POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Protecting Register Spilling in AArch64 Microprocessors

Author: Andrea Fanti

Supervisor: Paolo Ernesto Prinetto

Advisor: Carlos Chinea Perez

December, 2021

Abstract

Memory-related vulnerabilities can be a serious threat to modern software. Main memory is vulnerable in the way that it is not aware of its contents but instead it is designed following the von Neumann architecture. In this architecture, the CPU hands over commands to the memory, which executes them by retrieving or writing data from or to the specified location (address). Lack of content-awareness implies that sensitive data cannot be appropriately protected against tampering, unwanted eavesdroppers and the like.

Memory security has then to be implemented *upstream*: for instance, modern operating systems isolate running processes and forbid access of others' memory sectors. Unfortunately, this is not enough: some programming languages (notably C and C++) do not provide native memory protection features, and memory management and boundary checks are demanded to programmers, who are error-prone. This results in critical consequences.

Register spilling is a data management mechanism supported by modern compilers, whereby machine instructions are inserted in the compiled program to store temporary results in main memory (i.e., *spilled*), without the programmer being able to intervene. Register spilling occurs in two main occasions: when there are no more free registers to save a result generated by some calculation, and the result is needed later during execution of the compiled program, or when a just-called function uses one or more registers that, when respecting environmental constraints (i.e., ABI rules), have to be loaded into memory.

Register spilling represents a potential source of vulnerability, as registers spilled in main memory can be corrupted if the program contains a memory corruption vulnerability. A possible baseline mindset to solve this problem would be to treat memory like an untrusted peripheral where attackers have full control and proceed accordingly from there. This is precisely what is done in this thesis.

The present thesis describes experiments following this approach. In particular, the work focused on an implementation of a register spilling protector for the AArch64 backend of `llvm`, a framework used to create compilers, notably `clang`. The document presents how this feature works, alongside the analysis of its effectiveness and impact on performances if compared to similar countermeasures. The two different events that originate register spills are tackled separately, as in the latter some optimizations are possible.

Spillings can optionally be protected only against tampering (integrity protection) or also against passive attackers (confidentiality protection). The implementation uses Pointer Authentication instructions, introduced with ARMv8.3-A, that allows to calculate cryptographic authentication codes, not only for pointers, but also for generic data.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Memory corruption bugs represent a major legacy problem for computer software written in machine-oriented languages (such as C and C++), that still are present in modern applications running in a wide range of domains. These bugs originate mainly from the necessity of manually performing memory-management, e.g., directly handling memory pointers and memory allocation. Adopting memory-safe programming languages is not always possible: for instance, the software could be legacy and cannot be modified, or the performance loss would make this change unfeasible.

So far, history showed us that every time there seems to be a solution to the latest attacks in memory corruption, a new form of attack was born.

Initially, there was code injection in the stack, whereas an attacker could introduce directly executable code in the stack exploiting stack buffer overflows present in vulnerable programs [32]. To solve this issue, stack canaries [14] where introduced, and they still are in use today, almost 18 years after their introduction in compilers [40] due to their simple strategy of operation. Stack canaries implementation introduce a random value between at the bottom of the stack frame of a function that is checked against a reference value upon exit to detect memory corruption. Unfortunately, the reference value can be revealed if the program suffer from other memory vulnerabilities or if the reference value itself can be overwritten [7].

Code injection was ultimately defeated with the introduction of W$\oplus$X memory, that mandates each portion of memory to never be writable *and* executable at the same time. Unfortunately, code reuse attacks (CRAs), which do not insert any new code into memory but exploit already present code, have been discovered.

Return-Oriented Programming [36] reuse short code snippets ending in return statements to build macroinstructions called *gadgets* which can be used to create Turing-complete attacks, having the same outcome of directly executing attacker-provided instructions, all of this just by corrupting a vulnerable return address. To defeat ROP, Control-Flow Integrity [1] was and is still centered in research. CFI enforces checks during all control flow changes, to allow only predetermined ones based on the Control-Flow Graph.

Another CRA which is immune to CFI is Data-Oriented Programming [18]. In this attack, exploiters corrupt vulnerable variables that indirectly influence the control flow of the program (ex. condition variables) so that the program will decide *by itself* to take

certain execution paths. By tweaking vulnerable variables at the right places in the right way it is possible to create Turing-complete attacks.

Research is thus now focused on protecting systems against ROP and DOP.

In short, memory corruption vulnerabilities are dangerous. One safe space to store data, which cannot be corrupted by attackers, is the registers inside the processor.

Unfortunately, registers in a CPU are limited. The component that decides which data is stored on the CPU registers and which on main memory is the compiler, that translates source code into machine code.

Concerning the stack, the compiler decides to save and reload registers to it with a process named *register spilling*.

Register spillings are often overlooked security-wise, but what is happening is that some data is stored in memory without the programmer even knowing it.

Register spillings are necessary as the program could need to elaborate more data than what can be stored on the CPU at once, or simply because the processor manufacturer mandates that some registers have to be spilled into memory when different functions in the program interact [27]. In other words, these registers contain intermediate results which the compiler decides to store on the stack to make accommodation for other values that are needed more in the short-term.

AArch64 is a RISC architecture developed by ARM. As many RISC architectures do, AArch64 stores the return address of a function (i.e., the address of the instruction to execute once the function finishes execution) in a specific register: the *Link Register* (`LR`). This would exempt RISC architectures from stack-smashing attacks that target this value, but with a limited impact. In fact, to prevent overwrites, `LR` has still to be spilled on *non-leaf functions* (i.e., functions that will call other functions).

Being able to change spilled values prior to their reload in the corresponding register allows an attacker not only to perform all attacks where the return address of a function is modified (such as *code injection* [7] or Return-Oriented Programming [36]), but also to potentially control the execution flow of the program, if one of the registers is later used as part of a condition.

To aid programmers in developing more secure software, ARM introduced new instructions in its processors that permits the creation of integrity checks to verify if values are unexpectedly modified.

To protect programs against the threats of register spillings it is necessary to work at the compiler level in order to insert instructions as the executable binary is generated. Implementing security directly in the compiler has also the advantage of creating a form of protection that can be applied also to legacy software without having to modify the program, or perhaps to delay the need of intervention of programmers to rush in releasing a bugfix for a memory vulnerability that can potentially do more harm than good.

In this thesis, we present the creation of a *register spill protector* and its integration into the LLVM compiler suite. The register spill protector is capable of dynamically inserting instructions alongside the ones handling spills in order to prevent attackers to modify their values and optionally also obfuscate them with encryption.

The spill protector adds a minimum overhead on the executable size, composing just 7% of the final binary.

1.1 Outline

The remainder of the paper is organized as follows: in Chapter 2 we explain the motivation behind this thesis and all the notions needed to understand fully our solution to the problem of register spilling. In Chapter 3 we show what is the current state of the art on similar memory-related problems, with a focus on Pointer Authentication. In Chapter 4 we present our solution to the problem of register spilling based on LLVM. In Chapter 5 we wrap up what has been done in this thesis and present some possible future work.

Chapter 2

Background

Microprocessors represent the brain of a computer and are the units tasked with executing instructions. However, microprocessors only understand instructions in machine code, which is difficult for humans to write directly. Thus, programming languages were born: programmers write *source code* files in a programming language which are then compiled into machine code by a *compiler*, which is a program itself.

The process of compilation is not monolithic, but passes through many steps: in each of these, instructions get closer and closer to machine code and in the process get optimized by the compiler.

Register spilling is a mechanism used by compilers to alleviate *register pressure*. This is a technical term used in compiler design that indicates the availability of not-in-use registers. At any point during the execution of a program there is a certain amount of registers storing values that will be useful in the future during the execution of a program. When a register is holding a value that will not be used in the future, it is considered "available" by the compiler during the **register allocation** phase and hence the result of a calculation can be stored into it without overwriting any important data.

Having *high register pressure* means that the compiler has the need of storing much information in registers at the same time, while *low register pressure* means the opposite.

Every microprocessor has a limited amount of registers, and usually some of them have a special purpose and cannot be used to store calculation result or be used as operands. The `AArch64` ARM architecture (Section 2.2), as an example, contains 32 64-bit general-purpose registers that, when following the AAPCS (Section 2.2.1) (i.e., a set of rules that standardizes how functions should interact between them), they become even less. Another popular architecture, `amd64`, has even fewer registers: 16 [2].

The amount of registers in a microprocessor obviously imposes a limit on the maximum amount of values that can be stored on-chip at the same time. It may happen, however, that at a certain point during execution there are more values, needed for the execution of the program, than the amount that can be stored in registers. The compiler then has to find a place where to store these excess values. The choice often goes to the *stack*, which resides in a dedicate portion of memory.

Spilled registers are not obliged to reside on the stack, but the stack is a good place to store values local to a functions. The process of storing a register onto the stack is referred to as *spilling the register*, while loading back the register from memory to the chip

is *reloading the register*.

2.1 Motivation

Register spilling are seldom considered security-wise: the majority of work regarding spill protection is concentrated on protecting the function return address integrity rather than register spills as a whole. Moreover, programmers do not typically take into account (nor are they informed) of which variables in their code ends up in memory and when.

Google Chrome is the default web browser in the Android mobile OS. Android is, in turn, the most used mobile OS in the world [17]. Android phones often equip ARM processors due to their characteristic low power consumption that translates to a higher battery duration.

A web browser is a critical piece of software security-wise as it works with data originated from the internet (which is publicly accessible) and at the same time elaborates private information (like emails and passwords).

Due to these reasons in this thesis we investigated how often register spilling happens in an application where this could compromise security, Chromium[4] (the open-source program that serves as basis for Google Chrome).

To do this we slightly modified LLVM 12 in order to be able to count how many spills are generated per file and of which type.

In Table 2.1 one can see the overall statistics collected by compiling the chromium web browser.

statistic	*value*
nr. of functions	1041705
saved CSRs	3548372
spills inserted	133963
spill slots allocated	112163
reloads inserted	240375
emergency spilled registers	1

Table 2.1: Overall statistics of compiled code of chromium

From this data one can clearly see that the number of potential vulnerabilities is so high that among those it is almost certain that at least one is exploitable in some way (there is a very large attack surface).

An attacker will find on average 1 spill every 10 functions analyzed.

With more changes to the compiler and having a Chromium build with debug symbols (that retains more information about the original source code) a skilled attacker could extract more precise information as to where these spills are in memory and also, for example, in which function they reside.

2.2 AArch64 architecture

The AArch64 architecture is the 64-bit architecture developed by ARM. AArch64 is a *Reduced Instruction Set Computer* architecture (the original acronym of ARM was in itself "Advanced RISC Machine").

RISC architectures, also known as *load-store*, usually share some common design choices regarding registers and access to memory.

An AArch64 processor has to interact with the memory through the use of registers, i.e., it is not possible to operate directly on memory. The only instructions that can operate on memory are, at least in theory, *load* and *store* that transfer registers from/to memory. This drawback is counteracted with the presence of an high number of registers (32). Another property of AArch64 processors is that the instructions have a constant size.

2.2.1 Procedure Call Standard

The Procedure Call Standard for AArch64 [27] (AAPCS) defines a common set of rules to facilitate the interoperability between different binaries.

In particular, it illustrate the purpose for each register inside a function and during a call to one.

Integer registers in AArch64 can be referred as to with the names `X0` to `X31`, with some registers having aliases. When executing AArch32 instructions registers can be called `W0` to `W31`. The 32 bits registers overlap with the lower half of their 64 bits counterparts.

Let us see now broadly the classification of AArch64 when performing a function call:

- `X0-X7`: *arguments registers* and return value. These registers must contain the arguments to the function and, at the end of its execution, the return value (if the value is larger than 64 bits it will span over registers following `X0`.
- `X8`: *indirect result location register*. This register stores a pointer to the return value, in case it cannot be stored inside the first 8 registers (for example, when a large structure is returned).
- `X9-X15`: *temporary registers* or *scratch registers*. These register can be used inside the function without having to perform any action. Note that this also means that if a function needs values contained within these registers they have to be spilled prior calling a function otherwise the called function might overwrite them.
- `X16-X17`: aliased `IP0`, `IP1`, they are not relevant for the purpose of this thesis.
- `X18`: the use of this register depend on the platform the processor is used on. In other words, ARM did not mandate any particular use for this register.
- `X19-X28`: *Callee-Saved Registers* (CSRs). These registers can be used inside a function but to the caller function they have to appear as not changing. In order to do this, the called function has to save spill them on the stack and reload them before returning to the caller function.
- `X29`: aliased `FP`, the *Frame Pointer* register is used to refer to a portion of the stack of a function when this allocated numerous bytes on the stack and the code cannot refer

to all objects contained within it using `SP` + a constant contained in the instruction itself (ranging from values -256 to 32760)

- `X30`: aliased `LR`, the *Link Register* contains the address of the instruction to execute once the function returns. This register is automatically set by function-calling instructions `BL` and `BLX` (Branch and Link, Branch and Link X register).
- `X31`: aliased `SP`, the *Stack Pointer* is the address of the stack head. Most of the instructions cannot refer to `SP` (i.e. cannot modify it) directly. These instructions use instead `X31` to refer to the *Zero Register* `XZR` which is a special register that once read always returns 0 and that discards writes.

  The Zero Register is commonly found on RISC architectures and permits the simplification of design to the architecture itself: for instance, the commonly found instruction `MOV` X_{dest}`,` X_{source}, used to copy a register into another, is actually an alias for the `ORR` X_{dest}`, XZR,` X_{source} instruction which performs an OR between the source register and the Zero Register, obviously always obtaining X_{source}.

2.3 Memory corruption

Memory corruption is an legacy problem for software security that is yet to be completely eradicated. The problem is particularly present in those programming languages where memory management control is handled by the programmer (notably C and C++).

Memory corruption is also present in most modern software: it is sufficient to have a quick search for "buffer overflow" in the *Common Vulnerabilities and Exposures* (CVE)[1] records (which is a list of software exploits managed by MITRE Corporation[2]). MITRE reports buffer overflows (specifically, out-of-bounds writes) to be the most dangerous vulnerability in 2021[31] by taking into account the frequency of which it was reported and the security impact it had.

The presence of memory corruption is caused by modern processors (and computers in general) designed to following the von Neumann scheme, where memory is not aware of the contents it is being used to store. In the von Neumann architecture, memory operates independently from the microprocessor, but follows simple orders from the microprocessor (namely to read data at a specific address or to write specific data at a specific location). If the memory somehow contained the knowledge of which subset of it is in use, or which areas store "important" information, memory corruption would be a much lesser problem.

2.3.1 Buffer Overflow

Buffer overflow vulnerabilities are the most common memory corruption vulnerability in modern software.

[1] https://cve.mitre.org/cve/search_cve_list.html

[2] https://www.mitre.org/

A buffer overflow happens when an array is accessed outside of its boundaries, something that can easily happen when developing programs in C and C++ which do not check for an array length before operating on it (more precisely, the size of each array is not computed and/or stored anywhere automatically by these languages). The goal of this kind of attack is to corrupt memory outside of the array, e.g., to inject executable code or anyway corrupting sensitive data.

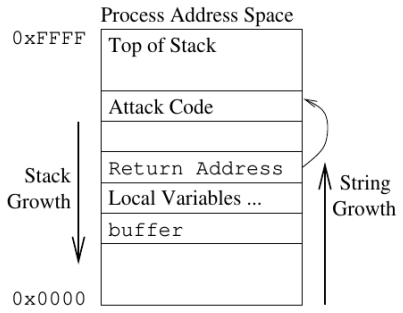

Figure 2.1: Memory configuration of a stack-smashing attack [14].

Executing custom code by exploiting a stack-smashing attack can be done for example by overwriting the return address (i.e., the address of the instruction to execute once the function has to return), which is stored at the bottom of the *function frame* in memory. This technique is visually illustrated in Figure 2.1

The function frame is a portion of the stack containing all the memory local to the execution instance of the function. It is stored on the stack, as the stack organization of Last-In First-Out (LIFO) fits for allocating such a space: when a function is called its frame is allocated on the stack, on top of the frame of the calling function, and is used while the function is executing. Once execution ends, this space gets freed (i.e. the stack pointer is moved down).

As is possible to see from Figure 2.2, if a fixed-size array is present on the function, assuming no floating point or SVE registers are used, by writing to an index larger than the length of the array, it is possible to overwrite whatever is present on the memory above the array location (above meaning with larger addresses, as addresses grow in the arrow direction).

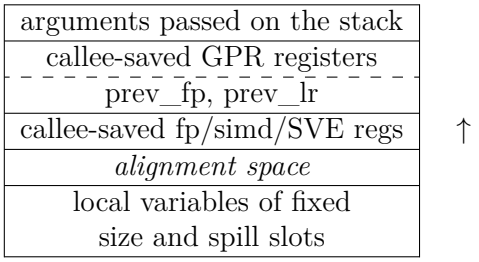

Figure 2.2: Frame structure under Linux on AArch64

Another trick used by attackers to increase the odds in their favor when injecting code is to prepend it with a series of `NOP` instructions (that leave the machine state unaltered). When this *NOP sled* is used, once the return address is modified it is not anymore necessary to jump exactly in the injected code but any of the `NOP` instructions will suffice: the processor will automatically retrieve and execute all `NOP`s between the current landing side and the code.

Possible attacks against an unprotected stack are detailed at [32]. Most of these attacks have been defeated by the introduction of W⊕X memory protection feature [30] in all major operating systems: according to this policy, a memory page cannot be both executable and writable at the same time. However, there are attack techniques that easily thwart such a kind of defense mechanisms, such as *Return-Oriented Programming* (ROP, Section 2.3.2).

2.3.2 Return-Oriented Programming

The impossibility of executing code in a writable memory page makes attacks based on code injection significantly more difficult new new attack strategies allows the exploiter to *reuse code* already present in the program memory image.

Return-Oriented Programming, or shortly referred to as ROP, is an attack technique in which code snippets ending in a return instruction (called *gadgets*) are exploited and used collectively to create a group of "macro-instructions" that, together, have Turing-complete computing capability [36] (i.e., they can perform any kind of task the machine is capable of). The attack is carried out by concatenating the execution of different gadgets to execute a malicious task.

The gadget catalogue is constructed by analyzing exit points of functions in the program. From the last instruction of the function (i.e., usually a POP restoring various registers), immediately-preceding instructions are added one by one until a useful sequence is found or the gadget is discarded. This attack leverages on the fact that by design with a jump instruction it is possible to redirect execution to any instruction of the process (and in particular, to a gadget entry point).

Attackers do not have to inject code because at the end of execution of a function the return address is retrieved from the stack and the processor will resume execution at the instruction pointer by that address. If the return address points to the beginning of a gadget the instructions it contains are executed, followed by another return instruction, which will again retrieve another return address from the stack (that the attacker possibly again corrupted).

An attacker does not even have to find the gadgets inside a program, but can take advantage of some library, rarely unused by programs, in which they have already been located (for example, the standard C library `libc`).

A simple program developed following the ROP ideology can be seen in Figure 2.4. Here a sequence of (ROP style) NOPs are executed. Recall that in the original stack smashing attack (Section 2.3.1) a NOP sled can be used to increase the odds in favor to the attacker by permitting a less precise jump prior to the injected code. With ROP, the same technique can be leveraged using gadgets in place of traditional instructions.

Once the return address is popped from the stack the first NOP gadget will be executed.

The general structure of a ROP program can be seen in Figure 2.3.

Given that ARM has fixed-size instructions (and other differences) from the `i686` architecture used in the initial paper by Roemer et al., the technique has to be slightly adapted in order to work, but the strategy used in an attack remains the same [10, 19].

2.3.3 Control-Flow Integrity

In a program, *control flow* refers to its execution sequence, and particularly to how this moves through the various interconnected *basic blocks* (see Section 2.5.2) making up the *Control-Flow Graph* (CFG).

Control-Flow Integrity (CFI) refers to the enforcement of respecting the CFG in all control-flow transfers (i.e. jumps, calls, and returns) [1].

CFI is often enforced in one of two ways:

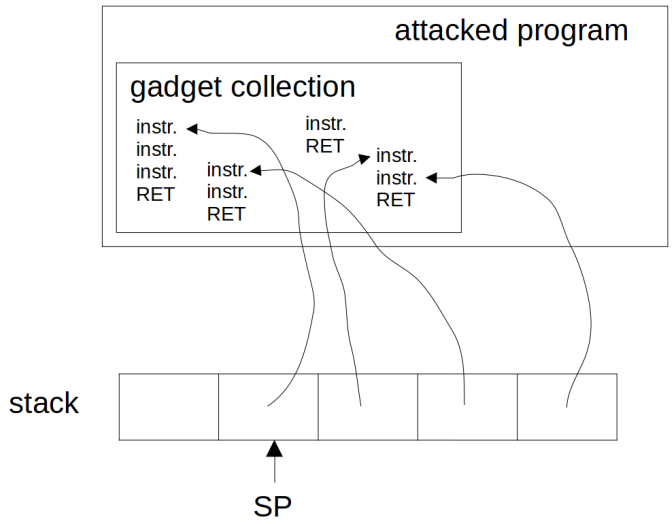

Figure 2.3: Execution of a ROP program. Note how some gadgets may contain other gadgets

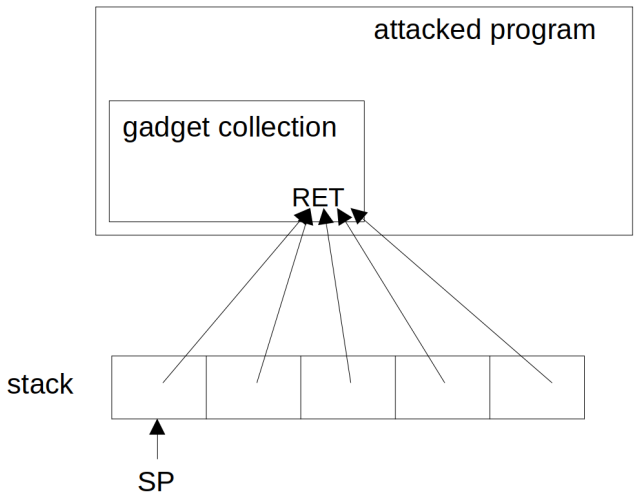

Figure 2.4: NOP sled structure in ROP.

- *Forward edge* control: from a basic block (or function), execution can only transfer to some specific other basic blocks or functions (i.e., the possible successors);
- *Backward edge* control: once a function returns, execution must resume from where the caller invoked the function;

In the original paper, Abadi et al. propose a technique on how to do such enforcement centered around modifying the binary executable. The idea is that instead of directly tracking each basic block with their addresses, each one is assigned a label (an ID), and jump instructions are modified so that at the calling site the target label is specified and upon jump completion the label is checked with the one stored at the beginning of the basic block. This is done in practice by virtually introducing new instructions to call and return to/from a function and to to perform conditional jumps that not only specify a target address, but also a target label.

Not all jumps in a function have to be converted to their "advanced" form, but only those ones that are vulnerable to tampering: for most jumps the target address is present already in the instruction itself and thus cannot be modified by attackers when W⊕X memory protections are active.

It is possible to see how this can severely reduce the possibility of a ROP attack being successfully performed: if the CFG is enforced, an attacker would need to search for gadgets among the possible targets and cannot jump anymore at any instructions inside the function (in fact, if the program is well structured, or in other words without `GOTO`s, the basic blocks available to the attackers will only be located at the beginning of targeted functions).

A problem with this implementation is when a function is called from many sites and/or an instruction can move the execution flow to a multitude of basic blocks. In these cases, if the call instruction only supports specifying a single target label, all the different targets need to have the same label. Similarly, all basic blocks that call a frequently used function need to have the same label. One way to reduce this scenario from happening, suggested by Abadi et al., is to *duplicate code* or insert support for *multiple labels* in the instructions.

The major difficulty in enforcing CFI is computing the CFG, which is computationally difficult to generate prior to the execution of a binary. This is why initial implementations of CFI have been using a coarse CFG, that is less precise and easier to compute and manage as it groups similar basic blocks together by assigning them the same label.

Other techniques that were initially thought as valid mitigation include recognize ROP attacks by noticing if gadget-like code snippets are executed in fast succession [13], or limit "return" instruction to only land on an instruction following a "call" instruction [33, 9, 41].

Unfortunately it has been shown that CFI cannot be securely realized when using a coarse grained CFG or other heuristic basic approaches. [11]. For this reason, fine-grained CFI is what is being studied nowadays.

2.3.4 Data-Oriented Programming

Contrarily to ROP where the attacker needs to be able to jump anywhere in the code, Data-Oriented Programming [18] (DOP) leverages use of the attacked code without breaking Control-Flow Integrity (CFI).

The idea behind this kind of attack is to corrupt variables which are critical in deciding the execution flow of the program. This idea is not new but the strategy in how these

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

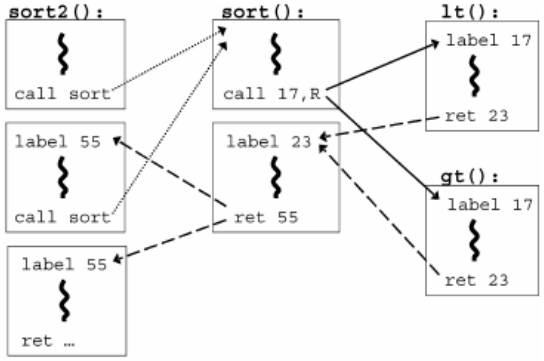

Figure 2.5: Layout of a program section and its corresponding CFG. [1]

vulnerable variable are exploited is: with DOP, just like in ROP, it is possible to trigger the execution of *gadgets* that together render the attack Turing-complete.

The attacker generates different gadgets by corrupting vulnerable variables that control the flow of execution within the program.

To pursue their target, Hu et al. defined a minimal language (MinDOP) composed of only 6 instructions that has been shown to be Turing-complete. This language can be seen in Figure 2.6.

Semantics	**Instructions in C**	**Data-Oriented Gadgets in DOP**
arithmetic / logical	`a op b`	`*p op *q`
assignment	`a = b`	`*p = *q`
load	`a = *b`	`*p = **q`
store	`*a = b`	`**p = *q`
jump	`goto L`	`vpc = &input`
conditional jump	`if a goto L`	`vpc = &input if *p`
p – &a; q – &b; op – arithmetic / logical operation		

Figure 2.6: MinDOP instructions. [18]

It is thus sufficient to find gadgets that implement these instructions and trigger their execution.

To execute a sequence of gadgets it is sufficient to find a *dispatcher* inside the codebase. A dispatcher is a piece of code that can be manipulated by the attacker to trigger the execution of a multitude of gadgets. One such piece of code could be for example a loop containing a selector statement in itself of which the loop control variable can be corrupted.

2.4 ARM Pointer Authentication

In 2016, ARM announced its version v8.3-A of the AArch64 Instruction Set Architecture (ISA) [25]. Among other enhancements, the *Pointer Authentication* (PA) extension has been added, which mandates its implementation in all processors [24].

The PA extension introduces a variety of instructions in the AArch64 instruction set, aiming to help compiler developers to solve or alleviate memory corruption vulnerabilities.

The idea behind pointer authentication is that the 64-bit physical address space is not used completely in most (if not all) of the AArch64 deployments, simply because the amount of bytes addressable with 64 bits is much larger than what the current applications require. This observation translates into the fact that all 64 bits are not needed to represent and address and can therefore be repurposed. When PA is employed, these bits are used to store a short cryptographic MAC (e.g. from 3 to 31 bits long on linux, but usually 16, depending on configuration[35]), to be used for notifying if the address has been manipulated.

Instructions to calculate MACs of generic data (i.e., not pointers) are also introduced, and these are the ones that are of an interest in this thesis.

Different secret keys are used by PA instructions depending on the nature of data (e.g., generic data, instruction pointers or data pointers) and the programmer choice (e.g., some instruction can use either of two keys , "A" or "B"). These keys are configured to be unreadable when the processor is running in user mode (Exception Level 0, EL0) [35], so only privileged processes (like the Linux kernel) can modify them while running in higher EL.

On Linux, new keys are generated for processes when their corresponding `exec()` system call is executed [37].

When the MAC is embedded into an address, there are no instructions to check it directly, but there exists commands to authenticate a PAC (Pointer MAC) and make the pointer unusable in case the verification process fails.

Being unbound from time of creation, generated MACs suffer from *reuse attacks*: attackers can try to substitute an authenticated address with another one generated in the past. In fact, once the MAC is checked, no issues can be found, as the signature is valid (and it will continue to be as long as the secret key and the modifier do not change). This is an issue that has to be addressed when designing processes that rely on these mechanisms.

2.5 The LLVM Compiler Infrastructure

LLVM[3] is a compilation infrastructure designed to help avoiding code duplication in different implementations (i.e., compilers for different languages and different architectures). This helps with producing target-independent and language-independent compiler optimization techniques that can be used in all compilation scenarios in a library-fashion, avoiding code duplication. In other words, LLVM is a collection of modules that can be used to easily create compilers.

In order to have modularity during the compilation macro-phases, LLVM defines an *Internal Representation* (IR): a low-level language with a high-level type system. This particular combination was chosen for optimization effectiveness [20].

The source code is translated into IR before being optimized. A number of optimization passes are then applied to the IR. Finally, the code is translated into Assembly which can be

[3] https://llvm.org

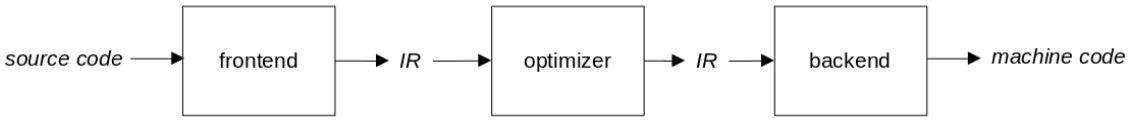

Figure 2.7: The 3-step LLVM compilation process.

trivially compiled by the target-specific assembler. During the translation into Assembly, the generated IR is not anymore "pure", as it slowly starts to contain more and more target-dependent instructions, until only those remain and the last phase of the backend is executed (i.e., actual assembly printing). It has to be noted that LLVM comprises many frontends and many backends. Thus, a better representation of Figure 2.7 would be with many frontends, one per supported language, all communicating with the same optimizer. Same thing would happen for the right side, where there is a backend for each supported hardware architecture.

In other words, the job of the frontend is to convert source code into *Static Single Assignment*-form (SSA-form) IR, that internally in LLVM will be represented with a series of `Instruction`s aggregated into `BasicBlock`s, grouped in various `Function`s. All of these objects are organized into an *Abstract Syntax Tree* (AST).

After optimization, the backend will convert all the objects into their `Machine-` counterpart before working on them.

Many different optimizing and compilation passes are implemented in LLVM, but, given the modularity nature of LLVM, there is no specific order in which to apply them. Some passes are even implemented more than once, using different strategies.

A compiler based on LLVM (such as `clang` [4]) is called *driver*, because it is nothing more than a program that defines which and in which order passes are executed.

Some drivers can also perform some language-specific, high level optimizations before translating the code into IR and using LLVM optimization passes. For instance, there exists a frontend for LLVM that transforms *Common Language Infrastructure* (CLI) into IR: in this case, for example during C# source code "compilation", there is some specific optimization for that language and then LLVM optimizations are run.

2.5.1 Compilation Passes

Many compilation passes are executed in order to transform source code into machine code. LLVM passes are mainly of type `FunctionPass` and `MachineFunctionPass`. A function pass receives a function as input, operates on it, and produces some output. The pass does not have to modify the function in order to be useful: many passes, like the live register analysis pass (Section 2.5.5), do not modify the function, but just analyze it.

However, all passes have to report if the AST was changed due to the execution of the pass itself.

[4] https://clang.llvm.org/

2.5.2 Basic blocks

As the name suggests, these are the lowermost instruction aggregates to which the compiler has to deal with. A basic block (defined in the classes `MachineBasicBlock` and `BasicBlock`) is a sequence of instructions that does not contain any branch instruction (i.e., they terminate with a branch instruction). This means that, by looking at a higher level, a program is just a sequence of interconnected basic blocks through which the execution flows.

Once execution starts at the beginning of a basic block, all the instructions inside the basic block are executed. This concept is important in compiler design, because it easily allows to apply some optimizations: for example, if an instruction is *data-independent* from the others in the same basic block (i.e., it does not use or generate data that is being generated or used by other instructions), it can be positioned by the compiler at any point in the basic block, maybe to speed up program execution time. For example, memory-reading instructions and other instructions that use the read data may be interleaved by data-independent instructions so that no time is wasted waiting for the memory to output data.

The notion of basic blocks is also used in other fields, for example in the design of superscalar processors.

Inside LLVM, a program is stored as a basic block graph, each terminating with a branch instruction (which may be conditional or not). Basic blocks aggregate into functions, which as we already stated (Section 2.5.1) are the information unit managed and elaborated by the different compilation passes.

```
isdivby4:       LDR X0, [SP, #8]
                MOV X1, X0
                TST X1, #0b11
                B.EQ divisibleby4
                MOV X0, #0
                BX LR
divisibleby4:   MOV X0, #1
                BX LR
```

Figure 2.8: A simple function subdivided in its basic blocks.

2.5.3 Machine instruction

`MachineInstructions` (MIs) are what compose machine basic blocks in LLVM.

It has to be noted that LLVM tracks more information than what Assembly programmers typically think of (i.e., AArch64 assembly language is at a higher level than "normal" machine Assembly). For example, a lot of Assembly instructions are *aliases* for less intuitive instructions, whereas in LLVM they have to be referred to with their real nature.

As a matter of example, the compare instruction (`CMP`) is always referred to as a subtraction that sets flags with the zero register as destination in LLVM.

Moreover, each instruction must specify for each register parameter if it is set or read, but also if that is the last read (register is marked as *killed*). This is important to track the *liveness* of registers, as live registers contain values used during execution of the program.

Now the focus moves to some compilation phases that are relevant to register spilling.

2.5.4 Register Allocation

In pure IR form (i.e., the form in which IR is as generated by the frontend), the code is memorized respecting the *Static Single Assignment* (SSA) form: when in this form, each variable is assigned once and once only. This makes easier for optimization passes to understand how information flows through the code being compiled: if a variable is not read by any following instruction, then it is safe to say that the value stored within the register is not useful for the execution of the program; in this case, on a RISC architecture such as AArch64, the register allocator may choose to assign this virtual register to the zero register `XZR` to effectively discard the result of the instruction. At the same time, a simple integer increase will be seen in SSA form by the definition of a new virtual register with assignment of the original plus one, the old virtual register will not appear in all the following IR lines.

Register Allocation is the pass in which each IR variable (i.e., *virtual register* in LLVM jargon) is assigned to a *physical* register.

To control register pressure, the register allocator may insert spilling and reloading code, and respectively allocate *spill slots* on the function stack frame.

The register allocator is actually a good example to show the modularity of LLVM. Indeed, the register allocator used by default in the AArch64 backend (the so-called *Greedy Register Allocator*) is not developed specifically for this architecture, but instead it draws information about it (in this case, how many registers are present, what are their purposes, and how they are overlapping) from an abstract description. This implies that, once that support for a novel architecture is added, it would only be necessary to describe this architecture in the aforementioned abstract format and all the quirks and perks of the Greedy Register Allocator (and for what matters, also other available register allocators) will be readily available for the newly-added architecture.

LLVM even includes a basic register allocator which assigns physical registers sequentially just to teach novel developers how a register allocator can be built.

The Greedy Register Allocator performs its task following the algorithm of *graph coloring* [12].

Graph coloring The problem of register allocation (finding out which variable to assign to which register and when and which register(s) to spill) is NP-complete, this means in practice that finding the optimal solution in a reasonable time is computationally challenging. The algorithm originally conceived by Chaitin is thus not developed to find an ideal solution but one of its approximations.

The algorithm tries to find a way to color the nodes of a graph $G = (V, E)$ using at most X colors. In this abstraction, the *nodes* V of the graph represent the *virtual registers*, while the available colors represent the available registers. The *edges* E of the graph represent interference between virtual registers: when two virtual registers interfere (i.e. their liveness range crosses) they will be connected in the graph through an edge. For this reason, G is called the *interference graph*.

The first step in the algorithm is to compute G from the virtual registers V and their computed live ranges. The following step is simplify the graph with the help of some observations:

1. If two nodes v_1, v_2 do not interfere (i.e. $(v_1, v_2) \notin E$) they can be considered as being

the same node v' with edges equal to the union of interferences of v_1 and v_2. This simplification is referred to as *coalescence of nodes*.

2. If a node interferes with a number of nodes less of equal to the number of colors, it is possible to avoid considering it during the execution of the algorithm as it will be surely colorable (this is because if $n_v < |X|$ whatever color we assign to the neighbors of v it is possible to assign to v the remaining color.

Coalescing two nodes is not always a good choice as the union of their interferences might be larger than the number of colors.

Spilling registers is needed when the graph is not $|X|$-colorable (i.e., there exist no combination of $|X|$ colors in such a way that the same color is never used on two connected nodes.

To find out if a graph is n-colorable is sufficient to apply the rules (1) and (2) sequentially. If at some point there is only one nodes in the graph consider (all have been either eliminated or coalesced and eliminated from the graph) then the graph is n-colorable: it is sufficient to assign a random color to the last node, then add back the last removed node and assign to it one of the remaining colors. The process continues until all nodes have been colored.

When is necessary to spill, it has to be chosen which register(s) to spill based on their spilling cost (registers with lower spilling cost are analyzed first).

The algorithm will notice it is necessary to spill when at some point the graph is no longer simplifiable and there still exists some nodes with degree greater than the amount of available colors. A spilled register s is represented as spilled by simply removing it from G.

The spilled register s has to be stored in a register when needed in computations. This fact is represented in the graph by treating it as a node t with a tiny liveness interval, only spanning that single assignment. Recall that it is straightforward to know when s is needed in computations as the IR prior to register allocation is in SSA form.

2.5.5 Live Register Matrix

This analysis pass is executed prior to register allocation and is used by the register allocator in order to know how virtual and physical registers interfere between them. Specifically, LLVM register allocators use the matrix produced by this pass to avoid allocating interfering virtual registers to overlapping physical registers (that could be for example `X1` and `W1` in AArch64 or `BX` and `BL` in 8086).

This matrix elongates in two dimensions: *register units* and frame indexes, having in each cell a list of live ranges (i.e., a instruction interval in which the register is used in the program). Register units are defined for each target supported by LLVM and are used to find registers contained within other registers. When a virtual register is allocated in a physical register, the live range of the virtual register is inserted into the matrix for each register unit belonging to that physical register.

Once a register is removed, the graph-reducing algorithm is launched again on the graph without spilled registers with the tiny interval(s) in place of the spilled register(s). If reduction stops again, more registers are spilled.

The cost of spilling a register can be estimated by counting how many times the corresponding virtual register is used.

2.5.6 Prolog-Epilog Inserter

The *Prolog-Epilog Inserter* (PEI) is the pass tasked with generating the code to set up the correct environment for the proper execution of each function in the program. This is done by surrounding each function with additional code that performs several tasks.

Most importantly, a prolog of a function spills *Callee-Saved Registers* (CSR): these are a group of registers, for `AArch64` defined in the ARM-ABI, that the called function has to spill in case they are used within the function itself. In other words, outside the environment of the function, these registers have to retain their original value.

The ARM-ABI also mandates where to store function call arguments, which registers serve which purpose, and where to store the return value of a function.

Another important task performed by the PEI is stack finalization (or *frame lowering*): in the IR, each function has an associated list of objects to be allocated on the stack, and the instructions that refer to an object on the stack temporarily use a frame index (FI, the index of the object on the function frame if one thinks of it as an array). After this pass, this list is no longer modifiable, and the references to the stack are converted to offsets with respect to the Stack Pointer (`SP`, a register holding the value to the stack *head*).

With stack finalization done, instructions to decrease `SP` (in the prologue) and increase `SP` (in the epilogue) are generated (in the case the stack grows down, as by default in `AArch64`). Usually, these instructions are merged with the first CSR save instruction with pre or post-indexing in order to speed up `SP` update if it is possible to so. Modifying `SP` means in practice allocating space on the stack for the local storage of the current function.

The clarifications made in these subsections pointed out that, in order to effectively protect register spillings, one has to intervene mainly in two spots inside the LLVM code: the register allocator and the PEI. The strategy adopted in both spill cases is detailed in Chapter 4.

Chapter 3

State of the Art

In this Chapter, the current state of the art on memory-protection mechanisms is offered. The summary is centered around ARM authentication.

3.1 Stack-Smashing Protection

The idea behind *Stack-Smashing Protection* (SSP) has been conceived quite some time ago [14], and is conceptually simple: in order to protect the stack from possible buffer overflows, a known value is positioned between local variables and the function stack frame (Figure 3.1), namely in the *canary* (derived from the fact that canaries were used in mining to check the presence of gas). If a buffer overflow happens, the faulty code will overwrite the canary value and the change of data will be noticed upon function return, where the canary value is compared against a known value.

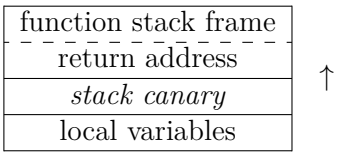

Figure 3.1: Stack canary positioned in a stack frame of a function

This technique was originally named StackGuard by the creators, and was eventually integrated into the `gcc` compiler in 2003 [40].

In order to be effective, the canary value obviously has to be unknown to attackers. The suggestion made in the original paper to keep canary values unknown is *randomization*: upon program launch, the random canary value is generated and the process will use that during its lifetime. This basic implementation is susceptible to brute-force attacks in some cases, for example in applications where the server forks to serve a client: in this case, the whole process memory is duplicated, comprising the secret canary. The attacker can try to guess the canary value and check if it was the correct one by detecting the server crashing while handling its request. This problem can be solved (and it has been) by randomizing the canary value at each execution of `fork()` [28].

Moreover, the stack canary, if not implemented correctly, can itself suffer from buffer overflow, depending where the canary correct known value is stored [7].

3.2 PACed Canaries

The strategy employed by Liljestrand et al. [21] removes the risk of the canary reference value leakage by generating it on a per-function basis with ARM PA. To mitigate MAC reuse attacks, the MAC is tied to the function signature and the stack pointer. In addition, to protect the return address, each function has additional canaries protecting one vulnerable buffer each. Ideally, each canary generated by this algorithm is unique.

Generation and verification of the value is done in chained fashion so that each value cannot be interchanged inside a function: apart from the one protecting the return address, each canary is actually just the signed address of the previous canary.

To check this type of canaries, the last one is loaded from memory, its value is authenticated and then dereferenced to load the previous one, up until the last one. If at any point data tampering is detected, dereferencing will yield a memory exception. In the paper, the first canary is just a plain MAC of the return address or the signed return address when `-msign-return-address` is in use.

3.3 Shadow Call Stack

The *Shadow Call Stack* (SCS) is a memory protection mechanism which avoids modifications to the return address saved into the stack by relocating all return addresses in another, separate stack. In current implementations in AArch64, this is done by reserving a register as stack pointer for the parallel stack [15].

For the correct functionality of this mechanism, the reserved register value has to be kept secret, otherwise it becomes possible for an attacker to try to get access to that particular memory area.

Even when the shadow stack pointer does not get revealed, this implementation does not protect the program against data-based attacks: by modifying data that is used in a test inside the process, an attacker can anyway influence the process control-flow.

3.4 PACStack

PACStack [22] designs a new strategy of creating an authenticated call stack that has the same overhead as previously designed hardware methods.

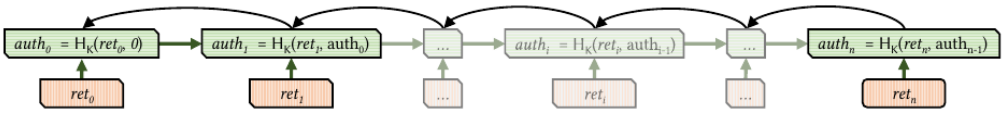

Figure 3.2: Return addresses chain generation. $H_K(data, mod)$ is a keyed hash function [22].

In particular, this novel technique is a re-visitation of the classical SCS that chains together authenticated return addresses in the normal stack. Basically, compared to the usual PA-oriented protection for AArch64 (which uses `SP` as modifier to sign the return address), this design uses the previous authenticated return address to sign the next one.

The latest authenticated address is kept on a reserved register (in the implementation provided by Liljestrand et al., this register is `X28`). Note how this cannot be avoided even with the presence of the `LR` register, because inside a function - where the address is supposed to be signed - `LR` will always be overwritten by the function calling instruction `BL` or `BLX`.

3.5 FIPAC

FIPAC, by Schilling, Nasahl, and Mangard [38], uses the latest PA extensions introduced to ARMv8: *EnhancedPAC2* and *FPAC* from ARMv8.**6**-A[26].

Contrarily to ARMv8.3-A, with these instructions it is possible to directly check if the authenticated address is valid or not (recall that when verification fails on 8.3, the address is modified in a way that when dereferenced an exception is triggered). What happens instead on 8.6 is that a trap is immediately generated upon failed verification. This method tries to protect CFI (Section 2.3.3).

The main idea used to enforce CFI in FIPAC is to update and check the *state* of the program, which is precisely defined in each of its points. In particular, the state value is managed by means of pointer authentication.

State changes are linked so checks do not have to be performed at every change (i.e., a single check at program termination would determine if the state got corrupted at some point during execution). In practice, it is better to check the state with a high frequency to notice a memory corruption reasonably soon after it happened.

3.6 RegGuard

Alongside to the work of this thesis, a similar work with a comparable goal was recently independently developed on the same environment (i.e., AArch64 with LLVM).

RegGuard[16] aims to protect vulnerable data that is present on the stack by keeping it preferably on registers, i.e., inside the CPU, thus removing the risk of an attacker modifying them.

To pursue this goal, Geden and Rasmussen developed an alternative register allocator (Section 2.5.4) for AArch64 on LLVM that assigns a *security score* to each variable in the program depending on their type which is assigned based on this safety ranking:

1. Pointers.
2. User-defined variables.
3. Condition variables.

This list has been ordered like shown based on what data attackers target the most while exploring for a possible vulnerability in programs. Indeed, pointers are the most exploitable when they can be corrupted due to the possibility for an attacker to route execution or read/write of data anywhere else in the address space.

As inevitably happens, when a function is called and CSRs have to be spilled on the stack, an integrity check digest of the CSRs is generated with HMAC-SHA256 (a keyed

hash function) and stored on the stack during the function prologue. In case the CPU does not support hardware-accelerated computation of the hash, a software hash calculating code is included that uses the SipHash2-4 algorithm[3] chosen for its fast and lightweight characteristics.

The work by Geden and Rasmussen reserves two registers: in one the secret key employed by the keyed hash function is stored, while on the other the latest computed hash is stored as to avoid storing it on memory

Chapter 4

Protector Implementation in LLVM

As already stated in Section 2.5, the implementation of the spill protector actually comprises two sub-projects: (i) protection of in-function spills and (ii) protection of CSRs.

In all cases, when a security check fails, there is no attempt nor possibility of recovery, and the process will call `__stack_chk_fail`, a function which was "borrowed" from the StackGuard plugin, normally invoked when the stack canary check fails (see Section 3.1 for more information on how StackGuard works). Given that the objective is protecting the stack against attacks, this function was deemed appropriate for our purpose.

Throughout of this implementation, the AArch64 instruction `PACGA` has been used. This instruction takes 3 register arguments:

1. the location where to store the integrity check (MAC);
2. the data to be protected;
3. a modifier to enhance randomness;

After its execution, the result register will contain 32 bits of pseudorandomness (a MAC). The instruction also draws information from a register not explicitly listed among the parameters (i.e., the secret key for the keyed hash) whose access is locked in an higher privileged mode of execution (Exception Level 1).

4.1 Threat Model and Assumptions

Implementing protection with ARM PA would be much harder if the attacker could read kernel memory, due to the kernel having to save the secret keys of all processes in their respective Process Control Block (PCB).

In this thesis, an attacker with powerful privileges is assumed: he or she can modify writable memory of the process during runtime, and can read all unprivileged process memory: it is assumed that kernel memory used to manage processes and their threads cannot be read by an attacker.

An ideal implementation of a register spill protector would be able to recognize an attack based on the value change of all kinds of register spills with overwhelming probability, optionally hiding their values with encryption when spilled in memory. At the same time, the solution has to bring a relatively low overhead, in order to not be a considerable burden on the system executing the secured program. Moreover, it preferably should not require additional hardware, as this adds to the cost of the solution.

4.2 Design Choices

4.2.1 Integrity Protection

Integrity protection refers to the ability of noticing modifications to some data and react accordingly when this happens. This is usually achieved by using some sort of keyed MAC (Message Authentication Code).

Keyed MACs are mathematical functions that create a digest from the data that is to be protected, and a secret key. If the MAC is cryptographically secure, an attacker cannot forge a new valid MAC for new, never-seen-before data. The attacker potentially has nonetheless access to all previously generated MACs and their corresponding data (still, MAC forging has to be computationally unfeasible when the algorithm is cryptographically secure).

To protect the spilled registers, it is necessary to compute a MAC protecting them and store the result somewhere. Since spilling a register is a hint that suggests the compiler is experiencing high register pressure, it is best to store the generated MACs off-chip, i.e., it is convenient to store them on the stack along with the data they protect.

To save stack space, it would be best to create a single MAC that protects all spills. This cannot unfortunately be easily done, since in-function spills and CSR spilling/reloading do not happen at the same time during the life of a function. To balance space and computation time, it has been chosen to create a MAC for each spill slot allocated for in-function spills and a single MAC encompassing all CSRs.

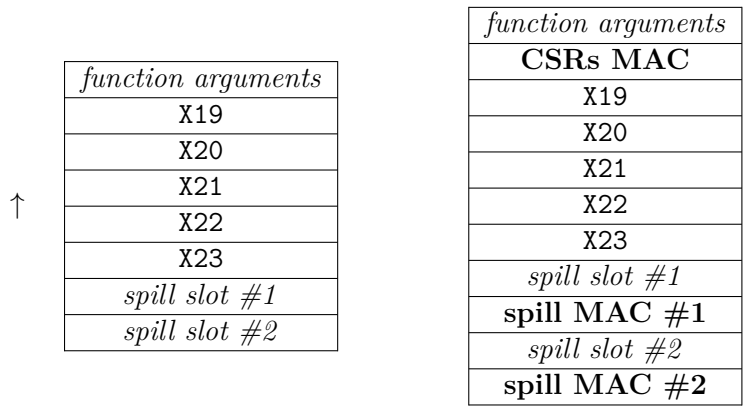

Figure 4.1: Example of a function stack frame structure with and without MACs inserted.

The CSRs MAC is computed during the prolog and pushed to the stack alongside the CSRs. Once the function terminates, the CSRs are reloaded to the CPU and their MAC

is recomputed and checked against the version present on the stack.

Regarding MACs of in-function spills, those have to be computed once the corresponding register is about to be spilled, because its value will be defined only at that point in the execution.

In order to make it more difficult for an attacker to successfully forge a MAC, the design takes advantage of a keyed tweakable MAC function with `SP` as modifier. In this way, the MAC randomness will be enhanced.

The MAC of CSRs is calculated by computing the MAC of the first CSR and then using this value to mask the following register. The masked register is then used as modifier to compute the MAC for the third register, and so on: in this way the computation is sped up as masking a register takes less time than calculating its integrity code.

The process is visually illustrated in Figure 4.2.

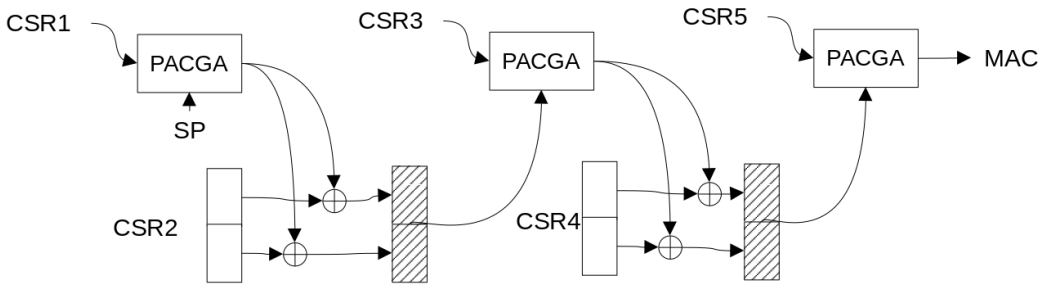

Figure 4.2: MAC generation process for 5 CSRs

4.2.2 Encryption

The strategy adopted for encryption is, for both spill kinds, to encrypt before computing the MAC, in order to take advantage of all benefits of *Encrypt-Then-Authenticate* encryption schemes [6].

For each function, LLVM maintains a list of objects that the function will allocate on its stack frame. Instructions interacting with the stack refer to an index in this array (called *Frame Index*, FI) rather than an actual `SP`+*offset* until the frame is finalized.

To encrypt registers, the same MAC-generating instruction is used as a pseudorandom number generator (PRNG) and its results are XORed to the registers. The PRNG is seeded with the stack pointer, and a small hash of the function signature and the frame index where the MAC for that(those) register(s) is stored. Both FI and the function signature are available at compile time, so the hash can be computed based on them. The process is illustrated in Figure 4.3.

It has been chosen to use an hash instead of simply generating a pseudo-random number within the compiler, because it is important that built programs have a machine code that can be uniquely derived from the source code and the compiler. In other words, built programs have to be reproducible, so that users can independently check that the binary executable they have is not compromised [29] [34]. More numbers are generated by applying the PRNG to its previously generated number.

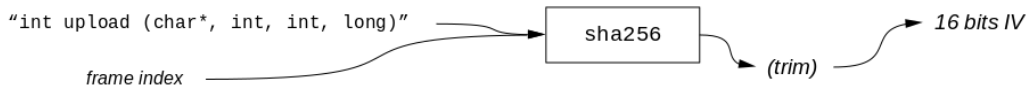

Figure 4.3: Generation of the Initialization Vector (IV) for the encryption.

Generating a value unique to the function is used to make less probable the fact that multiple registers are encrypted using the same keystream. It is appropriate to talk about a keystream, as the PRNG is used in a stream-cipher fashion. Using the same keystream over different data would be bad for confidentiality, as XORing different cipher streams would reveal the XOR of plain texts.

4.2.3 Security evaluation

In this design, when multiple spills occur within a function, equal spill values will imply the same MAC. At the same time, different spill values could be swapped successfully if also their MACs are switched. Both of these phenomena happen because the modifier used during computation (`SP`) is the same.

A way to mitigate this could be for instance to randomize `SP` prior to using it as modifier so that it is unique for each spill, similarly on how it is done in PACStack (see Section 3.4). Without changing the design, it is also possible to use encryption which completely masks the value of registers in a way that the same unencrypted values will not probably have the same encrypted value.

4.3 Implementation Details

The illustrated spilling protector was implemented in LLVM 12. To implement this design in LLVM, it is necessary to modify the AArch64 backend in several places.

4.3.1 Integrity Protection

Integrity protection has been implemented in in-function spills by defining two new pseudo-instructions in the AArch64 instruction information file (`AArch64InstrInfo.td`), namely `SecureSpill` and `SecureReload`.

Pseudo-instructions are commonly used in LLVM as placeholders so that in later stages of compilation they can be replaced with other instructions. The compilation process will fail if, at the latest stages, there are still some pseudo-instruction left in the IR.

In this case, these pseudo-instructions are inserted by the compiler once it invokes `storeRegToStackSlot` or `loadRegFromStackSlot`. These two functions are used by the compiler to select the correct instruction for spilling or reloading a specific register from/to a specific stack slot.

Normally, the compiler would choose a `STR` (store instruction) or `LDR` (load). When `-aarch64-enable-spill-protection` option is passed to the backend, these functions will select the defined pseudo-instructions in place of normal store or load instructions

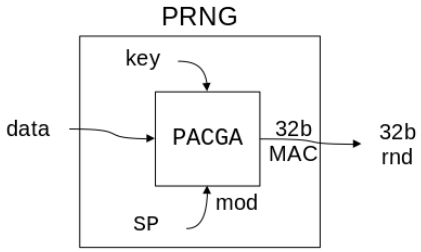

Figure 4.4: Construction of the PRNG function.

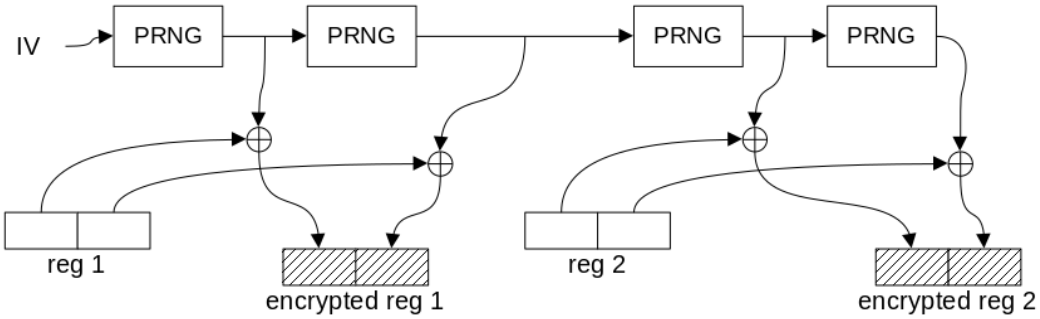

Figure 4.5: Encryption of multiple registers.

with the same arguments when compatible registers (64-bit general-purpose registers) are spilled (and the processor supports ARM PA).

However, `storeRegToStackSlot` or `loadRegFromStackSlot` are used also by other passes (for example, the register scavenger). Modifying those functions is thus a good approach, because in a single modification, all spills inserted by all stages are encompassed, even though the principal stage inserting spills will always be the register allocator.

A new pass has been inserted in the compilation pipeline after register allocation (presented in Section 2.5.4) to expand those instructions. In this function pass, all basic blocks in the function are searched for the two novel pseudo-instructions and, if found, they are expanded one by one. During the expansion of `SecureReload` instructions the implementation also has to modify the AST by splitting the basic block containing `SecureReload` and also insert a new basic block to call the error-handling function `__stack_chk_fail`.

After elaboration of a function, it might happen that a function has several of these newly-added basic blocks that only contain `BL __stack_chk_fail`. This is not an issue, as LLVM automatically removes duplicates towards the end of compilation in order to reduce code size.

The MAC is stored on a 32-bit stack slot, so it can fit without wasting space. Unfortunately, this requires the emission of additional instructions to shift `X15`, as `PACGA` computes the MAC on the high word of the result register. An alternative implementation could speed up MAC handling but sacrifice stack space.

Details on how the two pseudo-instructions are expanded are shown in Figure 4.6. It is possible to notice how the routine-calling instruction `BL` is placed at the end of the function. This is not by chance: microprocessors are normally faster in executing the instruction following a conditional jump with respect to the one pointed by the jump. This happens because while the branch condition is evaluated, the following instruction is elaborated. LLVM generates code like this because, in the implementation, the "successful" branch (i.e., the execution path where the MAC check passes) is marked as much more probable in order to favor execution in this case.

In short, the code is optimized for the case when all MAC checks pass, as this is what has to be prioritized: favoring instead the "fail" path would not improve the user experience, as the application is anyway terminating.

In order to calculate and check the MAC, two additional registers are needed. In this implementation, `X14` and `X15` have been reserved for the exclusive use of the register protector. This implementation choice was made because LLVM takes for granted that the spilling and reloading instructions only use the register being transferred, which is not the case in our implementation. Changing this behavior in LLVM was deemed too challenging for the purposes of this thesis, but represents a possible improvement over the current state of the spill protector.

MAC generation for an in-function spill is a single instruction that takes $4cc$ (clock cycles) to execute [23] [5]. The process is further slowed down by the necessity to store the MAC in memory.

During prologue and epilogue, it is not actually necessary to reserve any register, as the function code (that the compiler will compile so that it leverages on all available registers for performance) is not executing yet (or anymore). It has been chosen to anyway stick to `X14` and `X15` to be consistent with the in-function spill protector.

When a MAC check fails, it means that possibly an attack is in progress (it could also

```
...code...

SecureSpill X0, %stack.1


...code...


SecureReload X0, %stack.1


...code...
```

```
...code...
STR X0, %stack.1
PACGA X15, X0, SP
LSR X15, #32
STR W15, %stack.2
...code...
LDR X0, %stack.1
LDR W14, %stack.2
LSL X14, #32
PACGA X15, X0, SP
CMP X14, X15
B.NE check_fail
...code...
check_fail:
BL __stack_chk_fail
```

Figure 4.6: Before and after pseudo-instruction expansion with only integrity protection active. Note how addresses are not yet defined, as frame finalization will be done later in the compilation pipeline.

```
leaf_function:
  PACGA   X15, X25, SP
  EOR X15, X15, X15, LSR #32
  EOR X15, X28, X15
  PACGA   X15, X27, X15
  PACGA   X15, X26, X15
  STR X15, [SP, #-48]!
  STP X28, X27, [SP, #16]
  STP X26, X25, [SP, #32]

  ...function code...
```

(a) prologue

```
...function code...

STR X8, [X0]
LDP X26, X25, [SP, #32]
LDP X28, X27, [SP, #16]
LDR X15, [SP], #48
PACGA   X14, X25, SP
EOR X14, X14, X14, LSR #32
EOR X14, X28, X14
PACGA   X14, X27, X14
PACGA   X14, X26, X14
CMP X14, X15
B.NE    MAC_FAIL
RET
.MAC_FAIL:
BL  __stack_chk_fail
```

(b) epilogue

Figure 4.7: Prologue and epilogue as generated with integrity protection active in a leaf function using `X25-X27` among the CSRs.

just be a memory corrupting bug in the program, but that has to be avoided in any case as explained in Chapter 2). The MAC scheme adopted cannot detect what changed from when the integrity code was computed, so the safest thing to do is to terminate the program

immediately. For this, `__stack_chk_fail` is called. In Linux, this function prints a static error message and then exit the program.

In the case of CSRs, it is not necessary to insert pseudo-instructions and later expand them, as CSRs spilling and reloading code is generated in a specific place inside LLVM, namely `emitPrologue` and `emitEpilogue`. These generic functions are declared for any target supported by LLVM, and are in charge of generating the prologue (epilogue) code for a specific function in the given basic block. As in the case of AArch64 prologue and epilogue generation is a complex matter, it has been decided decided to inject the code alongside CSR handling routines: `spillCalleeSavedRegisters` and `restoreCalleeSavedRegisters`.

These functions are tasked with generating the code to store and load CSRs, and are declared withing the frame lowering (i.e., frame finalization) file `AArch64FrameLowering.cpp`. When the `-aarch64-integrity-protect-csr` option is passed to the backend, and a suitable environment is found (spilled CSRs are 64-bit GPRs and the processors supports ARM PA) instructions to compute and check the tag are inserted in the prologue and epilogue of the function (see Figure 4.7). In case the function is not a leaf function, `LR` and `FP` counts as CSR in LLVM, so they are protected as well by the MAC. For leaf functions, those registers are not spilled so they can be considered safe.

When calculating the MAC over CSRs, `X15` is marked as CSR so that LLVM will automatically save and reload it to/from a stack slot when the implementation needs it. Obviously, `X15` is not included in the MAC calculation even though LLVM considers it a CSR.

MAC calculation for CSRs takes up $3cc/CSR$ on average. Also, the instructions are inserted in a portion of code such that the liveness tracking information of the "default" prologue and epilogue does not need to change. When LLVM spills CSRs it marks them as *killed* in the IR: this means that those registers are available to be overwritten in the following instructions. At the same time, LLVM marks registers reloaded in the epilogue as *defined*, so the spill protector can read from them later, and the liveness analysis of CSRs will not be affected by the spilling protection system.

4.3.2 Alternative Implementation

For the MAC computation `X15` is not strictly needed as it is possible to just generate the MAC in the same register that is being spilled, overwriting it once it is already stored to memory in a fashion similar to Listing 4.8a.

Additionally, it is possible to reserve only a single register (instead of two) if one can assure the application to be *single-threaded*. An example can be seen in Listing 4.8b, and an explanation as of why this was not implemented follows.

To check the MAC, only one additional register is needed when the application is single-threaded, but we need to assume an attacker cannot write to process memory while the check is being done. This is a reasonable assumption if the attacker relies on some vulnerability in the compiled program to modify memory, as there is no unsafe instruction between the two register reloads that could lead to that scenario occurring.

If the applications was multithreaded and the alternative reload would be in use, there is the possibility (even if small) that a different thread than the one executing the reload could corrupt the spilled register between the two `LDR X25` (according to the example at Figure 4.8b): the thread checking the MAC would see that the tag is correct before

```
STR X25, [SP, #16]
PACGA X25, X25, SP
STR X25, [SP, #24]
```

(a) Alternative spill

```
  LDR X25, [SP, #16]
  LDR X15, [SP, #24]
  PACGA X25, X25, SP
  CMP X15, X25
  B.NE checkfailed
  LDR X25, [SP, #16]
  ...
checkfailed:
  BL
      __stack_chk_failed
```

(b) Alternative reload

Figure 4.8: Alternative implementations ideas of spill and reload, respectively

loading `X25` again from memory, but not computing anymore its associated MAC, thus experiencing a *time-of-check-to-time-of-use* race condition [8].

The alternative reload has been discarded because it requires more memory accesses than the one eventually developed, and because it would work only in single threaded scenarios, significantly reducing the applications where this implementation could be put to use.

Once (at least) a register is reserved, it becomes straightforward to see the drawbacks of the alternative spill code: it would render all the instructions data-dependent on `X25`, slowing down the processor pipeline. As a nice side-effect of always using `X15`, there is also the plus of having a more easily-readable LLVM codebase.

4.3.3 Encryption

For encryption, the spilled registers are XORed with a random bit stream generated (32 bit at a time) with `PACGA`. Each spill "session" inside a function is paired with a different initialization vector (IV) of 16 bits.

The IV, calculated at compile time with a hash function over the data indicated in the design (Figure 4.3), is stored in a register with a `MOVZ` instruction which, in contrast to most other AArch64 instructions, permits transferring a 16-bit immediate in one go. Recall that an immediate is a numeric value stored directly in the machine instruction, thus rendering it *immediately* available, contrarily to values stored in registers or even memory.

The main downside of encrypting with random bits generated in this way is that `PACGA` only generates 32 pseudo-random bits. Therefore, in order to completely mask a register, it has to be run twice: this means the process is at least 50% slower than it would have to be with 64-bit support.

For in-functions spills, the encryption code is generated when `--aarch64-enable-spill-encryption` is passed to the backend. The code is added during the function pass `AArch64SpillProtection`, while expanding the pseudo-instructions that were already used for enforcing integrity protection.

The CSRs encryption code is generated, when the `-aarch64-encrypt-csr` is present, before the registers are authenticated in the prologue and after a successful MAC check

in the epilogue. The code generated is similar to the one present in Listing 4.1 where, similarly to Figure 4.7 registers `X25` to `X28` are CSRs for this function.

```
MOV X15, #272              ; 272 = trim(h(function_id, FI))
PACGA   X15, X15, SP       ; Generation of 1st keystream block.
EOR X25, X25, X15          ; Masking the high word of 1st CSR.
PACGA   X15, X15, SP       ; Generation of 2nd keystream block.
EOR X25, X25, X15, LSR #32 ; Masking the low word of 1st CSR.
PACGA   X15, X15, SP       ; |
EOR X26, X26, X15          ; |
PACGA   X15, X15, SP       ; | Encryption of 2nd CSR.
EOR X26, X26, X15, LSR #32 ; |__
PACGA   X15, X15, SP       ; |
EOR X27, X27, X15          ; |
PACGA   X15, X15, SP       ; | Encryption of 3rd CSR.
EOR X27, X27, X15, LSR #32 ; |__
PACGA   X15, X15, SP       ; |
EOR X28, X28, X15          ; |
PACGA   X15, X15, SP       ; | Encryption of 4th CSR.
EOR X28, X28, X15, LSR #32 ; |
```

Listing 4.1: Function prologue extract encrypting CSRs

Note that in stream ciphers the same algorithm of encryption (i.e., XORing the keystream to the plain text) is used in decryption, so the code in Listing 4.1 is used by the function also in the epilogue.

It can be also noted how this process is cumbersome: if `PACGA` takes 4 clock cycles to execute, and `MOV` and `EOR` (the XOR instruction) takes one then the process to encrypt n CSRs takes c clock cycles, as defined below.

$$\begin{aligned} c &= n(2 \cdot \texttt{PACGA}_t + 2 \cdot \texttt{EOR}_t) + \texttt{MOV}_t \\ &= n(2 \cdot 4 + 2 \cdot 1) + 1 \\ &= 10n + 1 \end{aligned}$$

4.4 Security Evaluation

4.4.1 Spill Tampering Detection

Our solution generates code to calculate and store an integrity check value alongside spilled registers, be them Callee-Saved Registers or spills due to register pressure.

Attackers can swap spilled values with their MACs among themselves, but they are limited to functions that are executed with the same Stack Pointer value. When encryption is used, the register values are masked in a way tailored to each specific function instance by combining the MAC frame index, the function signature and `SP`. In this case, according to the well known *birthday paradox* [39] attack, an opponent has to collect about $2^{16} = 65536$ valid MACs before finding a collision with a non-negligible probability and be able to swap two spills without the spill protector not being able to notice.

Note also that the attacker has only one try when tampering with program memory, as the program aborts on a MAC mismatch.

4.4.2 Improvements

The implementation presents some margin for improvement:

- The register allocator does not take into account the registers used/defined by the spilling instruction, instead taking for granted that only the register being transferred is the one used. This is important also in case someone wants to implement a spill protector for some other architecture;
- Unless encrypted, in-function spilled register values can be swapped, alongside their MAC, inside functions that have the same `SP` value. The attack surface is still reduced with respect to not having a spill protector;
- Unless encrypted, spilled CSRs values are interchangeable between different functions as long as the two functions have same `SP` value *in the prologue/epilogue* (i.e., the same `SP` value in their parent function) and the spilled CSRs are the same ones. See Figure 4.9 for a practical example.

```
int parent_funct (int x, int y) {
    int i, n;
    for (i = 0; i < 10; i++) {
        n = vulnerable_funct_1(x);      ←
        x /= 7;
        vulnerable_funct_2(y, &x);      ←
        vulnerable_funct_3(y, n, x);    ←
    }
}
```

Figure 4.9: The functions indicated by the arrow will have the same value of `SP` during execution of the prologue and epilogue. If they spill the same CSRs, their values could be swapped among functions without the spill protector being able to notice.

4.5 Experimental Results

To evaluate the performance of the developed spill protector, the SPEC CPU benchmark [1] has been used. At its current version, SPEC (*Standard Performance Evaluation Corporation*) CPU 2017 is essentially a program capable of orchestrating building and running a preselected variety of open source programs of which it contains the sources, comparing their output to predefined ones which are known to be correct.

Among the different test suites, it has been chosen to use `intrate`, as this suite is the one that performs benchmarks on the integer throughput of the system. This suite puts special pressure on integer registers inside the processor.

The current implementation is still on a prototype level, and not all complex tests run to completion. Work is being done to solve these issues but reported in this thesis are only the tests that complete.

4.5.1 Executable Size

binary name	no protection	integrity	encryption+integrity
`perlbench_r_base`	3.0MB	3.3MB (+11.5%)	3.9MB (+31.7%)
`cpugcc_r_base`	11MB	13MB (+14.7%)	15MB (+41.3%)
`mcf_r_base`	688KB	696KB (+1.10%)	712KB (+3.49%)
`omnetpp_r_base`	2.6MB	3.0MB (+16.2%)	3.8MB (+50.2%)
`cpuxalan_r_base`	6.5MB	7.3MB (+12.2%)	8.9MB (+37.5%)
`imagevalidate_525_base`	623KB	627KB (+0.60%)	631KB (+1.29%)
`ldecod_r_base`	1.3MB	1.4MB (+8.10%)	1.6MB (+22.5%)
`x264_r_base`	1.3MB	1.4MB (+7.90%)	1.5MB (+19.0%)
`deepsjeng_r_base`	112KB	124KB (+10.8%)	144KB (+28.8%)
`xz_r_base`	793KB	821KB (+3.50%)	877KB (+10.6%)

Table 4.1: Executable sizes with different levels of protection applied.

The increased sizes of compiled executables can be seen in Table 4.1.

This outcome is expected, as essentially the spill protector replaces single instructions with a series of different instructions whenever a spill occurs.

It can be noticed how the increase of instructions is particularly relevant for encryption: on average, the code is:

- 7.9% bigger with only integrity protection active (with respect to no protection).
- 24.5% bigger with both integrity protection and encryption active (with respect to no protection).

These figures are large when compared to other means of protection presented in Chapter 3 which are already in use (stack canaries and SCS). This illustrates the fact that this

[1] https://www.spec.org/cpu2017/

work pinpoints a security issue that is prevalent, but often overlooked in prior microarchitectural security work: register spills are very common in compilation, and their presence is a potential integrity and also confidentiality risk.

It has to be taken into account that this design defends against modification (and optionally eavesdropping) of *all* register spills: this comprises also the return address `LR` and the frame pointer `FP`. This implies that, when this solution is used, the traditional way of using ARM PA to sign the return address is not as necessary.

When reading Table 4.1, one has also to take into account how executable size influences its running speed in the processor: apart from the larger number of instructions, the instruction cache (a fast memory that holds the instructions that probably will be executed next) will more frequently empty, thus requiring access to memory to be refilled.

Another aspect that can be noticed is how broad the size increase is among the selected executables: regarding integrity, the observed minimum is +0.60%, while the maximum is +16.2%, more than double. This aspect directly derives from how much an executable is in need of registers during its lifetime as a process. A program like `omnetpp_r_base`, which performs simulations of a large Ethernet network clearly needs to elaborate more data at the same time than a program such as `imagevalidate_525_base`, which is used by the `x264_r` benchmark just to check if the reencoded video is valid. A value such as the latter shows that the program in question could almost run entirely on the CPU without utilizing memory for spilling, while the former would benefit considerably if the number of registers on the CPU were higher.

Chapter 5

Conclusions and Future Work

In this thesis, we presented a register spill protector based on LLVM 12, capable of recognizing when registers spills of any kind have been modified during their stay in memory. Registers can optionally be encrypted to ensure confidentiality, rendering attackers without a clue of what is being stored within them.

Compiled executables are somewhat larger than the unprotected ones, especially when both encryption and integrity protection are employed, increasing their size by about 25%.

Unfortunately, it is not possible at the moment to obtain a performance speed evaluation as the spill protector, at its present form, produces files which are not executable when the source code is substantial. With this implementation completely working, programmers would be able to protect their developed code against attacks targeting the local stack (notably the return address and frame pointer therein stored) while also preventing the modification of integer variables that are spilled on the stack, thus reducing the attack surface for DOP.

5.1 Future Work

There are a number of improvements that can be applied to the work presented in this thesis:

- LLVM register allocators could become aware of what instruction is selected to spill or reload a register. In this way, spill protectors could avoid reserving registers on the machine, but rely instead on the register allocator to know that once a register has to be spilled another one (or two) needs to be available. This could possibly be done also by modifying the Live Register Analysis pass (Section 2.5.5).
- Preventing the possibility of swapping spilled registers (and their MAC) in different functions at the same stack level or within the same function when no encryption is used. This could be done by XORing a number unique to the spill in the modifier `SP` prior MAC generation.
- Preventing CSR MAC swapping between different functions at the same parent `SP` value when CSR encryption is not in use. To implement this is necessary to move the MAC calculation at the end of the prologue rather than its beginning. The matter

is more complicated at the epilogue where calculation still has to be done before `SP` gets updated but after CSRs are reloaded. In both cases it is needed to alter how LLVM sets liveness flags for instructions in the prologue and epilogue.

- Rendering the protection *selectable* on a per-variable or per-function basis rather than either enabling or disabling it globally for all spills: to alleviate the overhead, it should be possible to selectively decide which variable or function to protect with keywords in the source code. There could be also different keywords depending on if the programmer desires integrity protection only or also encryption. In order to do this one would have to modify an LLVM *frontend* to introduce new *attributes* that have to be maintained and caught in the backend to selectively apply the instrumentation.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. «Control-flow integrity principles, implementations, and applications». In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[2] Inc Advanced Micro Devices. *Processor Programming Reference (PPR) for AMD Family 17h Model 18h, Revision B1 Processors*. Apr. 14, 2021. URL: https://developer.amd.com/wp-content/resources/56176_ppr_Family_17h_Model_71h_B0_pub_Rev_3.06.zip.

[3] Jean-Philippe Aumasson and Daniel J Bernstein. «SipHash: a fast short-input PRF». In: *International Conference on Cryptology in India*. Springer. 2012, pp. 489–508.

[4] Various authors. *Chromium*. URL: https://www.chromium.org/Home.

[5] Roberto Avanzi. «The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes». In: *IACR Transactions on Symmetric Cryptology* (2017), pp. 4–44.

[6] Mihir Bellare and Chanathip Namprempre. «Authenticated encryption: Relations among notions and analysis of the generic composition paradigm». In: *Journal of Cryptology* 21.4 (2008), pp. 469–491.

[7] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. «Smashing the Stack Protector for Fun and Profit». In: *ICT Systems Security and Privacy Protection*. Ed. by Lech Jan Janczewski and Mirosław Kutyłowski. Cham: Springer International Publishing, 2018, pp. 293–306. ISBN: 978-3-319-99828-2.

[8] Matt Bishop, Michael Dilger, et al. «Checking for race conditions in file accesses». In: *Computing systems* 2.2 (1996), pp. 131–152.

[9] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. «Mitigating code-reuse attacks with control-flow locking». In: *Proceedings of the 27th Annual Computer Security Applications Conference*. 2011, pp. 353–362.

[10] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. «When good instructions go bad: Generalizing return-oriented programming to RISC». In: *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, pp. 27–38.

[11] Nicholas Carlini and David Wagner. «{ROP} is still dangerous: Breaking modern defenses». In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 385–399.

[12] Gregory J Chaitin. «Register allocation & spilling via graph coloring». In: *ACM Sigplan Notices* 17.6 (1982), pp. 98–101.

[13] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. «ROPecker: A generic and practical approach for defending against ROP attack». In: (2014).

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. «Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.» In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[15] LLVM developers. *ShadowCallStack*. 2021. URL: https://clang.llvm.org/docs/ShadowCallStack.html.

[16] Munir Geden and Kasper Rasmussen. «RegGuard: Leveraging CPU Registers for Mitigation of Control-and Data-Oriented Attacks». In: *arXiv preprint arXiv:2110.10769* (2021).

[17] Laurence Goasduff. *Gartner Says Global Smartphone Sales Continued to Decline in Second Quarter of 2019*. 2019. URL: https://www.gartner.com/en/newsroom/press-releases/2019-08-27-gartner-says-global-smartphone-sales-continued-to-dec.

[18] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. «Data-oriented programming: On the expressiveness of non-control data attacks». In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 969–986.

[19] Tim Kornau et al. «Return oriented programming for the ARM architecture». PhD thesis. Master's thesis, Ruhr-Universität Bochum, 2010.

[20] Chris Arthur Lattner. «LLVM: An infrastructure for multi-stage optimization». PhD thesis. University of Illinois at Urbana-Champaign, 2002.

[21] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N Asokan. «Protecting the stack with PACed canaries». In: *Proceedings of the 4th Workshop on System Software for Trusted Execution*. 2019, pp. 1–6.

[22] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. «PACStack: an Authenticated Call Stack». In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 357–374. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[23] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. «{PAC} it up: Towards Pointer Integrity using {ARM} Pointer Authentication». In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 177–194.

[24] Arm Ltd. *Arm Architecture Reference Manual Armv8, for A-profile architecture*. July 22, 2021. URL: https://developer.arm.com/documentation/ddi0487/gb.

[25] Arm Ltd. *Armv8-A architecture: 2016 additions*. Oct. 26, 2016. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-a-architecture-2016-additions.

[26] Arm Ltd. *Developments in the Arm A-Profile Architecture: Armv8.6-A*. Sept. 25, 2019. URL: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-developments-armv8-6-a.

[27] Arm Ltd. *Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)*. Oct. 27, 2021. URL: https://github.com/ARM-software/abi-aa/blob/320a56971fdcba282b7001cf4b84abb4fd993131/aapcs64/aapcs64.rst.

[28] Hector Marco-Gisbert and Ismael Ripoll. «Preventing brute force attacks against stack canary protection on networking servers». In: *2013 IEEE 12th International Symposium on Network Computing and Applications*. IEEE. 2013, pp. 243–250.

[29] Ed Maste. *Reproducible Builds in FreeBSD*. 2016. URL: https://people.freebsd.org/~emaste/AsiaBSDCon-2017-Reproducible-Builds-FreeBSD.pdf.

[30] mikben, v-kents, DCtheGeek, drewbatgit, and msatranjr. *Data Execution Prevention*. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[31] MITRE. *Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses*. 2021. URL: http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[32] Aleph One. «Smashing the stack for fun and profit». In: *Phrack magazine* 7.49 (1996).

[33] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. «Transparent {ROP} exploit mitigation using indirect branch tracing». In: *22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 2013, pp. 447–462.

[34] Mike Perry, Seth Schoen, and Hans Steiner. «Reproducible builds. moving beyond single points of failure for software distribution». In: *Chaos Communication Congress*. 2014. URL: https://media.ccc.de/v/31c3_-_6240_-_en_-_saal_g_-_201412271400_-_reproducible_builds_-_mike_perry_-_seth_schoen_-_hans_steiner.

[35] Inc. Qualcomm Technologies. *Pointer Authentication on ARMv8.3*. 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[36] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. «Return-oriented programming: Systems, languages, and applications». In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34.

[37] Mark Rutland. *Pointer authentication in AArch64 Linux*. July 19, 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/arm64/pointer-authentication.txt?id=fbedc599e9b891a6756b1c9bc2eead02b02cce77.

[38] Robert Schilling, Pascal Nasahl, and Stefan Mangard. «FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication». In: *arXiv preprint arXiv:2104.14993* (2021).

[39] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. «Birthday paradox for multi-collisions». In: *International Conference on Information Security and Cryptology*. Springer. 2006, pp. 29–40.

[40] Perry Wagle, Crispin Cowan, et al. «Stackguard: Simple stack smash protection for gcc». In: *Proceedings of the GCC Developers Summit*. Citeseer. 2003, pp. 243–255.

[41] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. «CFIMon: Detecting violation of control flow integrity using performance counters». In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE. 2012, pp. 1–12.

Acknowledgements

This thesis would not have been possible without the help from my collegues at Helsinki System Security Laboratory and other people from Politecnico di Torino.
I want to especially thank Carlos Chinea Perez, Jan-Erik Ekberg, Gianluca Roascio and Paolo Ernesto Prinetto.

I cannot avoid to thank also my family and Max for supporting and motivating me during my whole university career.