag bits are listed in the sections of the other policies. The tag ALU applies this rules for every instruction which is operating on processor registers. Also, memory writes and stores have to apply own rules, if the data is not 64-bit wide.

All 32, 16 and 8-bit memory operations have to manipulate the tags in an appropriate way for the given policy. Table 4 lists all necessary rules for instructions, which alter the return address tags. RA(rd) denotes the return address tag of the destination register of the instruction. RA(mem) denotes, that the return address tag for the given memory location that is altered.

Arithmetical and logical instructions always clear the return address tag. This ensures, that no calculation with registers can modify the return address, without losing the validity of the return address. The move operation copies the data and the tag from the source register without modification to the destination register. This is necessary, to allow more complex code that requires to copy the return address into another temporary register. Both direct and indirect jump and link instructions set the RA tag bit within the register ra. This way, a valid return address is generated on every function call. Full double word store operations, store the tag value of the source register without modification into the memory. The same applies for full load instructions. The tag is not altered and copied to the destination register. This concludes the rules for the basic return address protection. However, two additional rules are added, in order to prevent partial copies and partial modification.

When a partial store is performed, the RA tag is always cleared from the source register and not combined with previously stored tags. Therefore, the tag is overwritten. As a result, the partial store for the RA tag bit behaves the same, as the full store instruction and therefore does not allow modification of the return address. To prevent partial copies, partial load instructions always return “0” for the RA tag bit. It is not possible, to load partial values from memory, that include a valid return address tag, anymore. Return addresses are always 64-bit aligned in normal cases, and so this rule does not cause compatibility problems.

Loads on addresses within the IO space always introduce the value “0” for the RA tag bit.

Instruction	Example	Propagation
Arithmetic	add rd, rs1 , rs2	RA(rd) = 0
Logical	or rd, rs1, rs2,	RA(rd) = 0
Move	add rd, rs, 0(imm)	RA(rd) = RA(rs)
Jump [dircet/indirect]	jalr rd, rs, imm	RA(rd) = 1
Store [full]	sd [rs1], rs2	RA(mem[rs1]) = RA(rs2)
Load [full]	ld rd, [rs]	RA(rd) = RA(mem[rs])
Store [partial]	sb [rs1], rs2	RA(mem[rs1]) = 0
Load [partial]	lb rd, [rs]	RA(rd) = 0
Load [IO]	ld rd, [rs]	RA(rd) = 0

Table 4: Return address tag propagation rules.

5.3.2 Example Attack

The following section shows some attack situations and operations on return addresses, which are protected by the proposed policies. We provide code snippets, showing this situations, in the following section. All examples show the different types of modifications, that can be performed on an return address, stored on the stack. Note, that the functions, which cause the malicious return addresses need to call another function. The compiler will optimize the code to not storing the return address on the stack otherwise. They would be only stored within the ra register and therefore, the attack could not be demonstrated. All examples initialize a array of long variables with the size of 10.

- **Basic return address overwrite**

  With this policy, the integrity of the return address on the stack is protected. The code is shown in Listing 2. In the example, the function *performAttack* initializes an array on the stack. If the array is overwritten, the other values on the stack can be altered. In this case, the array index 12 is exactly at the same position as the return address on the stack. The address is then overwritten with the address of the *attack_successful* function. Once the RET instruction is performed, this function would be called and the attack would be successful. However, with the policy enforced, at this time a trap is triggered and the program is terminated.

```
void attack_sucessful()
{
  printf("Attack␣Sucessful!\n");
}

int performAttack(int testvar)
{
  int y = 1;
  y += testvar;
  long x[10];
  anotherTestFunc();
  x[12] = &attack_sucessful;
  return 0;
}
```

Listing 2: Basic return address overwrite.

- **Partial Modification**

  In this example, presented in Listing 3, the return address is only partly overwritten. Therefore, one byte of data is copied to the address of the return address on the stack. As a result, the processor performs the SB memory store instruction. Since the RA tag within the register of this data byte is zero, the RA tag is also reset in the resulting memory location. As a result, the return address is not valid anymore, when the RET instruction is performed. A tag trap is triggered.

```
int performAttack(int var)
{
  char test[2];
  test[0] = 0xdE;
  test[1] = 0xAD;
  int y = 4;
  y += var;
  long x[10];
  anotherTestFunc();
  (*(char*)(x + 12)) = test[0];
  return 0;
}
```

Listing 3: Partial modification of the return address.

Note, that this case is the usual way, the ROP attack is initiated. The code snippet in Listing 4 demonstrates the vurnarability within an application. The return address is byte wise overwritten with the data read by *scanf()*. If the input is bigger than the actual size of array x, the return address is overwritten. In this case, no sanity checks are performed and the attack can be initiated. With the return address policy, this is detected and not possible anymore.

```
int performAttack(int testvar)
{
  int y = 4;
  y += testvar;
  long x[10];
  scanf("%s", x);
  return 0;
}
```

Listing 4: Example of a standard vulnerability.

– **Partial Copy**

This time, a part of any valid return address is copied to the attacked return address. Therefore, a new valid return address can be assembled out of existing return addresses within the stack. Therefore, one byte of the return address is loaded in a temporary variable. Then it is copied to different bytes within the return address on the stack. Since the source byte also has a valid RA tag, the result register would also keep the valid tag with a straight forward implementation. However, with the proposed security measures, this technique also leads to an invalid return address. The trap is triggered when the RET instruction is performed. Note, that the example in Listing 5, does not actually create a new address, which would be useful for the attacker. It is just a demonstration of the process.

```
int performAttack(int var)
{
  char test;
  int y = 4;
  y += var;
  long x[10];
  anotherTestFunc();
  //Assemble return address out of valid return address bytes
  test = (*(char*)(x + 12));
  *(((char*)(x + 12)) + 1) = test;
  *(((char*)(x + 12)) + 2) = test;
  return 0;
}
```

Listing 5: Partial copy of the return address.

5.4 Function Pointer Protection

While return-oriented programming attacks are defended with the previously described return address protection, the system is still vulnerable to jump-oriented programming attacks. These attacks rely on the ability of modifying a function pointer within an application. Mostly, function pointers can be overwritten by overflowing a buffer on the heap. The attacker alters the value and when the call of this function is performed, an arbitrary code is executed. In order to provide a countermeasure, the integrity of this function pointers has to be ensured. This is more complicated, than the return address protection, since the function pointers behave as any other data within the memory and modifications are allowed in general. Detection, if a function pointer is valid or not, therefore is not possible using a straight forward method. Tagged memory architectures are capable to solve this issues and a few solutions where proposed by other research teams. However, most of them require compiler support. The compiler marks all function pointers and if the pointers are altered, the tag is reset and invalid. This also introduces the problem, that function pointers can not be altered by the application itself and in general, function pointers are hard to identify at compile time. In this thesis, we introduce a solution of protecting the integrity of function pointers, without any compiler modifications. Everything is done automatically and purely in hardware at run time.

Our countermeasure is based on a blacklisting approach for identifying user data. In general, the data is marked as trusted or untrusted. Data that is available at compile time, or modified by the application without user interaction is assumed trusted. However, data that is read from any kind of IO device is assumed untrusted. As a result, jump instructions which use addresses that where modified via IO are not allowed. This way, it is assured that attacker modified data can never be used as an operand of a jump instruction. The data is first marked untrusted from the processor, when it was read from a IO device. This information is then propagated within the processor and memory as with dynamic taint tracking. Dynamic taint tracking is a mechanism, used by researchers in order to analyse the spreading of untrusted data within a system. Once untrusted data is in some way connected with trusted data, the result is also untrusted. Mostly, this techniques are only implemented in software and are not existing in the final product. However, we implemented dynamic taint tracking within hardware using tagged memory and enforced this security policy. Note, that we furthermore use the terms valid and invalid instead of trusted and untrusted. This is more convenient, since the terms are used to define, if jumps are valid or invalid, respectively.

The policy introduces the INV (invalid) tag bit to identify valid or invalid data. If set to logical “1”, the data is invalid. The tag is only set to invalid in the destination register, if the RISC-V load instruction is executed and the address is located within the IO space. Therefore, a program execution, not relying on any IO device access, always contains valid data. As mentioned, the load with any address within the IO space, results in a INV tag bit, indicating that the data is invalid in the destination register. This tag bit is then propagated within the system using a set of defined rules. In general, computations with one or more registers, containing invalid data, will result in another invalid destination register. If the JALR instruction with invalid data in the address register is executed, a tag trap is triggered. As with the return address protection, the tag check unit verifies the validity of every register, which is used for the JALR instruction. The tag ALU on the other hand, takes care of the propagation of the INV tag within the operand registers. The traps are also caused within every privilege level of the RISC-V ISA. To ensure that the attacker can not reset the INV tag bit, the hardware has to ensure, that there is no operation or instruction, that will clear the bit. Only if the register is overwritten with valid data, the INV tag within this register is set to "0". However, the tag also needs to be cleared within the ra register, if a jump is performed. This ensures, that return addresses never include the INV tag bit, set to "1". This avoids spurious traps and also causes no security issue, since the return address is protected with the previously introduced return address policy. The INV tag bit is also stored within the memory, using the tagged memory architecture. With this policy, the attacker is not able to forge any kind of address, which is later used in the application, in order to change the program flow.

However, there exists a drawback of this solution. While stand alone applications won't cause any trouble, kernels and bootloaders will. The reason is, that every user application, that shall be run on the system, needs to be loaded from the hard drive by these two program types. At the moment, the binary is loaded from the disk, all data that is transferred, will be marked as invalid. Therefore, it is impossible for the program to run, if any function pointers are included. In order to solve this issue, it is necessary to add a possibility to enable the system to load binaries. The binaries need to be marked as valid, when they are loaded from the disk. This means, that in this case, software support is required. There are two options to achieve this behaviour. It is either possible, to use the settag instruction after the binary is loaded, in order to clear the INV tag. The other possibility, is to disable the invalid tag generation within the processor, while binaries are loaded from the IO. The first option would introduce a performance overhead, since the settag instruction has to be called for every double word of the binary. This is comparable to a memset. The other solution introduces possible race conditions, which can introduce security issues within the kernel. However, this can be fixed with some future work. We decided to provide a mechanism to switch off the tag generation while loading binaries, since we didn't want to decrease the efficiency of the system. All this modifications to the software don't introduce security issues, if done properly, since they are implemented within the bootloader and kernel, which is not assumed malicious. We give a detailed description of the implementation of this fix in Section 7.4.

Because the tags are 64-bit aligned in the memory, another issue arises, if partial stores are performed. The problem is, that if the memory section was valid before, and a partial store with invalid data is performed, the whole data field is tagged invalid. Therefore, originally valid data bytes or words are suddenly invalid, even if the valid data itself was not modified. As a result, more invalid data is propagated throughout the memory on further operations and inaccurate tag informations are introduced. However, this feature is necessary, since partial stores on function pointers, shall also introduce a invalid tag within the memory, if the data is invalid. Otherwise, the protection does not apply in most of the situations. However, this will not invalidate function pointers within the application by accident. Function pointers are always 64-bit aligned and therefore partial stores can not cause and invalidation of otherwise valid function pointers. They are never aligned in a way, that parts of function pointers can be overwritten by byte stores on other data bytes or words. Therefore, no incorrect traps are triggered.

5.4.1 Propagation Rules

In the following section, the propagation rules for the function pointer protection policy are introduced. The tag ALU applies them to all operation types and the memory management provides the additional rules for partial stores and writes. This rules only apply to the INV tag bit and are able to co-exist with the return address protection rules. INV(rd) denotes the INV tag bit of the destination register, while INV(mem) denotes this tag bit within a given memory location. Table 5 lists all propagation rules for the function pointer policy.

Arithmetical and logical instructions always need to propagate the tag bit to the destination register, if it is set within at least one operand register. Therefore, the tag ALU performs an or operation on both given tags. It is impossible to clear the INV tag bit with any operation from the destination register. The move instruction copies the INV tag of the source register to the destination register without modification. As mentioned, jump instructions need to reset the INV tag in the return address register ra, since corruptions are already handled with the return address policy. Full store and load instructions also copy the tag without modification from registers to the memory, or from the memory to registers, respectively. Also, partial load instructions do not alter the tag, in contrast to the return address tag propagation rules, where the tag is reset in this case. Partial stores need to perform a combination of the tag, which is already stored for the double word and the tag of the source register. This way, the INV tag is not cleared, if the memory location is already marked as invalid and the source register has a value of "0" for this tag.

In addition to this rules, one additional rule is introduced. The INV tag is always set, if the tag generation is switched on and a load instruction was performed on any address within the IO space. The processor therefore needs to know which addresses are within the IO space. Using this technique, malicious data from devices is marked.

Instruction	Example	Propagation
Arithmetic	add rd, rs1, rs2	INV(rd) = INV(rs1) | INV(rs2)
Logical	or rd, rs1, rs2,	INV(rd) = INV(rs1) | INV(rs2)
Move	add rd, rs, 0(imm)	INV(rd) = INV(rs)
Jump [dircet/indirect]	jalr rd, rs, imm	INV(rd) = 0
Store [full]	sd [rs1], rs2	INV(mem[rs1]) = INV(rs2)
Load [full]	ld rd, [rs]	INV(rd) = INV(mem[rs])
Store [partial]	sb [rs1], rs2	INV(mem[rs1]) = INV(rs) | INV(mem[rs1])
Load [partial]	lb rd, [rs]	INV(rd) = INV(mem[rs])
Load [IO]	ld rd, [rs]	INV(rd) = 1

Table 5: Invalid tag propagation rules.

5.4.2 Example Attacks

In the following section, we are providing and example attack, which is detected and not possible with the provided policy.

```
int valid_function(int value)
{
  printf("Normal function! %d\n", value);
  return 0;
}

struct TestStruct {
  char buffer[4];
  int (*function_pointer)(int);
};

int main(int argc, char** argv, char** envp)
{
  int ret = 0;
  struct TestStruct *test = malloc(sizeof(*TestStruct));
  char string[256];

  //Initialize function pointer
  f->function_pointer = &valid_function;

  printf("Please type some string:\n");
  scanf( "%s" , &string[0]);

  //Copy string to buffer in struct
  strcpy(f->buffer, string);

  //Execute the function, given in the struct
  ret = f->function_pointer(0xdeadbeef);

  return 0;
}
```

Listing 6: Overwrite of a function pointer within the heap.

In the example, shown in Listing 6, a structure, that includes a buffer with the size of 4 bytes and a function pointer, is allocated in the heap. It is initialized within the main function. The function pointer is assigned with the address of the function *valid_function*. This code snippet demonstrates the use of a function pointer, which is initialized on the heap. User input is read to a temporary array during the attack, using the *scanf* function. Afterwards, a string copy from the temporary buffer to the buffer within the struct is performed. This function does not check, if the size of the target buffer is exceeded during the copy process. This leads to an overwrite of all memory contents after the buffer if the string, read from IO, exceeds the size of the target buffer. In this case, the function pointer is overwritten with the input string. This infers a invalid tag bit on the function pointer, since the input data was also tagged with the invalid tag before. As soon the function pointer is used for a function call, the trap triggers, since this operation is not allowed by the policy.

5.5 User Taint Tracking

As already mentioned, dynamic taint tracking is used to follow the spreading of malicious data within the system. The principle is the same as with the previously described function pointer protection policy. Our proposed tagged memory architecture assigns 4 bit tags to every double word. Two of this bits are necessary for the proposed countermeasures. However, we decided to allow the user, to implement her own dynamic taint tracking with the remaining two bits. This can be used for analytic processes, but also for debugging purposes. Therefore, we also introduce a debug tag trap. This trap is also handled within the tag check unit. Unlike the other introduced policies, the debug tag trap is triggered instruction independent. As a result, it is always triggered, once the processor attempts to executes an instruction and one of the operand registers includes a set user tag bit. More precisely, only one of the two user tag bits per register has to be set, in order to trigger the trap. This trigger can be deactivated and the system can be used to analyse the flow of data.

Another possible usage of the user tags, is disallowing the usage of tagged data. The user could, for example, tag key bytes, which are used within the application. By the time the key is used for computation, the trap trigger is turned off. However, if it shall not be used or copied somewhere else, the trigger can be activated. A malicious use of the key can then be detected using this technique.

The supporting instructions storetag and loadtag allow the user to modify those tag bits, in order to implement additional features. As with the other policies, the tag ALU is performing the propagation of the user tag bits within the processor.

Note, that these instructions always affect all four tag bits in our prototype. Therefore, modifications to tags have to be done carefully, to not influence other policies.

5.5.1 Propagation Rules

This section introduces the propagation rules for the user tags. Table 6 lists this rules. Note, that the rules apply for each of the two user tag bits separately. Therefore, there is no connection between one user tag bit to the other. The rules are identical to the rules of the function pointer protection policy. However, loads on addresses within the IO space always introduce the value “0” for the user tag bits.

Instruction	Example	Propagation
Arithmetic	add rd, rs1, rs2	USR(rd) = USR(rs1) | USR(rs2)
Logical	or rd, rs1, rs2,	USR(rd) = USR(rs1) | USR(rs2)
Move	add rd, rs, 0(imm)	USR(rd) = USR(rs)
Jump [dircet/indirect]	jalr rd, rs, imm	USR(rd) = 0
Store [full]	sd [rs1], rs2	USR(mem[rs1]) = USR(rs2)
Load [full]	ld rd, [rs]	USR(rd) = USR(mem[rs])
Store [partial]	sb [rs1], rs2	USR(mem[rs1]) = USR(rs) | USR(mem[rs1])
Load [partial]	lb rd, [rs]	USR(rd) = USR(mem[rs])
Load [IO]	ld rd, [rs]	USR(rd) = 0

Table 6: User tag propagation rules.

5.5.2 Example Usage

The debug taint tracker feature can be, as previously described, used to debug tagged memory and also to implement own security policies. In Listing 7, the variable *key_byte* represents data, the user wants to monitor. Therefore, the user tag 0x08 is set for this data, using the function *stag*. Afterwards a x-or operation of the result variable and *key_byte* is performed. Since the tag is set on this variable, the processor traps, as soon this logical instruction is performed. The user therefore gains information, when the monitored data is used throughout the program. Furthermore, the user could switch off the trap activation, which would lead to a dynamic tracking of all data, which was combined with the monitored data in any way.

```
void main(void)
{
  uint8_t result = 0x00;
  uitn8_t key_byte = 0xE2;

  //Mark key byte with user tag bit 0x08
  asm volatile ("stag %0, 0(%1)" ::"r"(0x08),
               "r"(&(key_byte)));

  result = 0x03;
  result = result ^ key_byte; //Trap triggers as soon key_byte
      is used for logical operation
}
```

Listing 7: Example of user taint tracking.

Chapter 6

Related Work

After the introduction of the principles of tagged memory and our proposed countermeasures against code-reuse attacks, we now elaborate related work on these topics. As already mentioned, code-reuse attacks and their countermeasures are very important and popular research topics at the moment. For example, the book of Lucas Davi and Ahmad-Reza Sadeghi [24], covers different types of software-based countermeasures. Also, since tagged memory is no new idea and tagged architectures where already used for similar implementations as in this thesis, different mechanisms are outlined within this chapter. Since DEP is not preventing code-reuse attacks, only the new effective mechanisms are introduced.

The most commonly implemented countermeasure against these kind of attacks, is the so called Control Flow Integrity (CFI). Every program has a predictable control flow structure. This structure can be represented with a Control Flow Graph (CFG) for an arbitrary program. Using this graph, the allowed transitions of a program can be identified, if the graph was generated beforehand. Since code-reuse attacks like jump-oriented programming, alter the original control flow of the program, the inconsistency between the CFG and the actual execution can be detected. The team of M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, from the Microsoft research labs, developed a mechanism for CFI, embedded within a Windows operating system [30]. They first analyse the static program binary and add labels into the machine code. This labels represent the possible targets of call instructions and are calculated using the information from the previously generated CFG of the program. The downside of this approach is, that the insertion of labels effectively change instruction and jump target locations within the binary. This means, that the algorithm also needs to adjust this addresses. During the program is executed, the operating system checks, if a jump is directed to the correct possible label. If this is not the case, because a jump address was modified, the violation is detected, since the label does not match the actual jump target. However, this approach only applies to jump oriented programming attacks, since the label generation and checks are only performed for call instructions. Return instructions are not checked, since the CFG does not provide useful information for return control flow changes. This results from the fact, that return instructions can have multiple possible targets. The same function could be called from different functions throughout the program and the return target is therefore not predictable. However, a solution for this issue is given by applying shadow stack implementations. Shadow stacks can dynamically track the original return address, before the return address is overwritten. Then this values are compared. Both variants of CFI have a couple of drawbacks, when they are used in praxis. Although no modifications of existing programs are required to apply the countermeasure, the performance overhead while execution is significant. The operating system has to check the correctness for every control flow transition, which introduces additional instructions, that need to be executed. The performance penalty is even bigger with the shadow stack approach, since data needs to be stored and read on the shadow stack, as well as comparisons needs to be performed. Also, this policy requires modifications of the operating system. Our solution, targets this drawback, by providing a hardware solution with minor performance and memory overhead, while only requiring minor changes to operating systems in order to prevent false traps.

A further common strategy to prevent code-reuse attacks is Address Layout Randomisation (ASLR). This method was first introduced by S. Forrest [5]. Today ASLR is applied in all modern operating systems like Linux and Windows, since the countermeasure is promising and well accepted by researches [28]. This technique allows, to dynamically randomize the location of the heap, stack, libraries and the executable in the virtual address space. This is archived, by randomizing the start address of each of the mentioned sections and therefore, placing them on a different location in memory, whenever the program is started. As a result, the attacker is not able to calculate gadget addresses beforehand, since all entry points will virtually reside within a different address space, as during the static gadget discovery analysis. The countermeasure prevents attackers from mounting the attacks in the first place. While this strategy is promising, attackers are still able to mount code-reuse attacks, if they can guess the location of a single library function. If this address is known, all other library function addresses can be calculated, since the offset within the block is constant. This can be done with brute force approaches [6]. Furthermore, if an attack can be launched, no further checks are performed during runtime and no violations of the control flow are detected. Again our strategy differs from this approach, by not requiring software and the security policies are enforced by the hardware. Dynamic violations during executions can be detected.

Nowadays, tagged memory becomes popular again for security purposes. As a result, there are similar tagged architectures existing, which all focus on countermeasures against code-reuse attacks. Each of these architectures have different approaches, of how tags are used to identify control flow violations. The first architecture to be introduced is published by G. Edward Suh, Jaewook Lee and Srinivas Devadas [7] and targets on secure program execution by dynamic flow tracking. This architecture was implemented within an ARM processor, and uses a single tag bit for operation. The tag bit identifies, similar to our proposed solution, authentic and malicious data. It is assumed, that all data, read from the IO, is potentially malicious and addresses, generated from this data, shall not be used to perform control flow transitions. They succeeded in effectively detecting the malicious control transfers, by enforcing the tagged security policy directly in the core. All data tagged as malicious, can not be used as address for jump or return instructions. They manage tag partitions within the software. Their architecture is capable of tagging byte wise, instead of only incorporating tags per 64-bit data. In order to prevent tagging of each data byte when it is not necessary, this feature can be enabled page-wise. A page can hold tags per 64-bit, while the other is tagged byte wise. This allows to reduce memory overhead. The decision, which type is used for the current page, is dynamically performed, when a fine granular write was applied on the page. This is not the only difference to our proposed architecture. The IO invalid tagging is not performed by the processor, but by the operating system. Therefore, the operating system needs to coordinate the activation of the tagging process. Both the solution for different tag modes per page and the enabling of the IO invalid tagging in software, differ from our architecture. We perform IO tagging within hardware, while the operating system only needs to disable the feature when binaries are loaded. Furthermore, the operating system does not need to be aware of the memory management of tags at all.

Another architecture, which was recently published, also uses a single tag bit to build security policies within the core [22]. The design is, as in our thesis, integrated within the Rocket chip. The idea behind this architecture, is dynamic flow tracking. The tag bit is set by software for security critical data and jump addresses, with a tag set instruction. The tag bit is then propagated throughout the core registers and memory. However, if data, which holds a set tag bit, is modified, the tag is automatically reset by the core. Therefore it is possible, to check the modification of this tagged data. The check, if the tag bit is still set at any given time, is performed within software, by executing a special tag check instruction. This means, that the security policy is entirely enforced within software. This allows a big variety of use cases, the tagged architecture can be used for. But it also introduces the problem, that the software has to be aware of the tags and the programmer needs to implement tag set and check instructions within the critical code or data of the program. With our proposed architecture, the security policies are enforced within hardware and therefore, no potential erroneous introduction of these tag check instructions can occur. Also, this additional checks introduce a performance overhead, which was avoided with our architecture.

The next tagged architecture is very similar to our proposed design. The two master students S. Fingeret and J. A. González designed a tagged architecture called Taxi [10][25]. One part of their security concept matches our return address policy. A return address tag is set after a jump instruction. If the tag is reset on the return instruction, a trap is performed. They also propose a function pointer protection policy, which detects modification of the addresses and traps, if a modified function pointer is used as a jump target. However, this policy requires compiler support in contrast to our proposed function pointer policy, where all IO inputs are tagged as malicious automatically. Furthermore, the architecture was only implemented within the RISC-V instruction set simulator called Spike. This means, that no real hardware design is existing which is supporting this policies.

Chapter 7

Implementation

The previous chapters included the description of the proposed countermeasures for code -reuse attacks in theory. Also the background of the RISC-V ISA and tagged architectures is given. This chapter introduces the implementation details of our tagged architecture. We implemented the system on real hardware, which was tested on a Kintex-7 FPGA [12]. The tagged architecture is developed on top of the lowRISC SoC, which is based on the Rocket chip. In this context, the term chip is used to identify the whole chip architecture, except peripherals and memory, while the term core is identifying the main processing unit, including the the L1 caches. This Rocket chip implements the RISC-V ISA, is open source and developed by the University of Berkley. The Rocket chip is the desired choice for this thesis, since it gains more and more attention throughout researchers. The core can be vastly modified and parametrized, since a new hardware design language is introduced. This language is called Chisel, and is also developed by the University of Berkley. It eases development and still provides full control over the outcome of the chip, which is already proved by multiple tape outs. Researchers also profit from the extensible RISC-V ISA, which is of course fully supported by the rocket chip. They provide the GNU toolchain, needed for compiling test programs and the Linux kernel, as well as the developer tools for the chip itself. Also, the Linux kernel, which is adapted for the processor, is provided. First, the processor structure and the implementation of tagged memory is described. Afterwards, the policies are implemented within the core pipeline and the L1 data cache. In the end, the modifications to existing software and the Linux kernel is described.

7.1 lowRISC

The lowRISC project [14], is a non-profit organization at the University of Cambridge. They extend the 64-bit Rocket chip with features necessary, in order to build a fully open SoC. The goal is to produce a “Raspberry Pi for grown-ups”. The focus is on flexibility of the system and also security related topics.

Since lowRISC has a focus on security, it includes a tagged memory architecture [17]. The basic framework for building security or debugging features are given. The key architecture feature is a combination of a tag cache, which manages the tag and data combination and access of the tag region within the memor, and data cache, which is aware of the tags, stored per every 64-bit data. The core is not aware of tags and only provides the previously mentioned store- and loadtag instructions. This instructions directly modify tags within the data cache and does not require support by the core itself. This basis is suitable for the implementation of our proposed counter measures and so we build the extensions on top of this lowRISC processor. However, tagged memory was removed with the newest release of the chip, which is why we re-implemented this feature.

7.1.1 Rocket Chip

The rocket chip is a 64-bit architecture and implements the RISC-V ISA. The chip is both realizable with an ASIC and a FPGA design flow. As ASIC, the chip can be operated with a clock rate up to 1 GHz and reaches 1.72 DMIPS/MHz. Within a FPGA design, the maximum clock rate is limited to 50 MHz. The power efficiency is significantly improved when compared to a 32-bit ARM Cortex A5.

Figure 8 gives an overview of the Rocket chip's architecture. The Rocket chip can consist of multiple tiles, to enable multicore architectures. The tile is the processing element called Rocket core. Each tile consists of a core and L1 data and instruction caches. The core incorporates a single-issue in-order 5-stage pipeline. The core is able to support a floating point unit, based on the chosen configuration. The data and instruction caches are separated and provide their own interface to the core. These caches are connected to the outer memory system, using the so called L1-Network. This network is necessary to coordinate L2 cache accesses. Therefore, L2 cache requests are arbitrated from the multiple cores. The bus system, used for the communication with the outer memory system, is called TileLink. TileLink includes multiple channels, such as Acquire, Release, Grant, Probe and Finish, to allow coherency managers to coordinate data flow. The L2 cache banks reside in the outer memory system and are interconnected by the L2-Network. If a L2 cache requires data from the memory, or performs a write-back, the outputs are arbitrated. The output of the L2 caches is transferred using a TileLink bus. As a consequence, the bus needs to be converted to the memIO bus format, in order to communicate with the memory interface. The host communication is handled within the processor, using a host interface unit. It is used to perform communication with the host processor, which includes reset commands and Universal Asynchronous Receiver Transmitter (UART) input and output. It reads and stores values, that shall be handled as IO in registers and transfers them over the hostIO.

The Rocket chip is designed with a new programming language called Chisel. It is based on Scala, which eases the design flow in different aspects. The reason is, that object oriented programming is added and other programming concepts of Scala, like maps are available. Also, there is no strict separation of data path and control path necessary, since the code can be structured differently compared to existing Hardware Description Languages. The Chisel design code is then translated to a Verilog RTL description, which can then be synthesized by further design tools. Therefore, both ASIC and FPGA design flows are possible. Chisel also provides a cycle accurate C++ simulation of the hardware design. The time needed for development and tests is significantly reduced by this feature.

TileLink. The TileLink interface bus is a central part in the architecture of the Rocket chip, since most components are interconnected using this interface. TileLink is currently in the beginning phase of being standardized as on-chip communication bus [18][19]. As already mentioned, TileLink is used to coordinate hierarchical cache coherency within the rocket chip. Since it is an hierarchical system, there exist two different types, namely a manager TileLink port, as well as a client TileLink port. Clients can be caches, DMA engines or any other component, which does not necessarily keep local copies of data. The clients initiate the transaction, by sending requests to the master. Clients either request cache data blocks from the outer caches or write back their cached data. The manager, on the other hand, controls the access permissions and respond data to the client, if requested. Managers are located in the outer-level of the hierarchy. Note, that components within the hierarchy can incorporate manager and client ports at the same time. The client port communicates with components in the outer-level, while the manager port communicates with components in the inner-level. Taking the Rocket chip as example, the L1 cache's client ports are communicating with the manager port of the L1 network. The L1 network also includes client ports, which are communicating with the manager ports of the L2 cache bank.

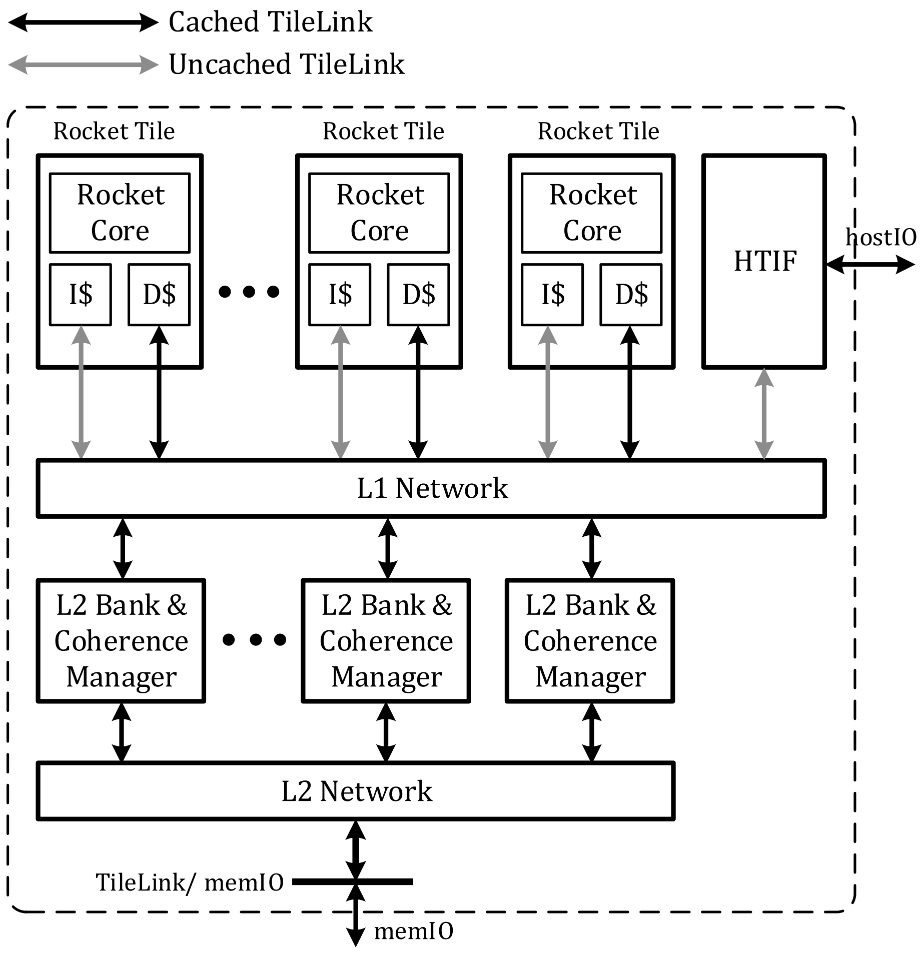

Figure 8: The overview of the Rocket chip, designed at the University of Berkley. The different types of the connections of the modules on the chip are defined in the legend in the top left corner. All components, ranging from the multiple cores to the memory connection of the L2 cache banks, are presented.

TileLink consists of five separate transaction channels: Acquire, Probe, Release, Grant and Finish. Each channel is designed as DecoupledIO, available within Chisel. This means, that every channel has ready and valid strobes. The sender signals valid data on the channel by setting valid to high. The recipient signals, that it is ready to receive information, by setting the ready strobe to high. A data transaction is performed, if both strobes are high at the same time. This allows a synchronization of data transfers between different components on the chip.

– The Acquire channel is used by the client, to initiate a read or write request. In a coherency system, this channel acquires access with proper permissions to a cache block. Data write requests are performed using an acquire, if the sent data is not cached by any component. The most important information, sent with this channel is the address, the client id of the component, the data and the beat number of the transaction. Also, the operation type is given, in order to handle atomic memory operations.

– Probe. This channel is used to probe a client i.e., to determine, if it has a specific cache block. It is also used to revoke permissions on the selected cache block. The probe channel's strobes, are the address of the cache block and the transaction type.

– Release is both used as response to a reception of a probe or to transfer voluntary write back data. If it is the response to a probe, the client signals the release of permissions for the cache block, while sending the dirty data. If it is used to voluntary write back the data, also data is sent using the release channel. The channel includes the address, the client id, as well as the data and the beat number. It also includes a signal to identify a voluntary release transaction.

– The grant channel is used to respond data to the original requesting communication partner in case of a acquire operation. If data was written, it serves as acknowledge that the write was performed. This is true for both uncached write operations and for voluntary releases in cached TileLink connections. It includes the type of the transaction, the client and manager id, as well as the data and the beat number.

– Finish. This channel is used to signal a completion of a TileLink transaction sequence by the master. This is used, to order synchronize transactions within the chip components.

The TileLink bus exists in two variations. The cached variant handles coherency between the components, by transmitting Acquire, Probe, Release and Grant messages. The uncached variant, on the other hand, is used, if data is not cached by any of the components and therefore the Release and Probe channels are not included.

7.1.2 Tethered Structure

The reason, why tagged memory is not included in the newest version of the lowRISC SoC yet, is a major architecture change between the tagged memory (v0.1) and untethered (v0.2) releases. The original Rocket core is tethered. This means, that IO and memory access is performed with the help of another ARM processing system, as for example, located on the ZedBoard [11]. The ARM core handles all IO accesses and passes them to the Rocket core, using a host interface. Memory operations are preformed by this coprocessor too. This implies, that the Rocket core is not able to perform bootstrapping on its own. Whenever a application shall be executed on the Rocket core, the ARM processing system prepares the memory, by moving the application code to the correct location in memory and resets the Rocket chip. This is archived with a frontend server application. The frontend server application runs within Linux on the ARM processor. When any application should be executed on the rocket core, it prepares the memory and re-routes the terminal IO to the rocket core. For example the Linux kernel can be booted on the Rocket core, using this strategy. The structure of this tethered architecture is shown in Figure 9. The Rocket core is connected with two different interfaces to the ARM processor. The hostIO is used for terminal input and output. It is then converted to an AXI-4 [15] bus system. All memory accesses are performed via memIO. This bus is converted into a AXI-4 HP(high performance) bus. The ARM processing system offers a direct interconnection of this bus to the DDR3 memory on the FPGA evaluation board. The memory controller is included within the ARM processing system.

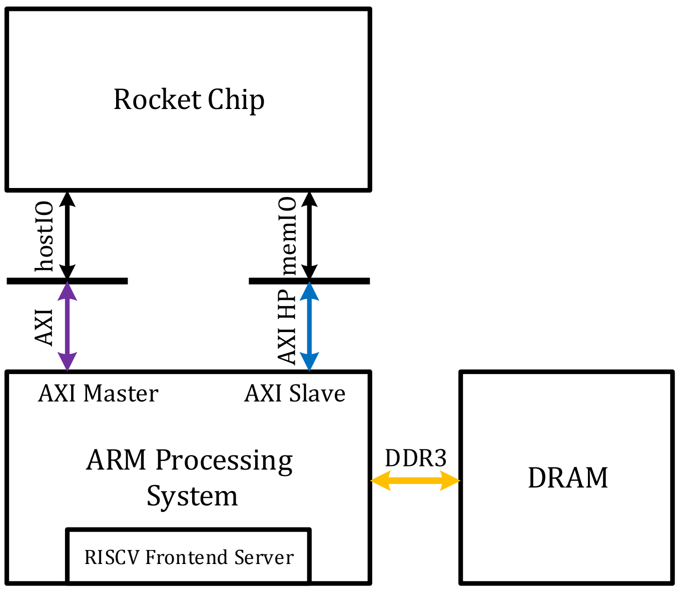

Figure 9: The tethered lowRISC system architecure. The Rocket chip is interfacing an ARM processing system in order to perform the communication with the host and the DRAM. The RISC-V frontend server application prepares the memory for the Rocket chip, if this core shall execute a program. Once the program is executed, the frontend server continues to run on the ARM processor in order to handle user interactions.

7.1.3 Untethered Structure

However, the tethered structure is not optimal, since it requires another co-processor within the SoC design. This makes it impossible, to design a standalone SoC. Therefore, the Rocket core was re-modelled to an untethered structure within the v0.2 lowRISC release. Unethering required a major change of the processor structure, since the concepts of memory mapped IO and direct memory access were not implemented before. The Rocket chip is now able to access the IO devices and the memory without the help of a co-processor.

As it can be seen in Figure 10, this version of the Rocket chip, does not include the hostIO and memIO, but instead the NASTI-Lite and NASTI bus. The NASTI bus handles the communication with the memory, while NASTI-Lite interfaces the peripheral devices directly. Note, that NASTI is the implementation of AXI-4 by the University of Berkley. Furthermore, a Block RAM (BRAM) is introduced alongside with the DRAM. This BRAM is used for the boot procedure. In order to select the appropriate memory, the Rocket chip is connected to a NASTI-Crossbar on the top level architecture of the chip design. Currently, this architecture is implemented for the Kintex-7 FPGA board from Xilinx and Xilinx Vivado is used for development. In order to interface the DRAM, the DDR3 memory interface is connected to the NASTI-Crossbar. It translates AXI-4 bus requests into memory operations within the DRAM. The available peripheral devices, at the current point in time, are a UART interface for terminal input and output and a Serial Peripheral Interface (SPI), in order to read and write data to an external storage (Flash), e.g., SD card.

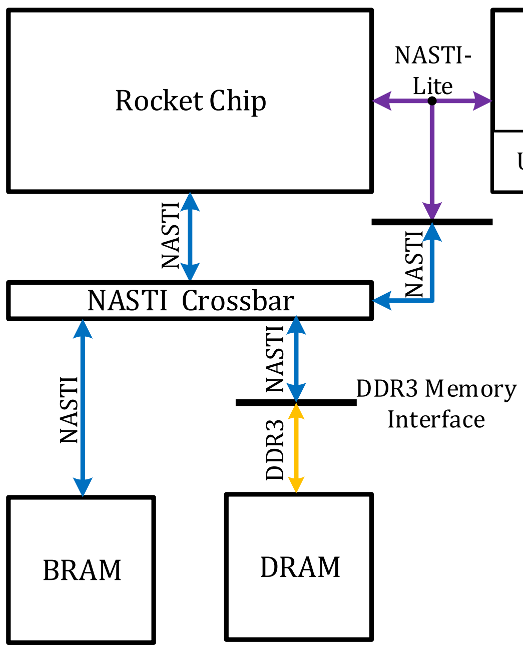

Figure 10: The untethered lowRISC system architecure. The Rocket chip is now directly connected to the memory and periheral devices. No seperate ARM processing system is neccessary, which allows the chip to operate stand-alone. The BRAM is introduced, to make the bootstrapping process possible, by including the start-up program.

To support all this changes, a couple of features had to be added to the processor. First of all, a memory map is introduced. It is possible to map addresses to the two different memory or to the IO space. Up to four memory and four IO mapped areas can be defined during runtime within the processor. Therefore, the system is flexible enough to support future modifications. To support memory maps, an additional Processor Control Register (PCR) controller and its registers are added. This is needed to regulate shared resources between multiple cores. In contrast to the already existing Control Status Registers (CSR), settings in the PCRs apply to all cores within the Rocket chip. The PCRs are realized as a subset of the CSR address space and includes global timers, the memory and IO maps, as well as interrupts. It is therefore possible, to add global control values, which do not only apply to the core. The recipients of PCR updates can be added within the processor, by implementing the PCR ports in the module. This interface notifies all recipients, when a register value is modified. Also the soft reset can be triggered using PCRs. Interrupts are handled using the PCR control as well. As mentioned previously, all peripheral devices are now directly connected to the Rocket chip and requests are performed in an uncached manner. Therefore the IO requests are handled within the L1 data cache. This is achieved by memory mapped IO. The L1 cache checks, if the address is within the IO space and if this is the case, an uncached data transfer is performed. IO accesses always trigger a miss within the cache, and are handled by a new IO miss handler. As a result, IO data is never stored within the L1 and L2 caches. For the core, there is no difference in addressing memory or IO, so the requests are handled in the same manner.

The problem with the tethered structure is the lack of the bootstrapping capability on the Rocket chip. With the newly introduced BRAM and PCR this problem is targeted. The content of the BRAM is stored within the bit stream for the FPGA configuration. It is also possible to exchange the content of the BRAM, using the *data2mem* function of Vivado. The chip does not have to be synthesized whenever this content needs to be changed. The BRAM has a size of 64 kB. Within the FPGA design, the BRAM is both used as boot ROM and Flash, which means that in an ASIC design flow, this memory would need to be realized by using two different memory types. The boot loader is stored within this memory, in order to control the boot strapping procedure. This program represents the first-stage boot loader. If Linux shall be booted, the Berkley Bootloader (BBL) needs to be loaded in the DRAM first. The BBL is the second-stage boot loader. In order to prevent parts of the BBL getting lost in the caches, the DRAM is first mapped into the IO space. This is archived by the memory and IO maps. This bypasses the L1 and L2 cache, when data is transferred to the DRAM. Afterwards, the BBL is loaded from the SD card and stored in the DRAM. Then, the DRAM is mapped to the boot memory location, which is on address 0x00000000. Since the BRAM is also located in the same area, it gets hidden. As last step of the first-stage boot loader, a soft reset is triggered and the processor starts execution of the BBL, which has its entry routine at address 0x00000200. The BBL is designed to initialize peripherals, set up the virtual memory and load the Linux kernel in the DRAM. When Linux is booted, BBL continues running underneath, in order to handle interrupts and IO requests form the kernel. The privilege level of BBL is therefore the machine mode, while Linux runs in supervisor mode. Table 7 shows the physical address mapping of the currently existing memory and peripheral devices.

Name	Physical address space	Size	Type
BRAM	0x00000000 - 0x0000FFFF	64 KB	Memory
DRAM	0x40000000 - 0x7FFFFFFF	1 GB	Memory
UART	0x80001000 - 0x8000FFFF	64 KB	IO
SPI	0x80010000 - 0x8001FFFF	64 KB	IO

Table 7: The address layout of memory and peripheral devices, currently available in the untethered lowRISC architecture. Note, that the size of the UART and SPI section is only resulting from the mapping of the base address. The addresses valid for each device is only as big, as the control and data registers, available for the device.

7.2 Untethered lowRISC with Tags

In this thesis, we re-implemented a tagged memory architecture in the untethered Rocket core, designed by the lowRISC project. This is necessary as a base for enforcing the previously proposed tagged security policies. The modifications, to enable tagged memory within this system range from the addition of the support within the L1 data cache, as well as all TileLink data widths, to the tag cache, which coordinates tag load and store requests to the memory. Furthermore, two instructions are added, to modify and read tag values at given memory locations. This is necessary to perform tests, give the user the ability to implement own tagged memory applications, as well as to adapt the Linux kernel to support the tagged memory policies. The tag management within the memory is done using the tag cache. In contrast to the tagged memory architecture within the tethered Rocket chip, the tag cache also needs to support the two different memory. Tags can be stored in each memory separately. With the tag cache, memory access overhead is reduced significantly.

As already mentioned, the implementation of the tagged architecture, used in this thesis is based on the tagged memory release of the lowRISC project. However, the tag cache had to be re-written in order to match the untethered structure. This section describes the mentioned changes of the Rocket chip in general. First the overview of the new architecture is given, followed by general changes and finally the tag cache implementation. The handling of tags and the corresponding security policies within the core pipeline is described in detail in Section 7.3.1.

7.2.1 General Modifications

Figure 11 shows the simplified structure of the untethered Rocket chip with support for tagged memory. The L1 and L2 network is not shown, since the system was developed using only one core and L2 cache bank. We did not perform tests using multiple cores within this thesis. The networks still exist in our final implementation, but can be imagined as a direct connection for explanation purposes.

The architecture is designed in a way, that the number of tag bits can be changed within a configuration file, written in Chisel. This parameters are read at the time, the Verilog code is generated and are constant. They can be also used to set the number of cores, bus-width and so forth. Currently, the number of tag bits is set to 4 per 64-bit data. Every number of tag bits to the power of two is possible. However, the minimum number of tags is limited to 4. This results from the implementation of the L2 cache, which will be explained later in this section.

The Rocket core itself is not aware of tags in the general tagged memory implementation within the lowRISC chip. However, in this thesis, we go a step further and also transmit the tags to the core on memory store and load requests. The tags are transmitted via the HellaCacheIO interface. Tags are transmitted together with the data, by increasing the interface's data channel. The core can then processes tags, if the corresponding policies are implemented. Additionally, two new instructions are added to the instruction set to manually modify tag value. These instructions perform a modification of tag values directly within the L1 data cache/memory. Therefore, arbitrary tagged applications can be developed by the user. These instructions are denoted *stag* and *ltag*. As every other memory instruction within the RISC-V ISA, they directly access the L1 data cache without modification of core registers. Therefore, the same instruction type can be used and involves less modification to the core. The load and store instructions for tags, therefore only change the tag portion of data within the L1 data cache. Especially the store tag instruction is necessary to allow full compatibility of operating systems with the tagged security policies (see 7.4.2), or to perform custom tag operations. Within the core, multiple memory request types are defined, as shown in Table 8. Tag loads and stores require a new message type M_T within this set of message types. This type indicates the L1 data cache, that the data shall be loaded and stored from the tag portion, instead of the data portion within the corresponding cache line. In addition, the new instruction identifiers are added to the RISC-V GCC, so that the assembly can be created properly. However, this modification was already included in the original tagged lowRISC and is integrated again, without modification.

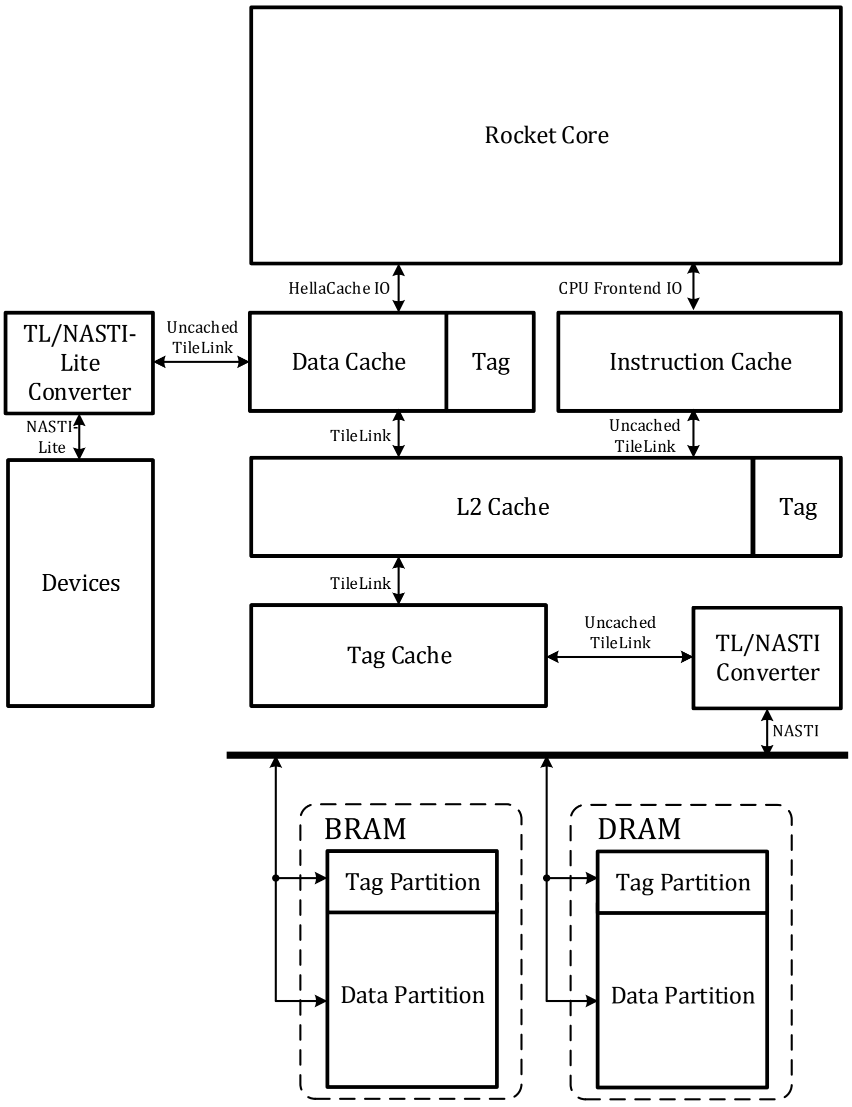

Figure 11: Overview of the untethered lowRISC system in combination with tagged memory support. In order to handle tagged memory operations, the tag cache is introduced. Therefore, the L2 cache is not anymore the last component before the memory requests are performed. Furthermore, the memory themselfes are splitted in seperate data and tag address ranges.

L1. The L1 data cache handles tag loads and stores and therefore has to be aware of tags. This is realized by increasing the cache line size by the number of tag bits per 64-bit data, rather than adding another memory, dedicated for tags. One cache line within the L1 holds 8 * 64-bit of data. This results in 512 bits. By adding 4 bit of tags, the total size is increased to 544 bits. As a consequence, the data cache needs to receive this exact amount of bits, when requesting data from the L2 cache at a cache miss. This is archived, by modifying the width of TileLink's data strobes. The TileLink data transfer is performed in a burst with four beats, each carrying 128 bits of data. Within four clock cycles, 512

Abbreviation	Encoding [3:0]	Description
MT_B	0000	Byte (8-bit) memory operation
MT_H	0001	Half-Word (16-bit) memory operation
MT_W	0010	Word (32-bit) memory operation
MT_D	0011	Double-Word (64-bit) memory operation
MT_BU	0100	Upper Byte (8-bit) memory operation
MT_HU	0101	Upper Half-Word (16-bit) memory operation
MT_WU	0110	Upper Word (32-bit) memory operation
MT_T	1111	Tag memory operation

Table 8: The message types for memory transfers. The message type MT_T was added to allow tag-only modification of the given address. The message types are interpreted by the L1 caches of the Rocket chip.

bits of data are transferred. By increasing the width of one burst to 136 bits, the same amount of data, including the tags is transmitted, without adding another bus cycle. If the data transfer size would be still equal to 128 bits, a fifth burst with incomplete data needs to be sent. Therefore we decided, to increase the transfer size of each beat. The assembly of data and tags within this TileLink bursts is shown in Figure 12.

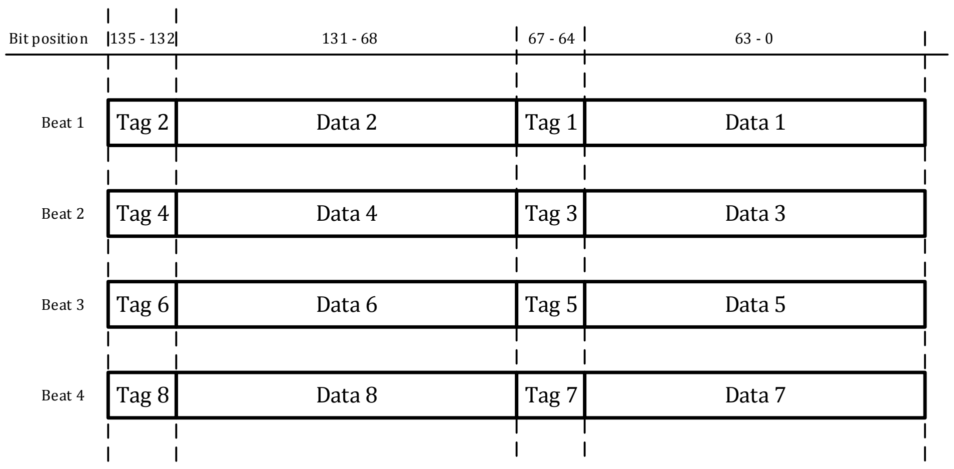

Figure 12: The TileLink data and tag transfer. The transfer consists of 4 beats, with the size of 136 bits. In each beat, two 64-bit data fields with the corresponding 4-bit tag fields are transmitted. Processor modules, which need to operate on tags or data separately, shall strip out or add the chosen component out of each burst.

Further modifications are necessary within the L1 data cache in order to handle tags. A detailed description is given Section 7.3.2. The main modifications include the extension of data wires within the L1 cache stages. For the implementation of the untethered lowRISC with tagged memory, the tags do not need to be transferred to and from the core with every load or store. The tagged system operates using the two new instructions load and store tag. Therefore, tags are per default stripped from the data, loaded from the cache memory, before the response is sent to the core. Also, when the core stores data, the old tag is used without modification. In order to change the values of tags, the new instructions store- and load tag can be used.

The L1 instruction cache does not hold any tags. Instructions, fetched from code pages are assumed to be trustworthy. However, the addition of tags to code words would be also possible with this architecture. As a consequence, no changes are performed within the instruction cache. The only modification needed for the instruction cache, is to remove the tags transferred by TileLink before storing the data into the cache line. This is necessary, since the TileLink burst size is increased within the whole chip and the L2 cache always transfers data with the data + tag scheme.

L2. The L2 cache itself is ignorant to the tags in its operation, as it only appears as a bigger cache line size. It contains both data and tags within the cache lines. The L2 cache performs its communication with the tag cache and the L1 cache using the the TileLink interface. To the L2 cache, data received and sent through the TileLink interface, appear as normal untagged data. This increases the transparency of the tagged architecture and also eases the implementation. Therefore, all communications include data and tags in order to be big enough to fit into a L2 cache line. The cache lines are also increased to 544 bits in total. No further modifications are needed for the tagged operation within the L2 cache.

As mentioned before, the number of tag bits per 64-bit data can only be defined as a power of two and a minimum of 4 bits. The reason is the implementation of the L2 data array and the TileLink transfer. The data array can be addressed byte wise. Both the total count of bits in a L2 cache line and the TileLink transfer size, has to be a multiple of 8 bits. This is satisfied, even if the number of tag bits would be equal to two. In this case the total cache line size would be 528 bits, which is a multiple of 8. However, the problem occurs from the TileLink data beats. Each burst contains 128 bits of data, which requires two additional tag fields. If the count of tags would be equal to two, this TileLink burst would consist of 132 bits, which is not a multiple of 8. Since the data is immediately stored when the burst arrives, this amount of bits causes the data to be not correctly aligned in the L2 data array. The beat size of the TileLink is not a multiple of 8 in this case. In order to fix this issue, sub-byte write masks for the L2 data array would be necessary and also sequential writes, would occur, since the write address is shifted. Out of performance and size considerations, this fixes would introduce to much overhead and complexity.

However, every other number of tags, which is a power of two and bigger than 4, is allowed within this system. For example, including 8 tag bits per 64-bit data, would lead to a TileLink burst size of 144 bits, which is again a multiple of 8 and can be transferred to the data array interface.

Tag cache. Compared to the untethered structure without tagged memory, the L2 cache is not directly connected to the TileLink/NASTI converter anymore. Instead, a tag cache is introduced and connected to the L2 cache. It acts as a manager, which coordinates the data and tag memory access. When the L2 cache requests data form the tag cache, the data and tag requests are sent to the memory in an sequential manner. Same applies also for the write back of data and tags. This additional memory accesses only need to be performed, if the tag cache array, does not include the necessary tag portion for the requested data or the tags have to be written back to the memory, before new tags can be fetched. Therefore, memory transactions are reduced and the overall performance is not decreased drastically. The memory system consists of two separate memory, as already described within the bootload procedure. As a result, all tag partitions are separated for each memory. The tag cache can manage tag partitions in each of this two separate memory. Table 9 shows the address mappings for the tag partitions in the DRAM and BRAM. More specific details of the tag cache are described in Section 7.2.2.

Memory Name	Type	Phys. Address Range	Size
BRAM	Data	0x00000000 - 0x0000EFFF	~60 KB
BRAM	Tag	0x0000F000 - 0x0000FFFF	~4 KB
DRAM	Data	0x40000000 - 0x7BFFFFFF	~980 MB
DRAM	Tag	0x7C000000 - 0x7FFFFFFF	~64 MB

Table 9: Physical addresses and sizes for data and tag partitions.

TileLink/NASTI Converter. The last element, before the memory accesses are performed, is the TileLink to NASTI converter. It receives either data write or read requests from the TileLink acquire channel and converts the requests to the NASTI bus format. NASTI consists of write address, read address, read, write, and write response channels. In this particular case, the data channels, namely read and write are 128 bits wide. This matches the size of TileLink data transfers in an untagged architecture. Also, the requests currently consist of 4 beats, which equals to a total data transfer size of 512 bits. Whenever the converter receives a write request, the address is set within the address write channel and transmitted to the memory controller, while simultaneously transmitting already the first data beat within the write channel. The next beats from TileLink are then forwarded, using the write channel. The last transaction of data is indicated by setting the "last" strobe of the write channel to high. The memory then responds a valid write via the write response channel, indicating that all data was received. The converter responds to the tag cache, that the transaction was competed. When a memory read operation is issued by the tag cache, the read address channel is filled with the appropriate address information and is sent to the memory controller. The memory controller then responds with beats of data within the read channel, indicating the last packet by setting the "last" strobe to high. The data bursts are replied to the tag cache, whenever valid data arrives from the memory.

7.2.2 Tag Cache

In the previous sections, the overall system architecture of tagged memory within the untethered Rocket chip and its general modifications were introduced. In this section, the tag cache is described in detail. A detailed illustration of the tag cache and its components is given in Figure 13. As already mentioned, the tag cache is responsible for managing tag and data accesses whenever a request is issued. Also, it is the last element within the chip, before the TileLink requests are converted to the NASTI interface and the memory access is performed. The tag cache is able to issue additional tag requests whenever necessary. The L2 cache banks are connected to the tag cache by the tag cache network. This is a network, which is used to arbitrate the multiple TileLink ports of each L2 bank. However, since we did not use multiple L2 cache banks, this network can be imagined as direct connection from the tag cache to the single L2 cache bank.

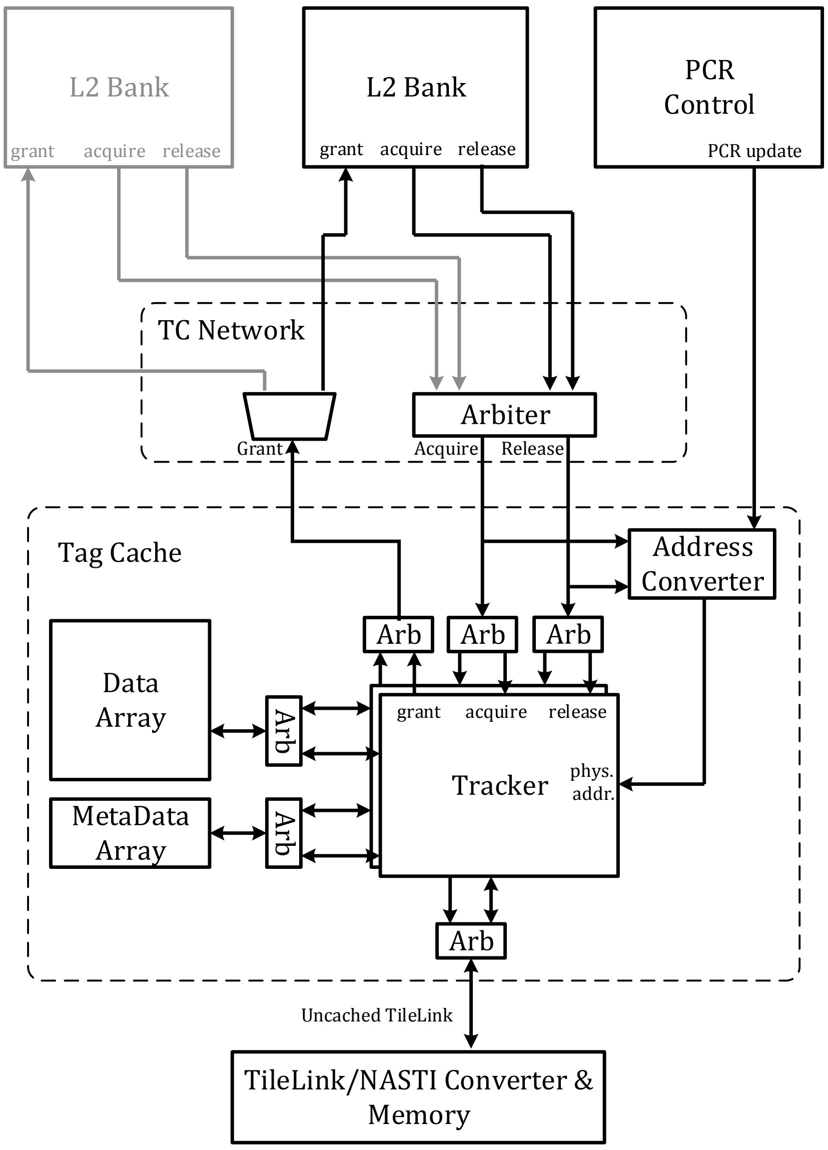

Figure 13: The tag cache structure in detail. The tag cache consists of a data array, where the tags are stored, a meta data array to keep track of the validity of the cached tags, as well as an address converer and multiple trackers. The address converter converts core addresses to physical addresses, while the tracker manages all memory requests and tag operations.

The tag cache itself is consisting of an address converter, a data array, a meta data array and one or more trackers. The address converter, converts the core address of a memory access into the real physical address of the memory. This is necessary, since the address space of the different memory can be changed by the memory map, defined within a PCR. The address converter is only existing once for all tag cache trackers. Parallel requests can be handled, since the address conversion only takes one clock cycle, and the requests have at least a clock cycle delay. The data array holds all tags, which where fetched from the different memory. The cache lines within this array, have a size of 512 bits. The meta data array stores the meta information to the cached tags, like the validity of the tag, the address of the cache line in memory and an additional value, which indicates, if the tag is located in the DRAM or BRAM. The tag cache tracker is the managing unit, which issues the memory requests for data and tag operations, as well as data array and meta data array transfers. The tag cache can consist of multiple trackers, in order to increase the performance by simultaneously serving TileLink requests from the L2 cache bank. The inputs of the tracker are the arbitrated TileLink acquire, release and grant channels. Also the output of the address converter and the interfaces for the meta data array and data array, as well as the TileLink output for memory requests, are connected to the tracker.

Tag Cache Tracker. The tracker is the central control element within the tag cache. It handles all incoming requests from the L2 cache and issues memory requests to the TileLink/NASTI converter. To operate with tags in a transparent manner to the core, it is necessary, to perform memory requests for tags in a dedicated control logic. From the outside, the tag cache appears like any other cache, which requests data from the memory. The L2 cache is therefore ignorant to tags. The tag cache tracker assembles data and tags, when new data is loaded or performs dedicated tag write backs, in case the tag data array within the tag cache is full. All this control logic, to manage the sequential memory transfers for data and tags, is located within the tracker.

The tracker becomes active, once a valid TileLink acquire or release is issued from the L2 cache. A TileLink acquire can either request a memory read operation from the memory, or a write operation to the memory. However, in case the L2 cache performs a voluntary release, a memory write operation is issued, using the TileLink release channel. Which channel is used for write operations, depends on the caching strategy. If the write trough strategy is applied, data is written with the acquire channel, every time the caches are modified. In the case of the write back strategy, the TileLink release channel is used to write data to the memory. The Rocket chip operates using the write back caching strategy. This is true, because the L2 cache only needs to perform a memory write, if the data within the cache needs to be written back to the memory. This is the case, if either the data array can not hold anymore data or a cache line with the same address tag needs to be exchanged. The tag cache tracker supports acquire read and write, as well as release write operations. As soon as one of these channels is valid, the processing of the tag cache tracker is initiated.

The operation is based on multiple phases. The first phase is the processing of the original data request. This means that data is either written to or read from the memory. In this process, no tags are transferred between tag cache and memory. In the second phase, the meta data of tags is read from the meta data array, in order to distinguish cache misses from cache hits. This information is important, to evaluate, if tags need to be read from memory or written back, before new tags can be read. Also, the tags are read from the data array, in order to assemble the response to the L2 cache in case of a memory read operation. The third phase is the tag request, which is only performed in case of cache misses. In this phase, the tags are either written or read from memory.

All outgoing requests to the memory are performed with the TileLink client interface. More specifically, the acquire channel is used for both read and write requests. The addresses, within the incoming TileLink channels are not valid for the outgoing memory TileLink interface. The reason is, that the incoming addresses are containing the core addresses. This address does not match the physical address, if the memory map was modified. As already shown in Table 7, the BRAM is located on physical address 0x0000000, while the DRAM is located at 0x40000000. If the memory map has any other configuration, the address would be wrong if the memory is accessed. Also, the tracker has to know, which memory is currently used, in order to manage the meta data and tag requests. Therefore, an address conversion can not be performed after the output of the tag cache tracker. The converter always places the physical address of the request on the physical_address input wire of the tracker. Since the physical address of the memory is constant, the tracker can always pinpoint the requested memory, and perform address calculations of the tags properly.

The transfer sizes for data and tags are equal. The data array in the tag cache contains cache lines with a size of 512 bits. This exactly matches the data transfer size of 4 TileLink beats, if no tags are transferred. This means, that if tags are read from the memory, the 512 bits of tags are matching to more 64-bit data bits, than necessary for a single data requests. Therefore, the tag cache already contains the tags for the next couple of memory data requests, once an address, in the same range was read. With Formula 3 this number of data blocks, that can be served with a single tag memory request, is calculated. A data block is identified as the number of bits, transferred with one memory request.

$$Blockcount = \frac{TagCachelineSize_{bits} * 64}{DataCachelineSize_{bits} * TagbitsPerWord} \tag{3}$$

We apply 4 tag bits per 64-bit of data and so one tag request serves 16 data requests, which is equal to a memory region of 1 kB . In our case, both $TagCachelineSize_{bits}$ and $DataCachelineSize_{bits}$ are equal to 512 bits. The additional memory requests because of tags, can be significantly reduced, by transferring this much tags in a single operation. Therefore, the tag cache improves the performance of the tagged architecture.

Furthermore, the address of the tag block, which includes the tags for the requested data block, needs to be calculated within the tag cache. Therefore, the index of the data block is calculated by Formula 4.

$$Blocknum_{data} = \frac{Address}{DataCachelineSize_{byte}} \tag{4}$$

For this calculation the physical address of the requested data and the transfer size, given in bytes, is used.

With this information, the index of the tag block within the tag memory area can be calculated, using Formula 5.

$$Blocknum_{tag} = \frac{Blocknum_{data}}{Blockcount} \tag{5}$$

Finally, the address of the tag block, which is needed to load the tags for the requested data block from memory, is calculated by Formula 6. The result of the formula is the physical address of the tags. Note, that $Tagbase_{phyical}$ denotes the start address of the tag section in the chosen memory, as listed in Table 9.

$$TagBlockAddr = Blocknum_{tag} * TagCachelineSize_{bits} + Tagbase_{phyical} \tag{6}$$

State Machine

The tag tracker is controlled by a state machine, which manages the previously mentioned phases of operation. Figure 14 summarizes these three phases. As it can be seen in the figure, two different operation modes are implemented, based on the transaction type of TileLink. Either data is written or read from the memory. On the top, the read operation is visualized, while on the bottom, the write operation is shown. The phase one is reading

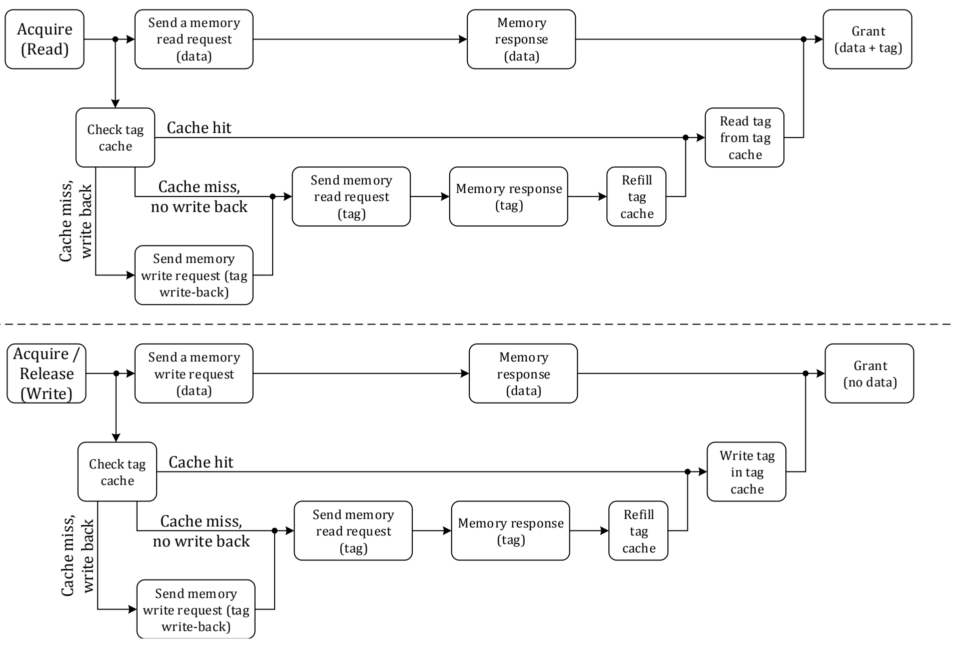

Figure 14: A summarizing visualisation of the different processes and modes of the tag cache tracker. On the top, the operation mode for a read request is shown, while the write opration mode is presented on the bottom.

data from memory or writing the data to the memory. This is the original request handling, which is initiated by the L2 cache. The phase two reads meta data from the meta array in order to decide tag cache misses or hits. This phase is performed in parallel to the phase one, and if phase one is finished, the tags are either read from, or written to the data array and combined with the data. Phase three is executed in case of a cache miss and is the same for both write and read operations. If the tags need to be replaced in the data array, the tags are written back to the memory. Afterwards, the new tags are read from the memory and after the response is received, the data array is refilled with these tags.

The transaction blocks of each visualization are matching to the corresponding states in the tracker state machine. In Figure 15, the state machine of the tracker is presented.

When no valid acquire or release channel is existing, the tracker remains in the *IDLE* state. When a request is performed by the L2 cache, the state is transferred to *Meta Read*. While this state change is performed, the first phase is executed. In this phase, the original data request is handled. This means, that the tracker performs read or write requests to the target memory in order to process the original request. If data is read, it will be buffered until it can be merged with tags and returned to L2 cache. If the data is written, the data is separated from the tags with every incoming TileLink data beat and redirected to the outgoing memory acquire channel. In this case, the tags are buffered until they are written in the tag cache data array. For this data transfer, no address calculation needs to be performed and the physical address is used for requests.

In the state *Meta Read*, a meta array read request is issued, in order to gather the meta information of stored data in the data array. This is the first part of the second operation phase. Afterwards, the state is transferred to *Meta Response*, where the tracker waits for

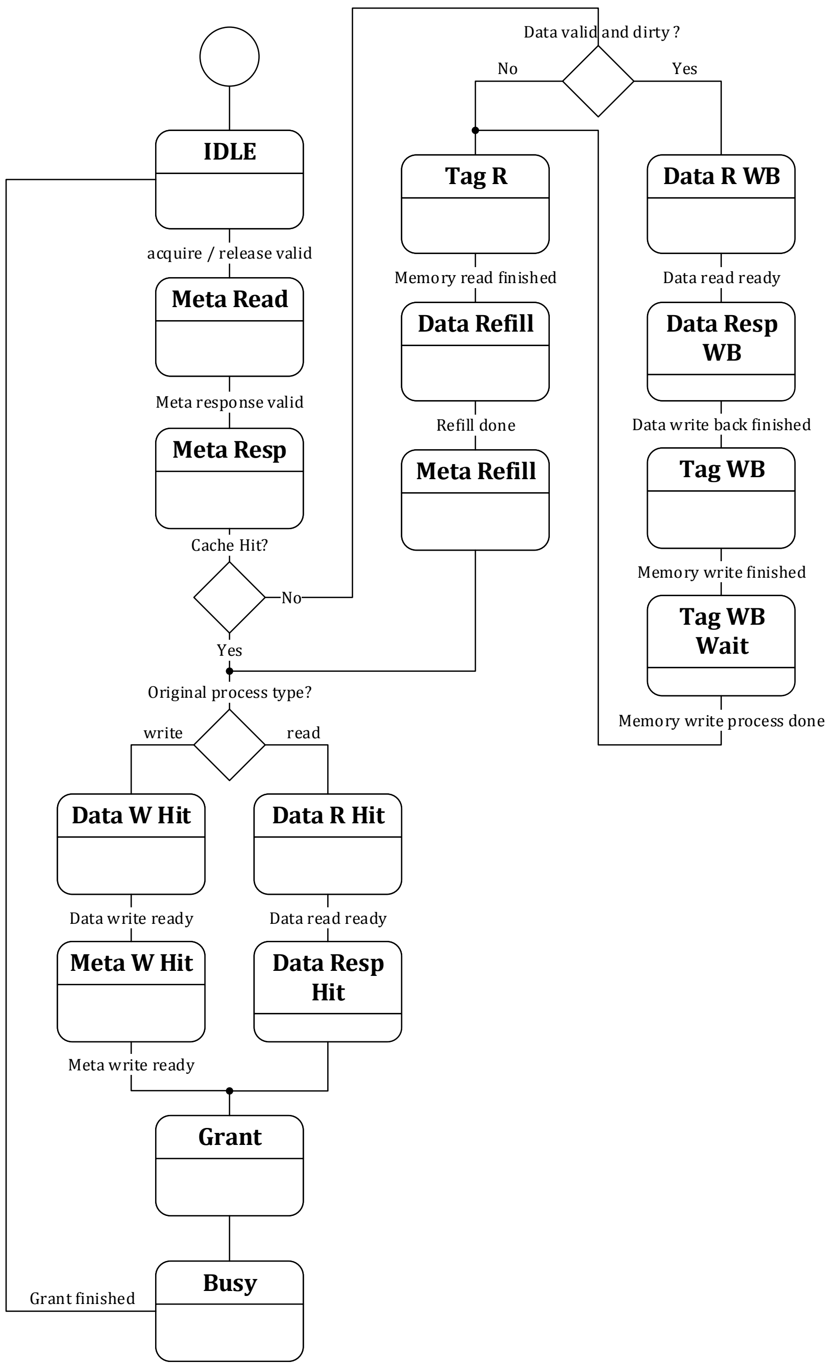

Figure 15: The state machine of the tag cache tracker.

the data from the meta data array. In some cases, this request could take a couple of cycles, if for example a second tracker also requests the meta array at the same time. Therefore, it is necessary to switch in the waiting state.

In the *Meta Response* state, further operating modes are evaluated as soon the meta response is valid. Based on the data in the response, different follow up states are possible. First, it is determined, whether a cache hit was issued. This means, that tags for the requested data are existing within the data array of the tag cache. Important to note is, that the cache hit case never requires a tag memory request. The phase three is requesting tags from memory and is therefore never entered on a cache hit. If the check results in a cache hit, the state machine evaluates, if the original request was a write or read process. In case of a write process, it translates to the state *Data Write Hit*, where the tags are written to the caches data array. If it was a read process, the tags are available within the data array and the state machine translates to *Data Read Hit*. However, if a cache miss was detected, the tags for the requested address are not existing within the data array. But since it is possible, that valid tags for a different address are stored on the same cache line, the evaluation of the meta tag is performed. If the tag indicates, that the stored tags are valid and previously modified, a write back has to be performed, before the new tags can be read from the memory. The state machine translates into *Data Read Write Back*. In case no valid tags where stored or they where not modified from previous operations, no write back is needed and the tag can be requested from the memory within the state *Tag Read*. In other words, tag cache misses always result in tag read or in tag write back + tag read memory requests, depending on the previously described factors. This scenario is the opposite of tag cache hits, where no additional memory requests are performed.

The state *Data Write Hit* performs a data array write request, with the tags, that where received from the L2 cache. Therefore, the state needs to wait for the completion of the original memory request. If phase one, where the original data request is handled, is finished, all necessary tags are available and can be stored in the data array. The tags are shifted to the correct position within the data write request, by calculating the offset from the physical address. This is necessary, since the input of the data array is 128 bits wide, but only 32 tag bits are received from the L2 cache. Only this small fraction of the cache line is then overwritten with the new tags. The state machine translates in the state *Meta Write Hit*, once the write request was issued.

In the state *Meta Write Hit*, the meta array is actualized. The cache line is set to dirty, which implies, that the tags where modified. Also the information, to which memory this tags are assigned is written as meta data. If the physical address is translating to the physical address of the DRAM, the meta tag bit is set to "1". After the meta array is written with the new information, the state machine continues with the *Grant* state.

The previously described state transitions are performed, if a cache miss was identified and the original process was a write. However, the state *Data Read Hit* is responsible for handling the case of a cache hit and read process. As the name suggests, the data array is read in order to retrieve the tags, needed for the request. Since the data is existing and valid within the data array, the data array read request is issued. While the read is performed, the state machine translates to *Data Response Hit*.

In the state *Data Response Hit*, the response of the data array is received. Once the response is valid, the reverse operation of writing tags to the data array, as in the state *Data Write Hit*, is performed. The retrieved tags are shifted by an offset, matching the requested tag portion and stored in a temporary buffer, which will be combined with the data in the *Grant* state. Once this state is processed, the state machine translates in the *Grant* state, which performs the data and tag response to the L2 cache.

The state *Data Read Write Back* is entered, if a cache miss is detected and the tags within the requested cache line are modified by previous operations. In this case, the full cache line is read from the data array, because it has to be written to the memory. Reading the data array also requires 4 beats, since one valid response only holds 128 bit of tags, but 512 bits are required before the tag memory write operation can be issued. The data is stored in a temporary buffer, until the whole data is retrieved. In order to wait for completion, the state changes to the *Data Response Write Back* state.

While the *Data Response Write Back* state is active, new data array requests are issued, until all bursts of response data are collected. Once this condition is true, the state machine continues with the state *Tag Write Back*.

The state *Tag Write Back* is part of the third phase of operation. In this state, the write back operation of tags is performed. Therefore, an acquire message, with the previously gathered tag data is assembled. Before the acquire channel can be set to valid, the original data process needs to be finished. Data and tag requests are performed sequentially. The address, where the tag shall be stored within the memory, needs to be calculated, using the information, to which memory the tags shall be written. Once the start address of the tag section for the given memory is gathered, the exact location is calculated using Formula 6. This address is used for the acquire channel. Since writing all tag data to the memory also requires four data beats, this state waits for the completion, until it translates to the state *Tag Write Back Wait*.

The state *Tag Write Back Wait* is required, to wait for a valid grant channel, issued from the memory controller. In this case, the grant signals the completion of the write process. Only if the grant channel is valid, the state machine is allowed to continue with the state *Tag Read*. Since the tags are written to the memory, in order to create place for new tags in the data array, the requested tags need to be read from the memory. This is why a tag read process always follows the tag write back process.

As already mentioned, the state *Tag Read* handles the memory request to read tags and is also part of the phase three. Since this state can be also entered directly after the state *Meta Response*, it also has to wait until the original request was performed. If this is the case, the tag read memory request is issued. Therefore, the acquire read request is assembled and sent to the memory controller. The memory responds with tag data within four TileLink beats on the grant channel. The tags are stored in a temporary buffer, every time a valid beat is received. Once this process is finished, the state machine translates to the state *Data Refill*.

In the state *Data Refill*, the data array is re-filled with the received tags. In total, four beats of data are stored in the data array, by issuing multiple data write requests. The write address is incremented every time, so that the cache line is filled properly. After the refill is finished, the state *Meta Refill* is entered.

Within the state *Meta Refill*, a similar operation is performed as in the already described in the *Meta Write Hit* state. The difference is, that the meta data is marking the cache line as not modified. The tags, which are read from memory, are not dirty, at the moment they are retrieved. The tracker remains in this state, until the meta array write request is finished. In order to conclude this path of the state machine, the follow-up state is evaluated. If the original request was a write process, the state *Data Write Hit* is entered. If not, a state translation to *Data Read Hit* is performed. This is possible, since the current state of the tracker equals the situation when the first meta array read resulted in a cache hit.

When the *Grant* state is entered, all necessary steps in order to respond to the L2 cache with a valid grant, are performed. The grant channel is either responding the assembled data and tags, if the original request was a read process, or notify the completion of a write process. The grant for a read process, like the acquire data process, includes four

TileLink beats. The grant for a write process includes no data and just informs the L2 cache, that the operation is finished. The state machine translates to the *Busy* state, in order to wait for the end of the grant operation.

The *Busy* state is always the last state, the tracker enters, before a new process can be initiated. Once the grant is finished, the state translates back in the *Idle* state, in order to signal, that the tracker is ready to start a new process.

7.3 Tagged Security Policies in Core

In the previous sections, we described the implementation of tagged memory within the untethered lowRISC chip. Also, a brief introduction of the necessary modifications to the L1 data cache and the core is given. As a result, all components outside of the core, necessary to support tagged memory, including the tag cache, are fully described. However, until this point, the system is only able to operate with tags within the memory and caches, but the security polices are not implemented yet. This means, that no automatic propagation of tags is performed. The framework to implement more sophisticated applications with tagged memory, is given with this changes. In this thesis, we also implement tag awareness of the core in order to support the proposed tagged security polices as an application for the tagged architecture.

In order to implement the proposed counter measures in the core, the pipeline has to be modified. A part of this modification is the addition of the proposed tag ALU and tag check unit. This two new components manage tag propagation and enforce the security policies by triggering tag traps on wrong behaviour of the system. To support a tag aware core, the interface to the data cache is modified, while the cache itself needs to process tags properly, when data is exchanged. The core is also extended with a tag register file, which matches tags to the corresponding register.

The modifications of the data cache range from increasing data widths within the cache architecture, to specific handling of rules for the tagged security policies. Furthermore, the data cache is responsible for handling the IO invalid tagging mechanism for the function pointer protection policy.

In this section, we will first describe the modifications of the core and how the security policies are enforced. Afterwards, the details of the L1 data cache modifications are explained.

7.3.1 Core Modifications

Before we dive into the implementation details of the core, the general architecture of a the Rocket core is shortly elaborated. The Rocket core is part of a so called tile and can exist multiple times within the Rocket chip in order to allow multi core architectures. Each tile includes an instruction cache, a data cache and the core itself. The instruction cache reads program code from the memory and transfers the instructions to the core. The data cache handles data read and writes. While the caches perform memory related work, the core processes the instruction. The core operates on the 32 core registers, by executing instructions within a core pipeline. Core registers are 64-bit wide and can hold arbitrary data. As every other common processor core, the core within the Rocket chip includes an ALU, as well as a divider and a floating point unit (FPU), if configured. Also, instructions to alter the control flow, like direct and indirect jumps and conditional branches are handled within the core. Every computational instruction takes data within registers or immediate values as operands, while the result is always stored in the destination core register. Within the RISC-V ISA no instruction, which performs memory requests and then executes a computation or vice versa is existing. As a result, all computational operations always operate on core registers only. Memory operations are performed by sending requests to the data cache by executing dedicated memory load/store instructions, as described in Section 3.2.

In order to enable tag propagation, a tag register file is added. Its load and store operations are synchronized with the core register file and also the addressing scheme matches the addressing of core registers. Every core data register is combined with a tag. Tag propagation and computation is performed within the tag ALU. Its computation does not match the ALU computation, but follows the rules of the security policies. The tag check unit is introduced to execute tag traps and stall the pipeline if a security policy is violated. In the following, the processor pipeline is described in detail.

The Rocket core operates with a single-issue in-order 5-stage pipeline. These five stages are instruction fetch, decode, execute, memory and write back. In the instruction fetch stage, the next instruction to be executed is loaded from the memory into the core instruction register. The processor interprets the instruction within the decode stage and sets the control registers appropriately. In the execute stage, the instruction is executed. The memory stage handles memory responses, if a memory read was issued. The write back stage is responsible for writing the result of the instruction in the destination register. Figure 16 gives a detailed view of the pipeline and which components are accessed at which stage. The components, added for the tag handling, are marked in blue color. Note, that the figure only shows 4 pipeline stages. This is the case, because the L1 instruction cache is responsible for the instruction fetch stage. The instructions are automatically inserted into the core pipeline by this instruction cache. If the L1 instruction cache needs to access the memory, the pipeline is stalled in the meantime, until the instruction data is valid. However, if the pipeline is not stalled, the stage transition is performed by updating the instruction, the control and the data registers of the following stage, with the contents of the previous stage. Therefore, the pipeline advances each clock cycle.

Decode. The first relevant stage for tag handling is the decode stage. As it can be seen in the figure, the data form the requested operand registers is read. The result is then stored in the *ex_rs* register at the next clock cycle. The address of the operand registers to be read, is parsed from the instruction by the decoder. Since the tagged architecture adds a tag register file, this registers are read at the same time with the same index as the regular register file. The result is, analogous to the data registers, stored in the *ex_tags* register, which will be valid on entering of the execution stage. Using this technique, the operation with data and tags combined, is transparent and it is ensured, that always the correct data/tag pair is read.

Execute. The execute stage is the main computation phase within the pipeline. The operand register data is forwarded to the chosen computation element. This is for example the ALU, the divider or the FPU. Also, memory read or write requests are issued to the L1 data cache in this stage, if a corresponding load or store instruction is executed. The result of the computation is stored in the register *mem_reg_wdata* with the next clock cycle, in order to allow further processing. The operand input data can be coded within the instruction or it can also be the current program counter. The instruction AUIPC for example, requires the program counter and a constant as input operands of the ALU. The original register value form the second operand register is always directly stored in the register *mem_reg_rs*2. Memory store instructions always use the data of this register. However, the operation for tags is done differently.

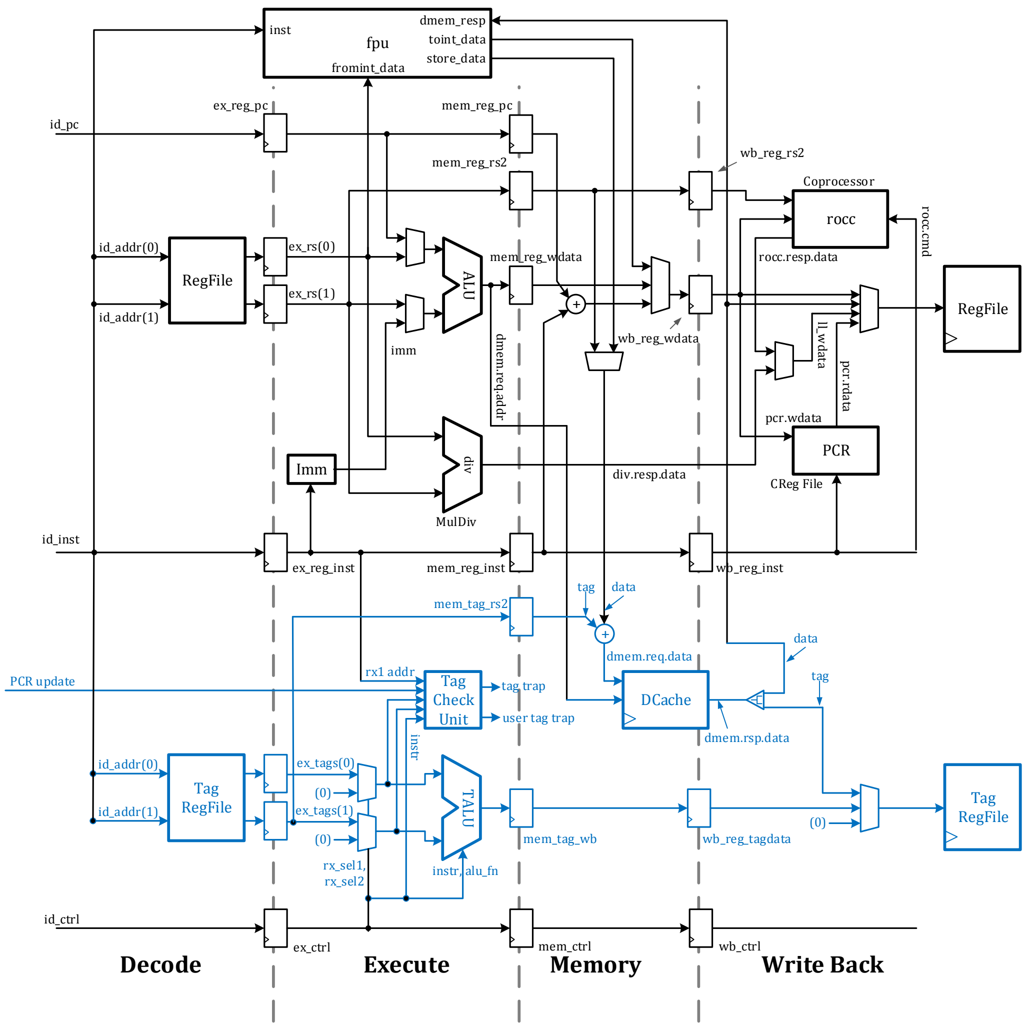

Figure 16: A visualisation of the pipeline within the Rocket core and the modifications introduced in order to manage and enforce tagged security policies. The pipeline stages decode, execute, memory and write back are seperated by dotted lines. The newly introduced components, and modified wires are marked with blue color.

Both tag propagation and the check of tag values is performed within the execute stage. The tag ALU, as the ALU, has two operand inputs, which are the tags of the selected registers. The tag ALU performs the propagation of tags by the given rules of the security policies. The result is stored in the register *mem_tag_wb* at the next positive clock edge and is then used for further processing. The other input of the tag ALU is the information about the current instruction. An important requirement for the tagged operations within the core is, that all states and values of tags have to be well defined, for every possible situation. It is not allowed to include undefined behaviour or values for tags. For example, the initial values of the tags, are always set to zero. A random value is not allowed, since the tag ALU would propagate the tags through the system, introducing wrong tag values and as a result, may cause false traps. This is the reason, that the tag operand input of the tag ALU is multiplexed. The values, read from the tag register file is only used, if the instruction clearly defined, that the values where really selected and correctly read. If this is not the case, the input will remain zero.

To enforce the security policies, the tag check unit is introduced. As with the tag ALU, the main inputs of the tag check unit are the tags of the two operand registers. Also the update channel form the PCR controller, the current selected jump register, as well as the current instruction are connected to this unit. If a trap is triggered by the unit, the pipeline is finishing the current instruction in the pipeline and then jumps to the trap handling procedure.

Memory. The memory stage is responsible for issuing memory read and write requests to the L1 data cache. Also, special write back data is assembled within this stage. Therefore, the register *wb_reg_wdata* is updated either with the integer output of the FPU, the output of the ALU, or with an addition of the current program counter and an immediate, coded in the instruction. The address for memory access operations is an output of an address calculation, performed within the ALU. The data source for memory write requests is multiplexed either from the FPU or from the previously mentioned *mem_reg_rs*2 register. Since memory write requests always consist of data and the corresponding tag within our implementation, the request data is assembled by combining the data and the output of the *mem_tag_rs*2 register. In addition to assembling data and tags for the write back stage, the memory stage also forwards the tag, stored within *mem_tag_wb* to *wb_reg_tagdata*. This register will be used, if the result tag has to be written back in the destination tag register.

Write Back. The last stage of the pipeline is the write back stage. Within this stage, the data and tags are written back into the destination registers, if a computation instruction with result was executed. Also, the response of the data cache is processed within this stage, if a memory read instruction was executed. An additional feature of the Rocket core, is the availability of so called Rocket accelerators. This elements can be seen as additional co-processors with special capabilities. Therefore, a Rocket accelerator can be used as a cryptographic co-processor, for example. In this stage, the input and output of rocket accelerators is handled. However, since we did not implement tag support for these accelerators, it is not relevant to elaborate further details on this topic. The data, written to the register file, is chosen from the *wb_reg_wdata* register (results of the computation), from the response of the data cache, from the Rocket accelerator, the divider, or from the CSR registers. CSR write or read instructions are entirely performed within the write back stage. Note, that no tag support for CSRs is necessary yet, to enforce our proposed security policies. The tags are written back in the tag register file, by choosing between the calculated tag of the previous stages from register *wb_reg_tagdata*, the tag portion of the data cache response or a constant zero, in all other cases. The response of the data cache always consists of data and tag, which is why this two parts are split and then forwarded to the correct target register file.

Tag ALU. As already mentioned, the tag ALU applies the tag propagation rules for the security policies, as described in Chapter 5. The rules of the return address and function pointer protection, as well as the rules for user tags, can be summarized in a combinational logic, as shown in Figure 17. The logic is composed of three layers, which results are multiplexed for the different instruction types.

The first layer handles the arithmetical and logical rules of all policies. The return address tag (Bit 1) is always set to zero, while the invalid tag (Bit 0) is always a result of the ored tag inputs. The user tag bits (Bits 3 and 4) are separately ored like the invalid tag bit.

The result is an input of the multiplexer in the second layer. This multiplexer takes either the output of logical computations or only the tag of the tag input 1 in case of a move instruction. The move instruction always moves data from the register rs1 to a destination register. Therefore, only the tag of this register is used without modification.

The result of this combinational logic is transferred to the multiplexer in the third layer. This multiplexer performs the selection of the final output. It is responsible for the handling of call (JAL/JALR) instructions. If a jump instruction is performed, the new tag on the output of the tag ALU will be set to 4b'0010. This means, that every tag bit is set to zero, except the return address bit. It is ensured, that the return address tag will be set for the return address, after a jump instruction. If no jump instruction is executed, the output of layer two is taken as final output tag. This concludes the propagation of tags within the core. Note, that the rules for memory load and store instructions can not be applied within the core. This is performed within the data cache, since only there, all necessary information is given in order to apply rules to partial memory load and store instructions. This implementation is described in Section 7.3.2.

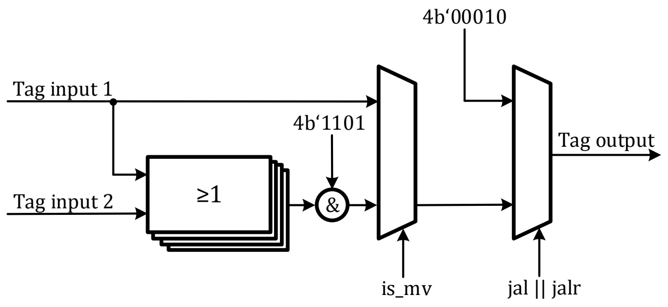

Figure 17: The three combinational logic layers of the tag ALU. Note, that multiplexers select the value on the top, if the select input is logic "1".

Tag Check Unit. The tag check unit enforces the security policies in case a violation of the good behaviour is detected. This is done, by triggering two different types of traps. The unit holds a register called *reg_tag_ctrl*, which identifies, which rules are active and shall be enforced in the current state of the processor. This register is updated by the PCR control, if the address of this newly introduced PCR register is written. The register is located at address 0x800. Table 10 shows the possible settings of the tagged security mechanisms. The checks for the security policies and for the user tags are performed parallel to each other in a combinational logic.

If the return address tag check is enabled, the return instruction (JALR dst[x0], rs1[x1]) is monitored. It is validated, if the return address bit is set in the tag of the input register rs1 (tag input 1). It is not necessary, to perform this check on the tag of input register rs2 (tag input 2), since the jump instruction only takes the value of register rs1 for an indirect jump. If the tag bit is not set and the check is activated, the condition, that the chosen register is the register x1, has to be satisfied. Return instructions always read the value from register x1. Only if all those conditions are satisfied, the current situation would lead to an invalid jump.

A similar computation is performed for the function pointer protection. If the invalid tag check is enabled, the chosen register is not the return address register and the invalid tag bit is set to “1”, an invalid behaviour is detected. Note, that in both cases, the actual trap signal is only triggered, if the current instruction is actually a indirect call instruction (JALR/JR). If this is the case, the output *tag_trap* is set and the processor initiates the trap handling sequence.

The user tag trap is triggered, if the user tag check is enabled and any of the tag bits 3 or 4 of both inputs is set. In this case, the output *user_tag_trap* is set. This trap triggers independent from the currently executed instruction.

Bit	63 : 4	3	2	1	0
-	RFU	user tag check	invalid tag generation	invalid tag check	return tag check

Table 10: Bit representation of the tag control PCR. If a bit is set to logic “1”, the corresponding feature is enabled. The bits 63 to 4 are currently Reserved for Future Use (RFU).

7.3.2 Data Cache Modifications

The data cache is responsible for the handling of memory access operations. In order to support the tagged architecture and the proposed security mechanisms, some changes need to be performed to the existing untethered lowRISC implementation. The data cache operates with a four stage pipeline, which coordinates read and write requests. The data cache is connected to the core with the HellaCacheIO, which is similar to the TlieLink interface. It is also a decoupled interface type, which allows ready and valid signals to coordinate data flow. The requests from the CPU are sent using the request channel. This channel always includes the address of the data to be accessed, as well as the memory command and the access type. The command denotes, if a memory store or load is performed. The access type denotes a double word, word, half word or byte access. If a memory write is performed, the request channel also includes the data to be written. The response channel signals the core the completion of the memory access and can also either include data or not. As already mentioned, data transferred with this interface always includes the data and tag bits, which leads to a total width of 68 bits. The communication with the L2 cache is performed using the cached TileLink interface.

The data cache also handles IO data transactions. If data is transferred between the IO and the data cache, the uncached TileLink IO interface is used. This means, that no IO data transfer is ever cached, since the L2 cache is bypassed and the L1 data cache also does not keep local copies of IO data. The cache consists of the miss handler, the IO miss handler, a meta data array, the actual data array and the AMOALU. The miss handler is comparable to the tracker within the tag cache. Memory requests on misses and write backs are performed by this unit. The IO miss handler is responsible for managing the data transfer between peripheral devices and the cache. The data cache stores data and tags together in cache lines with the size of 544 bits. This is equivalent to the size of the cache lines of the L2 cache and whenever data is requested from, or written to the L2 cache, the whole cache line is transferred. The meta data array contains the information, if cache lines are dirty and valid. Finally, the AMOALU handles partial loads and stores, based on the given memory access type.

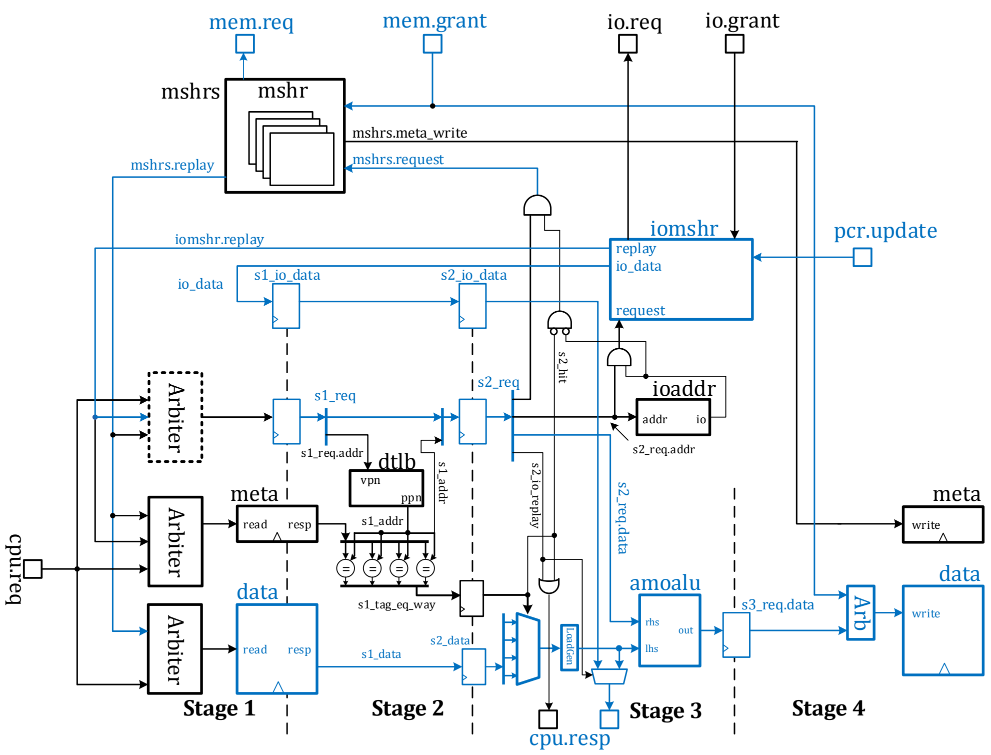

Figure 18: The pipeline and the components of the L1 data cache within the Rocket core. The four stages of the cache are seperated with dotted lines. Components and wires, that where modified in order to support tagged memory and security policies, are marked with blue color.

The components and the stages of the data cache are shown in Figure 18, in a simplified form. The first stage is receiving the requests from the core. Both the data and meta data array is read in this stage, independent of the memory command. The whole request is stored in the register *s1_req* on the next positive clock edge. Replay data form the IO miss handler is stored in the register *s1_io_data*, if needed. A replay is always executed, if data shall be read from the memory, but a cache miss was issued. In this case, the data is requested from the L2 cache and a new replayed core request is generated, which starts the execution of the first stage again. Then, the whole process is executed all over, but this time the data is existing within the data array. In the case of IO replays, the data is located in the register *s1_io_data* at the end of this stage. Note, that all blue marked wires were already existing in the architecture without tags, but were extended to a total size of 68 bits, in order to transfer both data and tags at once.

The second stage performs the virtual to physical address translation. Therefore, a Translation Lookaside Bufer (TLB), denoted as *dtlb* is used. The resulting address is then saved in the *s2_req* register. If no address translation is necessary, the original address from the *s1_req* is stored in this register in the next clock cycle. The data, which is read from the data array is forwarded to the register *s2_data*, while IO replay data is saved in the register *s2_io_data*.

Stage three issues the response to the core. This is valid for both read and write operations. If a read was performed, all data is available at this point within the pipeline. The response data is multiplexed either from the output of the LoadGen, or from the IO data, which was received from a peripheral device. The LoadGen is a combinational logic, which correctly assembles data and tags, by given rules, in case of a partial load or tag read instruction. The input of this module is the data within the *s2_data* register. If a memory or IO read operation is performed, the work of the data cache is completed, as soon the response was handled by the core. In order to decide, whether the request is a memory or IO access, the request address is compared to the IO map. If an IO access was detected, the request is initiated using the IO miss handler. However, if a memory write request is performed, the response is also sent to the core, but then the forth stage of the data cache gets active. This stage is needed to perform the write to the data and meta data array, but the CPU does not need to wait for its completion. If this is the case, an additional step is necessary, before the pipeline continues with stage four. This additional step involves the AMOALU, which handles the partial store operations, as well as the atomic memory operations. Therefore, the inputs of the AMOALU are the data and tags, read from the data array, and the data to be stored within this data field. The previously existing data is the input *lhs*, while data to be stored is the input *rhs*. The output of the AMOALU is the correctly modified data and tag, to be stored back in the data array. This output updates the register *s3_req_data*.

The last stage of the data cache pipeline handles the write back of data in the data and meta data array. This stage is only entered, if a memory write operation is requested, or when a data is requested from the L2 cache, and the response is received. The meta data is gathered from the miss handler. The data, which is written to the data array, is multiplexed either from the previously calculated output of the AMOALU, or directly from the memory response, which is received and handled by the miss handler. At the end of this stage, the cache line holds valid and modified data and tags from the store instruction.

IO miss handler. As mentioned above, the IO miss handler manages the data transfer with peripheral devices, using the uncached TileLink interface. This component also implements the functionality, necessary to satisfy the function pointer protection policy. Since this is the only module, which directly communicates with the IO, the appropriate tag is inserted at this point. The IO miss handler therefore includes the tag control register, which is updated by the PCR control, if the corresponding register is modified. As already described in Table 10, bit 2 defines the state of the invalid tag generation. If this feature is enabled, all data from IO needs to be tagged as invalid. In this case, the tag 4b'0001 is appended to the IO data. Otherwise, the tag is set to 4b'0000. This combined tag and data output is then used within the core response, if a IO read is performed. The tag is removed, before data is sent to the IO device, since tags are not supported on IO devices.

LoadGen and AMOALU. This two modules handles the the partial load and store rules, given in Chapter 5 and the AMOALU also performs atomic memory operations. Both modules solely perform combinational operations and deliver the correctly assembled output for the given situation. A memory access can either read or write a full 64-bit double word, a 32-bit word, a 16-bit half-word, a single byte or only the tag portion of the data field. Therefore, the access type of the memory operation is analysed. The LoadGen prepares data and tags, when data is read from the cache. All access types always concatenate the 4-bit tag to the read data. In the case of a tag message type, the data portion is filled with the tag instead of data. When partial loads are performed, the data input of the LoadGen is first shifted, masked accordingly and afterwards the tag is concatenated to the response. As mentioned within the return address protection, partial loaded data never includes a return address tag, set to “1”. In this case, the return address tag is set to zero, before the tag is added to the response. For all other policies, the tag does not have to be modified on partial loads.

The AMOALU is modified, in order to handle partial stores of tags, respectively. Therefore, the data is shifted and masks are calculated, to perform the corresponding operation. The input *rhs* represents the data, which shall be written to the data field. This data also includes the tag. The input *lhs* is the data, which is read from the data array in previous stages. In the case of a full double word write instruction, no combination of *lhs* and *rhs* performed and only *rhs* is used to form the output data and tag. However, on a partial store operation, the data of *rhs* is combined with the data of *lhs*, in order to apply single byte modifications, without altering the whole 64-bit data field, for example. Note, that if the message type indicates a tag store instruction, the tag portion of *rhs* is transferred to the output, without changing the data, existing in *lhs*. On partial stores, the output tag is also a combination of the tags in *lhs* and *rhs*. The combinations are logical or-operations. However, the return address protection policy requires a reset of the return address tag in case of a partial write. Without this additional rule, a partial store could never clear the return address tag, if there was a set return address tag in *lhs*. Therefore, this tag bit is masked out, before the combination is performed.

7.4 Software

The previous sections covered the hardware modifications, which are necessary in order to implement tagged security policies on top of the Rocket chip. Although the architecture and the policies are designed to be as transparent as possible to the software, a few software modifications are required to archive full support. We already mentioned in Chapter 5, that some special cases are not handled entirely by hardware or would cause false traps as result. These special cases occur within bootloaders and the Linux kernel. Also, the interrupt and trap handling at the machine privilege level has to be modified, in order to handle tag traps. Note, that in general, user applications, that run on top of the Linux kernel or in a bare-metal mode, do not require changes in order to support the tagged architecture. Also, no compiler changes are performed, except the addition of the new store- and loadtag instructions. Modifying the tag control register changes the behaviour of the processor, while handling the security policies. Therefore changes to this register are necessary during runtime in order to archive full support. This register keeps the value, set by software, even after a soft reset is performed.

In the following sections, we outline all necessary software modifications within the bootloader and the Linux kernel.

7.4.1 Bootloader

As described in Section 7.1.3, the boot procedure of the untethered lowRISC is consisting of two stages. The first stage bootloader prepares the memory, by loading either the second stage bootloader (BBL) or an arbitrary user application form the SD card. After this preparation, a soft reset is triggered, to launch the loaded program. The second stage bootloader is responsible for loading the Linux kernel into the DRAM and start its execution. The BBL always continues to run underneath the Linux kernel in order to handle interrupts, traps and peripheral device accesses. The initial value of the tag control register

```
int main (void)
{
  FIL fil;                  /* File object */
  FRESULT fr;               /* FatFs return code */
  uint8_t *boot_file_buf = (uint8_t *)(get_ddr_base())
                            + 0x38000000;
  uint8_t *memory_base = (uint8_t *)(get_ddr_base());

  //Modify tag control register to support binary loading
  //Checks on / invalid tag generation disabled
  write_csr(0x800, 0x3);
  ....
}
```

Listing 8: Bootloader initialization.

```
static void enter_entry_point()
{
    write_csr(mepc, current.entry);

    //Checks on / invalid tag generation enabled
    write_csr(0x800, 0x7);
    uart_send_string("Start Linux\n");
    asm volatile("eret");
    __builtin_unreachable();
}
```

Listing 9: Linux entry code within BBL.

is 0x7. This means, that the return address and the function pointer protection checks, as well as the invalid tag generation is enabled. The only check, which is not enabled after a reset, is the user tag check, since this is not required for the tagged security policies.

The bootloaders both load binaries from the IO, which are then executed. Since the invalid tag generation is enabled, this would cause invalid function pointers within the binary. This happens, because all read data and code is automatically tagged with the invalid tag bit, which causes traps, if the processor performs a jump to any function pointer. This behaviour is one of the situations, which are not handled automatically by the processor. The processor can not differentiate between the loading of a binary and user data from the IO. To solve this issue, the invalid tag generation has to be switched off in the beginning of the first stage bootloader. This modification is shown in Listing 8, where a write of value 0x3 is performed to the tag control register. This changes are performed in the file *boot.c*.

After the first stage and second stage bootloader finished its preparations and the Linux kernel is loaded into memory, the invalid tag generation needs to be switched on again. Therefore, all security mechanisms are enabled, while the Linux kernel is running. This is done within the BBL in the file *bbl.c*. The function *enter_entry_point* performs the jump to the Linux kernel start address, by exchanging the value of *mepc* with the entry address and execution of the *eret* instruction. Listing 9 shows the applied changes.

Finally, the support of tag traps has to be added to the BBL. The core can trigger two new types of traps, namely the tag trap and the user tag trap. The tag trap is triggered, if a violation of the return address or function pointer protection is detected. The user tag trap is triggered, when a user tag is set and the user tag check is activated. In the core, this is realized by the addition of this new traps, which are identified by the trap numbers 12 and 13, respectively. Whenever a interrupt or trap is triggered, the processor jumps to the entry address, corresponding to the current privilege level. The entry of the user mode is located at address 0x100, the supervisor mode at 0x140, the hypervisor mode at 0x180 and the machine mode at 0x1C0. The interrupt or trap number is stored in the CSR register *mcause*. The BBL has to handle this traps and interrupts correctly, which is why it has to be aware of the additional trap identifiers. The assembler routine, which is handling this entries, is located within the file *mentry.S*. To support the new traps, the trap handlers, written in C, are added to *trap_table*, as shown in Listing 10. Also, there are two defines, which denote, if a trap shall be handled in machine mode, when a trap was either caused in user or supervisor mode. If the trap occurs within the Linux kernel execution, the trap is still handled within the machine mode at the moment. This could be changed in order to allow Linux to deal with the consequences of a tagged security violation. Anyway, since we handle everything in the machine mode, the tag traps are added to the defines and identified as handled in machine mode.

The corresponding and referenced functions, which handle the tag traps, are located in the file *mtrap.c*. This functions print the trap identifier and the program counter of the instruction, which caused the trap. At the moment, this functions never return and halt the processor. The trap handlers are shown in Listing 11.

7.4.2 Linux Kernel

The Linux kernel has to be modified in order to fully support the tagged security policies. Without modification, the Linux kernel can cause false traps, which leads to an unreliable system. The issues occur, when the kernel manually replaces the return address with a new value during the code execution. This situations occur at a signal handler execution, fork operations and in general, when the context is switched and the assembly routine performs a return in order to continue execution. Another issue is similar to the problem with the binary load process of the bootloaders and the invalid tag generation. Linux is able to load binaries from the disk and execute the applications. While this processes are performed, the invalid tag generation also needs to be switched off.

In order to prevent false traps from return address protection violations, all situations, where the kernel itself causes an invalid return address tag, have to be resolved. This is the case, whenever the return address register or any return address in the memory is overwritten. To archive this, the return address tag is manually set to the value “1”. The first code segment, where this change is applied, is within the function $setup\_rt\_frame$, in the file *arch/riscv/kernel/signal.c*. When a signal is performed, the user context is saved. The return address is overwritten, when the signal returns and the user context is restored. The user code is not directly continued, but the function returns by using a trampoline code. The return address is overwritten before this return is performed. The return address is validated, by writing the value 0x2 with the instruction *stag*, to the address of *regs→ra*. Listing 12 shows the applied changes.

The second case, in which the similar problem occurs, is within the fork feature, when a thread is copied and then executed. The problem is, that the return address of the thread entry, is changed to the entry point of assembler routines. This routines are responsible for restoring the thread context and then continue with the thread execution. Within the file *arch/riscv/kernel/process.c*, the function *copy_thread* is altered as shown in Listing 13. In

```
#include "mtrap.h"

  .data
  .align 6
trap_table:
  .word bad_trap
  .word bad_trap
  .word illegal_insn_trap
  .word bad_trap
  .word misaligned_load_trap
  .word bad_trap
  .word misaligned_store_trap
  .word bad_trap
  .word bad_trap
  .word mcall_trap
  .word bad_trap
  .word bad_trap
  .word tag_trap
  .word user_tag_trap
#define HTIF_INTERRUPT_VECTOR 14
  .word htif_interrupt
#define TRAP_FROM_MACHINE_MODE_VECTOR 15
  .word trap_from_machine_mode
#define IO_INTERRUPT_VECTOR 16
  .word io_irq_service
  .word bad_trap

#define HANDLE_USER_TRAP_IN_MACHINE_MODE 0         \
  | (0 << (31- 0)) /* IF misaligned */            \
  | (0 << (31- 1)) /* IF fault */                 \
  | (1 << (31- 2)) /* illegal instruction */      \
  | (0 << (31- 3)) /* breakpoint */               \
  | (1 << (31- 4)) /* load misaligned */          \
  | (0 << (31- 5)) /* load fault */               \
  | (1 << (31- 6)) /* store misaligned */         \
  | (0 << (31- 7)) /* store fault */              \
  | (0 << (31- 8)) /* user environment call */    \
  | (0 << (31- 9)) /* super environment call */   \
  | (1 << (31- 12)) /* tag trap */                \
  | (1 << (31- 13)) /* user tag trap */          \

#define HANDLE_SUPERVISOR_TRAP_IN_MACHINE_MODE 0  \
  | (0 << (31- 0)) /* IF misaligned */            \
  | (0 << (31- 1)) /* IF fault */                 \
  | (1 << (31- 2)) /* illegal instruction */      \
  | (0 << (31- 3)) /* breakpoint */               \
  | (1 << (31- 4)) /* load misaligned */          \
  | (0 << (31- 5)) /* load fault */               \
  | (1 << (31- 6)) /* store misaligned */         \
  | (0 << (31- 7)) /* store fault */              \
  | (0 << (31- 8)) /* user environment call */    \
  | (1 << (31- 9)) /* super environment call */   \
  | (1 << (31- 12)) /* tag trap */                \
  | (1 << (31- 13)) /* user tag trap */          \
```

Listing 10: Trap function and privilege level definitions.

```
void __attribute__((noreturn)) tag_trap()
{
  panic("machine_mode: tag_trap triggered [%d] @ %p",
           read_csr(mcause), read_csr(mepc));
}

void __attribute__((noreturn)) user_tag_trap()
{
  panic("machine_mode: user_tag_trap triggered [%d] @ %p",
           read_csr(mcause), read_csr(mepc));
}
```

Listing 11: Trap handling functions within BBL.

```
static int setup_rt_frame(struct ksignal *ksig,
                          sigset_t *set,
                          struct pt_regs *regs)
{
  struct rt_sigframe __user *frame;
  long err = 0;

  ...

  /* Set up to return from userspace. */
  regs->ra = (unsigned long)VDSO_SYMBOL(
                current->mm->context.vdso, rt_sigreturn);
  //Manually validate the return address
  asm volatile ("stag %0, 0(%1)" ::"r"(2),
                 "r"(&(regs->ra)));
  ...
}
```

Listing 12: The signal return routine.

```
int copy_thread(unsigned long clone_flags, unsigned long usp,
  unsigned long arg, struct task_struct *p)
{
  struct pt_regs *childregs = task_pt_regs(p);

  /* p->thread holds context to be restored by __switch_to() */
  if (unlikely(p->flags & PF_KTHREAD)) {
    /* Kernel thread */
    const register unsigned long gp __asm__ ("gp");
    memset(childregs, 0, sizeof(struct pt_regs));
    childregs->gp = gp;
    childregs->sstatus = SR_PS | SR_PIE; /* Supervisor, irqs on */

    p->thread.ra = (unsigned long)ret_from_kernel_thread;
    asm volatile ("stag␣%0,␣0(%1)" ::"r"(2), "r"(&(p->thread.ra)));
    p->thread.s[0] = usp; /* fn */
    p->thread.s[1] = arg;
  } else {
    *childregs = *(current_pt_regs());
    if (usp) /* User fork */
      childregs->sp = usp;
    if (clone_flags & CLONE_SETTLS)
      childregs->tp = childregs->a5;
    childregs->a0 = 0; /* Return value of fork() */
    p->thread.ra = (unsigned long)ret_from_fork;
    asm volatile ("stag␣%0,␣0(%1)" ::"r"(2), "r"(&(p->thread.ra)));
  }
  p->thread.sp = (unsigned long)childregs; /* kernel sp */
  return 0;
}
```

Listing 13: The copy thread function.

both cases of the if-statement, the value 0x02 is written to the address of *p→thread.ra*, using the instruction *stag*.

The assembly code, located in *entry.S*, also includes multiple manual overwrites of the the return address register. The code sections are handling the return after an exception. The assembly routines *handle_exception*, *work_pending*, *ret_from_fork* and *ret_from_kernel_thread* set the value of the register ra to the beginning of the routine *ret_from_exception*. When the instruction *la ra, ret_from_exception* is performed, the return address tag is automatically set to zero and the next return instruction causes a tag trap. To fix this issue, the return address tag in the ra register is validated, after the value is modified. In order to do this in assembly, a global help variable is required, since the *stag* instruction can not directly modify tags of registers, but only tags, located within the memory. Therefore, the address to be returned to, is saved in the register ra, while the address of the help variable is stored in register s3. Then, register ra is stored on the address of the help variable. The value 0x02 is loaded in register ra, followed by a *stag* instruction, which stores this value as tag of the help variable. Afterwards, the data and tag of the help variable is loaded into register ra, which preserves the return address tag. All code segments, where these changes where necessary, are shown in Listing 14.

As already mentioned, the Linux kernel is able to load binaries from the SD card, in order to execute programs. But every time, data is fetched from the IO interface, the data is

```
.comm help_var, 8

.text
...
ENTRY(handle_exception)
  SAVE_ALL

   ...
1:   auipc gp, %pcrel_hi(_gp)
     addi gp, gp, %pcrel_lo(1b)

     la ra, ret_from_exception
     la s3, help_var
     sd ra, (s3)
     li ra, 2
     stag ra, (s3)
     ld ra, (s3)
   ...

work_pending:
  /* Enter slow path for supplementary processing */
  la ra, ret_from_exception
  la s3, help_var
  sd ra, (s3)
  li ra, 2
  stag ra, (s3)
  ld ra, (s3)
      ...
END(handle_exception)

ENTRY(ret_from_fork)
  la ra, ret_from_exception
  la s3, help_var
  sd ra, (s3)
  li ra, 2
  stag ra, (s3)
  ld ra, (s3)
     ...
ENDPROC(ret_from_fork)

ENTRY(ret_from_kernel_thread)
  call schedule_tail
  /* Call fn(arg) */
  la ra, ret_from_exception
  la s3, help_var
  sd ra, (s3)
  li ra, 2
  stag ra, (s3)
  ld ra, (s3)
    ...
ENDPROC(ret_from_kernel_thread)
```

Listing 14: Return routines in entry.S.

```
/*Only switch on invalid tag generation and
   leave the rest*/
#define invalidTagGenOn() ({\
 uint64_t tag_ctrl_state = read_csr(ptagctrl);\
 tag_ctrl_state |= INV_TAG_GEN;\
 write_csr(ptagctrl,tag_ctrl_state);\
}) \

/*Only switch off invalid tag generation and
leave the rest*/
#define invalidTagGenOff() ({\
 uint64_t tag_ctrl_state = read_csr(ptagctrl);\
 tag_ctrl_state &= ~ INV_TAG_GEN;\
 write_csr(ptagctrl,tag_ctrl_state);\
}) \
```

Listing 15: Helper functions to enable and disable the invalid tag generation.

tagged as invalid. In order to not to break binaries, all cases, where binaries are loaded by the kernel, have to switch off the invalid tag generation. This is done, by modifying the value of the tag control register. When a new binary shall be loaded, the process is prepared within the exec function of the kernel. It reads only the ELF header, in order to prepare the memory. This is done, using the exec kernel function within the file *fs/exec.c* (Listing 16). When the function is entered and before any data is read from a peripheral device, the invalid tag generation is switched off. Whenever the function is returning, the generation is turned on again, using the helper functions *invalidTagGenOn()* and *invalidTagGenOff()*. The helper functions are shown in Listing 15 and are located in the new kernel file *tag_ctrl.h*

However, this modification only ensures, that the first data and the header of the binary is read without invalid tags. Further data of the binary is read on demand, within the page fault handler. This implies, that when the page fault handler is entered, the invalid tag generation needs to be turned off. When the page fault handler returns, the generation is turned on again. We disabled the invalid tag generation for all page faults for the sake of simplicity. Not all page faults read data from IO, which is needed for binary files. Finer granular enabling/disabling for the different page fault types is a possible topic for further work. This modification is applied in the function *do_page_fault*, in the file *arch/riscv/mm/fault.c*. The modifications are shown in Listing 17.

Note, that the modifications for switching on and off the invalid tag generation within the kernel, are not thread safe. The problem is, that the tag control register is currently the same for the whole processor, processes and threads. Thus, it could happen, that one process executes a exec system call in order to load a new program, while another is reading some data from peripherals. In this case, during the time, that one thread switched off the invalid tag generation, the other thread can load data from the IO without retrieving invalid tagged data. We propose a solution for this issue in Section 8.

```
/*
 * sys_execve() executes a new program.
 */
static int do_execve_common(struct filename *filename,
        struct user_arg_ptr argv,
        struct user_arg_ptr envp)
{
  struct linux_binprm *bprm;
  struct file *file;
  struct files_struct *displaced;
  int retval;

  invalidTagGenOff();


  if (IS_ERR(filename))
  {
    invalidTagGenOn();
    return PTR_ERR(filename);
  }

  ...

  /* execve succeeded */
  current->fs->in_exec = 0;
  current->in_execve = 0;
  acct_update_integrals(current);
  task_numa_free(current);
  free_bprm(bprm);
  putname(filename);
  if (displaced)
    put_files_struct(displaced);

  invalidTagGenOn();
  return retval;

  ...

out_ret:
  putname(filename);
  invalidTagGenOn();
  return retval;
}
```

Listing 16: The exec routine.

```
/*
 * This routine handles page faults.  It determines the address and
     the
 * problem, and then passes it off to one of the appropriate routines.
 */
asmlinkage void do_page_fault(struct pt_regs *regs)
{
   ...
  invalidTagGenOff();

  fault = handle_mm_fault(mm, vma, addr, flags);

  invalidTagGenOn();
      ...
}
```

Listing 17: The page fault handler entry.

Chapter 8

Results and Future Work

The previous Chapters covered the elaboration of the theory and the implementation of countermeasures against code reuse attacks, based on a tagged architecture. During the research, we discovered some pitfalls and possible vulnerabilities of our policies in the current development state. As a result, the tagged security policies and its corresponding architecture are in a prototype stage. However, as we evaluated the system, it is safe to say, that the policies work as expected and protect the machine from code-reuse attacks in the common cases. Also, the performance and complexity overhead is minimal. Within this chapter, we evaluate the outcome of this thesis and propose possible future work, in order to improve the overall system even further.

In order to test the effectiveness of our tagged policies, we designed test cases, which represent initiations of a code-reuse attacks. The full attack is not executed, since the focus of this theses is to prevent the mounting of those attacks. The test cases were deployed as stand alone applications in the bare metal mode, as well as within the Linux kernel. In order to test the return address protection, we implemented a test case, which fits the example attacks in Section 5.3.2. This means, that we first overwrite the full return address. The trap is correctly performed, as soon the function, which performed the overwrite executes the return instruction. The other two scenarios are similar, but the return address is once modified in a partial manner, while the other case performs a partial copy of the return address. Note, that the test program executes the overwrite by itself, without external IO input of the user. This proves, that this policy is very strong, since not even the program itself can modify the return addresses. The test case for jump-oriented programming is following the example attack code, presented in Section 5.4.2. If executed within Linux, multiple input variants, which shall overwrite the function pointer on the heap, are tested. It is possible to use the *scanf* function, as proposed in the example attack. Also, the string can be passed as argument to the test case, when executed from the Linux shell. The third variant reads the string from a file. This even allows to inject an exact address of a gadget function, located within the test case. In our evaluation, all cases correctly marked the function pointer as invalid, since all input types are associated with peripheral devices. The trap is triggered, as soon the function pointer is used for the function call. When the function pointer protection is switched off, the attack function is executed when the correct address was injected. This shows, that the function pointer protection is correctly enforced within the Linux kernel, while still allowing standard operation of the kernel.

While the Linux kernel is stable and does not cause false traps, when executing user applications and the shell, there are some cases when false traps could occur anyway. We discovered, that this false traps only occur, if the function pointer protection policy is activated. The problem is caused by the library function *memset*. This function is implemented as a assembly routine within the RISC-V toolchain. It causes an interesting behaviour, which violates general control flow rules. The input parameters of *memset* are the start address of the memory, the character, which shall be written to the memory range, and the length of the region to overwrite. The code includes multiple entry points, which are used for unaligned address processing. Depending on the alignment of the address, a different code section is executed. The problem with this approach is, that the address of the target is calculated, by using the start address parameter. As a consequence, a input parameter of the function is used to calculate jump addresses. The false traps occur, when the register with this jump target address is tagged as invalid. This happens, if the start address parameter was calculated, by reading a file header. In this case, the start address is tagged as invalid, which propagates until the indirect jump instruction in the *memset* function. In order to solve this issue, all library functions, implementing this kind of behaviour, have to be re-written in the future. At the moment this modification is not performed.

We also performed a size comparison of the untethered lowRISC with our tagged memory implementation. We developed the tagged architecture on the Kintex-7 KC705 evaluation board, featuring the XC7K325T-2FFG900C FPGA [12]. The cache sizes for both designs and are listed in Table 11. The increase of the cache size is resulting from the additional tag bits, stored in every cache line.

Cache	Sets	Ways	lowRISC size (Bytes)	Tagged security size (Bytes)
L1 Data	32	2	4096	4352
L1 Instruction	32	2	4096	4096
L2	32	2	4096	4352
Tag Cache	32	2	-	4352

Table 11: Cache size comparison of the lowRISC and tagged security implementation.

Table 12 shows the utilization for each component of the FPGA. The values where extracted from the post-implementation evaluation of Vivado. The biggest differences occur for the Flip Flops (FF) and Lookup Tables (LUT). The FF utilization is increased by 9.45% and the LUT utilization by 7.22% in our implementation. This results from the added control logic and the tag cache, as well as the increased cache size for tags. This results match our expectations and show, that tagged memory and tagged security policies do not significantly increase the chip complexity and size.

Type	Total available	lowRISC utilization	tagged security utilization	Difference
FF	407600	31197 (7.65%)	69708 (17.10 %)	9.45%
LUT	203800	54469 (26.73%)	69200 (33.95 %)	7.22%
Memory LUT	64000	4324 (6.76%)	4345 (6.79 %)	0.03%
I/O	500	123 (24.60%)	123 (24.60 %)	0%
BRAM	445	30.5 (6.85%)	33.5 (7.53 %)	0.68%
DSP48	840	24 (2.86%)	24 (2.86 %)	0%
BUFG	32	5 (15.62%)	5 (15.62 %)	0%
MMCM	10	2 (20.00%)	2 (20.00 %)	0%
PLL	10	1 (10.00%)	1 (10.00 %)	0%

Table 12: Utilization of the different FPGA elements for both the original lowRISC and the tagged security implementation.

As already mentioned in Section 7.4, the current mechanism for switching on and off the invalid tag generation within the Linux kernel is not thread safe. This results from the global tag control PCR register, which is not tied to a specific thread or core. We suggest an advanced implementation of the tag control register for future work. The problem, that all threads have the same state of the invalid tag generation can be solved, by implementing a context switch mechanism for this register. The state of the tag control register needs to be saved in an additional register on a context switch in the same manner, the stack pointer is preserved. When the context is restored, the state can be retrieved. The other thread can therefore have its own state of the tag control register. This modification implies the necessity to switch the tag control register from the PCR in the CSR register space. This will also allow multi-core architectures, since every core would have its own tag control registers.

Chapter 9

Conclusion

In this thesis, we proposed a new hardware-based countermeasure against code-reuse attacks. We used two tag bits in order to distinguish trusted and untrusted return addresses, as well as function pointers. By implementing tagged security policies on the lowRISC SoC, a prototype architecture was developed. We developed and tested the system within an FPGA based design flow. The prototype is fully capable of executing Linux and user applications, while the policies are active and the system is protected. Furthermore, we accomplished to design the architecture and the policies in a way, that no compiler support is necessary. Previous research failed on the automatic protection of function pointers without software support.

The results of this thesis could affect future processing systems, by implementing a much stronger protection against state of the art attacks. Tagged architectures, like the proposed one, are currently experiencing a resurrection. With our contribution to the topic, code-reuse attacks can be prevented from happening in future general purpose hardware.

We conclude, that although the current outcome is still in a prototype stage, it can be seen, how tagged memory can improve the security of processing systems. The goal to defend return-oriented and jump-oriented programming attacks was reached and we proposed future work, in order to exit the prototype stage and apply the tagged security policies in final products.

References

1. Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume I: User-level ISA, version 2.1. `https://riscv.org/specifications/`, 2014. [Online; accessed September 2016].
2. Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual volume II: Privileged architecture version 1.9. `https://riscv.org/specifications/`, 2015. [Online; accessed September 2016].
3. Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. `https://chisel.eecs.berkeley.edu/chisel-dac2012.pdf`, 2012. [Online; accessed September 2016].
4. Burroughs Corporation. The Descriptor - a definition of the B 5000 Information Processing System, 1961.
5. Forrest S., Somayaji A., Ackley, D. Building diverse computer systems. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), 1997.
6. Liu L., Han J., Gao D., Jing J, Zha D. Launching return-oriented programming attacks against randomized relocatable executables. In: Proceedings of the 10th International Conference on Trust, Security and Privacy in Computing and Communications, 2011.
7. G. Edward Suh, Jaewook Lee, Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking, 2004. [Online; accessed September 2016].
8. Edward A. Feustel. The Rice research computer: a tagged architecture, 1972.
9. Edward A. Feustel. On the advantages of tagged architecture, 1973.
10. Julián Armando González. Taxi: Defeating Code Reuse Attacks with Tagged Memory. `http://people.csail.mit.edu/hes/ROP/Publications/Julian-thesis.pdf`, 2014. [Online; accessed September 2016].
11. AVNET Inc. Zedboard developement kit. `http://zedboard.org/product/zedboard`, 2016. [Online; accessed September 2016].
12. Xilinx Inc. Kintex-7 KC705 Evaluation Board. `http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html#overview`, 2016. [Online; accessed September 2016].
13. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack, 2011.
14. lowRISC. A fully open-sourced, Linux-capable, RISC-V-based SoC. `http://www.lowrisc.org`, 2016. [Online; accessed September 2016].
15. ARM Ltd. AMBA AXU and ACE Protocol Specification. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html`, 2016. [Online; accessed September 2016].
16. Microsoft. Data execution prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006. [Online; accessed September 2016].
17. W. S. A. Bradbury, R. Mullins. Towards general purpose tagged memory. `http://riscv.org/workshop-jun2015/riscv-tagged-mem-workshop-june2015.pdf`, 2015. [Online; accessed September 2016].
18. University of Berkley. TileLink 0.3.3 Coherence API Specification. `https://docs.google.com/document/d/1vBPgrlvuLmvCB33dVb1wr3xc9f8uOrNzZ9AFMGHeSkg/pub`, 2016. [Online; accessed September 2016].
19. University of Berkley. TileLink 0.3.3 Specification. `https://docs.google.com/document/d/1Iczcjigc-LUi8QmDPwnAu1kH4Rrt6Kqi1_EUaCrfrk8/pub`, 2016. [Online; accessed September 2016].
20. University of Berkley. Vscale Microarchitectural Implementation of RV32 ISA. `https://github.com/ucb-bar/vscale`, 2016. [Online; accessed September 2016].
21. University of Berkley. Z-scale Microarchitectural Implementation of RV32 ISA. `https://github.com/ucb-bar/zscale`, 2016. [Online; accessed September 2016].
22. Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, Yunheung Paek. HDFI: Hardware-Assisted Data-flow Isolation, 2014.

23. RISC-V Foundation. The RISC-V website. `https://riscv.org`, 2016. [Online; accessed September 2016].
24. Lucas Davi, Ahmad-Reza Sadeghi. *Building Secure Defenses Against Code-Reuse Attacks.* Springer, 2015.
25. Samuel Fingeret. Defeating Code Reuse Attacks with Minimal Tagged Architecture. `http://people.csail.mit.edu/hes/ROP/Publications/sam-thesis.pdf`, 2014. [Online; accessed September 2016].
26. Erik Buchanan, Ryan Roemer, Hovav Shacham, Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC, 2008.
27. Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), 2007.
28. Cristiano Giuffrida, Anton Kuijsten, Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization, 2012.
29. Team TESO. Exploiting Format String Vulnerabilities. `http://julianor.tripod.com/bc/formatstring-1.2.pdf`, 2001. [Online; accessed September 2016].
30. Martìn Abadi, Mihai Budiu, Ùlfar Erlingsson, Jay Ligatti. Control-Flow Integrity. `https://www.microsoft.com/en-us/research/wp-content/uploads/2005/11/ccs05-cfi.pdf`, 2005. [Online; accessed September 2016].
31. RICE University. RICE University Computer-Basic Machine Operation. `http://archive.computerhistory.org/resources/access/text/2015/09/102726213-05-01-acc.pdf`, 1962. [Online; accessed September 2016].
32. Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, Andrew Waterman. The Rocket Chip Generator, 2016.
33. Peter W. Shantz, R. A. German, J. G. Mitchell, R. S. K. Shirley, C. R. Zarnke. WATFOR—The University of Waterloo FORTRAN IV compiler, 1967.