Computers & Security 129 (2023) 103213

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

RegGuard: Leveraging CPU registers for mitigation of control- and data-oriented attacks

Munir Geden*, Kasper Rasmussen

Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

ARTICLE INFO

Article history:
Received 2 December 2022
Revised 16 February 2023
Accepted 27 March 2023
Available online 29 March 2023

Keywords:
Security
Compiler
Register allocations
Memory attacks

ABSTRACT

CPU registers are small discrete storage units that are used to store temporary data and instructions within the CPU. Registers are not addressable in the same way memory is, which makes them immune to memory attacks and manipulation by other means. In this paper, we take advantage of this to protect critical program data with integrity guarantees that cover register spills. This protection effectively addresses control- and data-oriented attacks targeting the stack, even by adversaries with the full knowledge of program memory. Our solution RegGuard is a software-based mitigation technique that uses existing CPU registers and cryptographic primitives to protect critical variables with hardware-level assurance. Unlike conventional register allocation methods, RegGuard prioritises the security significance of a register candidate over its expected performance gain. Our scheme also deals effectively with saved registers to the stack, i.e., when the compiler frees registers to make room for the variables of a new call. With RegGuard, register values saved to the stack are protected, including strong adversaries with arbitrary read and write access capabilities. While our primary design focus is on security, performance is important for a scheme to be adopted in practice. RegGuard is still benefiting from the performance gain normally associated with register allocations and provides practical protection. Despite being adaptable to different CPU architectures, we showcase the performance of RegGuard using different benchmark programs and the C library on the ARM64 architecture as a proof-of-concept.

1. Introduction

Despite many years of effort, memory bugs continue to be one of the root causes of software security problems, especially in applications developed using unsafe languages like C, which are commonly used in systems programming and performance-critical applications. Since there is no built-in safety in those languages to prevent unintended access to critical program data, an attacker exploiting a memory bug in the program (e.g. stack buffer overflow) can overwrite control and data objects beyond the abstraction given in the source code.

Several schemes have been proposed to mitigate such exploits of memory bugs. The majority of these focus on *control-oriented* attacks in which code pointers are targeted. For example, stack canaries (Cowan et al., 1998) place random values to detect overflows onto return addresses. But these canaries fail to catch well-targeted corruptions, e.g., format string attacks, that alter certain addresses and leave the canary untouched. More powerful control-flow protections such as CFI (Abadi et al., 2005) and CPI (Kuznetsov et al., 2014) do not make assumptions about how the corruption happens. Those either use a shadow stack to detect corruptions of (shadowed) control data or a safe stack to protect them from being altered. In general, control-flow protections do not cover *data-oriented* attacks that selectively target non-control data, for example, a function argument or a condition variable deciding on the execution of a privileged branch. Proposed defences against those attacks, e.g., data-flow integrity (DFI) (Miguel et al., 2006), generally require a more thorough check of all stack accesses in addition to code pointers, and in the process they introduce high performance overheads.

Regardless of their limitations, current proposals for both attack classes face three common challenges in general. The first one is the performance overheads due to the instrumentation accompanying each legitimate memory access, which worsens as the coverage expands (i.e. non-control data). The second challenge is that their success is dependent on how well the instrumentation data (e.g. shadow stack) is protected within the same address space. Current techniques either hide the location of those through ran-

* Corresponding author.
E-mail addresses: munir.geden@cs.ox.ac.uk (M. Geden), kasper.rasmussen@cs.ox.ac.uk (K. Rasmussen).

https://doi.org/10.1016/j.cose.2023.103213

domisation or implement some access policies for them. However, integrated attacks that reveal or search the location of instrumentation data can break the promises of those schemes (Evans et al., 2015; Göktaş et al., 2016). The final and the third issue is the lack of deployability by different device types and architectures. For strong assurance, many proposals either require changes in the instruction set (ISA) (Christoulakis et al., 2016; Davi et al., 2015; Song et al., 2016) or features provided by specific architectures, e.g., Intel MPK (Burow et al., 2019), which makes them deployable only for future devices or a small portion of existing systems. Also, the majority of defences are designed for high-end devices with a reliable operating system, whereas primitive architectures and systems (e.g. bare-metal) are often ignored.

This paper presents RegGuard, a novel scheme that leverages CPU registers to protect critical program data in use with further integrity assurance even if their states are saved to the stack. Our scheme successfully addresses all three challenges mentioned above and differs from the previous proposals by providing practical and robust protection against both control- and data-oriented attacks. It is practical because RegGuard is designed as a software-only scheme that does not require any new hardware. Besides, replacing memory accesses with registers still compensates for most of the performance overhead. It is robust because CPU registers, as unaddressable storage units, provide a strong hardware root of trust for the storage of critical data in use. Thanks to our cryptographic assurance covering register data at rest on the stack, RegGuard does not need to worry about integrated attack scenarios as it does not generate any instrumentation data that must be hidden or protected in the same address space. Lastly, because RegGuard is built on one of the fundamental building blocks of computers, i.e., CPU registers, it can be adapted to different device types and architectures, including both modern and legacy systems, with trivial changes on the software stack.

To verify that the integrity checks introduced by RegGuard do not make the performance of the resulting binary unusable, we have implemented a proof-of-concept using LLVM compiler for the ARM64. Our results show that for many programs compiled, the performance cost is within a few percent of a normal optimised binary.

We summarise our contributions as follows:

1. Mitigation of attacks targeting control and data objects on the stack with an architecture-agnostic approach that relies on basic hardware primitives (i.e., registers).
2. Proposal of a security-oriented global register allocation scheme favouring program variables that are more critical to runtime integrity.
3. Leveraging CPU register file as reusable trusted storage for hosting critical data by introducing cryptographic integrity checks on saved values to the stack.

The rest of the paper is organised as follows: First, Section 2 provides the necessary background about our attack scope and register allocations, which are necessary to follow the rest of the paper. Section 3 explains our motivation, and delivers the system and threat model. Next, Section 4 presents the design of our scheme. Section 5 provides the details of our proof-of-concept implementation, whereas Section 6 provides an evaluation based on this implementation. Following Section 7 reviewing related work, Section 8 discusses certain design decisions and further extensions that can be taken forward.

2. Background

This section provides information about the attack classes mitigated and explains how conventional register allocation schemes work.

2.1. Control- and data-oriented attacks

Even if the program code is set as read-only and the stack is set as non-executable, an attacker can still exploit memory bugs in different ways. The first option is to hijack the control flow for a code-reuse attack. By carefully crafting code pointers, the attacker can express his attack using existing instructions and execute them with the order and the data he would benefit from. To achieve this, he can modify return addresses, e.g., return-oriented programming (ROP) (Checkoway et al., 2010), or indirect branch targets, e.g., jump-oriented programming (JOP) (Bletsch et al., 2011), which we describe as *control-oriented* attacks in general. Control-flow protections mitigate those scenarios by checking (Abadi et al., 2005) or ensuring (Kuznetsov et al., 2014) the integrity of control data. However, these techniques fall short of protecting against the attacks that corrupt only program variables without touching any code pointers. Such *data-oriented* attacks (Chen et al., 2005) enable the adversary to reach his goal indirectly, for instance, by altering a condition variable that decides on a privileged branch execution (i.e. control-flow bending attacks (Carlini et al., 2015)). Apart from specific scenarios, those attacks can be Turing-complete using data-oriented programming (DOP) (Hu et al., 2016; Ispoglou et al., 2018) techniques in case of a suitable vulnerability. For a DOP scenario, the attacker must exploit a bug that can compromise a loop (the dispatcher) providing necessary branches and instructions (attack gadgets).

2.2. Register allocation

Because accessing CPU registers is much faster than the memory, the compiler prefers mapping all program variables to the registers. However, there is no practical constraint on the number of variables that can be defined in a program, despite the limited number of registers (i.e. usually no more than 32 general-purpose (GPR) and 32 floating-point registers (FPR) on most architectures). Hence, a register allocation scheme must decide on how to share out registers to the variables. Thankfully, not all variables are concurrently live (i.e. code scope describing a variable definition to its final use) throughout the program execution. The compiler can thus utilise registers more efficiently by assigning the same registers to different variables (i.e. live ranges) at different times. If the number of live variables is more than available registers at any program point, called high *register pressure*, the compiler handles those situations by *spilling* some variables into the memory (Chaitin et al., 1981). The allocation scheme usually decides which variable to be spilled based on *spill costs* that estimate the number of accesses for the candidate, weighted proportionally to its loop depth (Chaitin, 1982). The compiler can also store a variable both in the memory and registers by *splitting* for better utilisation (Cooper and Taylor Simpson, 1998; Wimmer and Mössenböck, 2005).

Register allocations can happen at basic block, function or program level. If the basic block is chosen as the optimisation boundary, this is called *local* register allocation (L. P. Horwitz et al., 1966). Since local allocations have to save and restore registers at basic block sites, they are not considered as optimal as *global* allocations happening over the whole function (Chow and Hcnnessy, 1984). On the other hand, program level (interprocedural) allocations can only be meaningful for small programs with short procedures (Santhanam and Odnert, 1990). Therefore, global allocations are commonly used in practice. Global allocation enables reusing the same registers repeatedly for each function call. Depending on the calling convention in place, if a register is described as a *caller-saved* register, its state is saved/restored at call sites by caller functions. Otherwise, the function called is responsible for saving and restoring a *callee-saved* register. These opera-

tions are mostly performed through simple push-pop instructions as part of the function's prologue and epilogue.

Global schemes, which utilise registers at the function level, can adopt different approaches to solve the allocation problem. Graph colouring (Briggs et al., 1994; Chaitin et al., 1981) is the most well-known technique. It starts by building an interference graph, where the nodes represent variables and the edges connect two simultaneously live variables. The problem is formulated as two inferring (interfering) nodes (variables) cannot be coloured with the same colour (register). For a node, the degree of which is greater than the number of available colours (registers), meaning register pressure, the compiler acts based on the *spill costs*, which estimate the performance loss of leaving a variable on the memory. Alternative to graph-colouring, linear scan (Poletto and Sarkar, 1999) techniques aim for faster compilation times. Those generally maintain an active list of live variables, the intervals of which are chronologically visited to perform register allocations. Because linear techniques do not backtrack, they might result in less optimal allocations. However, proposals such as *second-chance binpacking* (Traub et al., 1998) utilise lifetime holes, i.e. a scope where the value is not needed, by placing a spilled value on a register back again.

3. Problem setting

Separation of memory into (read-only) code and (non-executable) data addresses in most systems has made it harder to perform simple code-corruption and -injection attacks. In response, more sophisticated code-reuse scenarios such as ROP have become more prevalent. Although control-flow protections (Abadi et al., 2005; Kuznetsov et al., 2014) mitigate these attacks targeting control data (i.e. code pointers), they fail to capture more challenging cases where non-control data objects (e.g. condition variables) are targeted. Addressing those attacks has proven difficult in practice as they either introduce heavy instrumentation costs (Miguel et al., 2006) provisioning each memory (data) access or require expensive hardware changes (Song et al., 2016). Furthermore, software-based approaches must secure their instrumentation data within the same address space. However, commonly used techniques such as hiding can be circumvented when the location of the data is revealed through an integrated attack (Göktaş et al., 2016). This paper takes those drawbacks into account while mitigating attacks targeting the stack.

3.1. Motivation

In order to modify a stack object, the attacker must either overflow some buffer onto the target object (i.e. relative address attack) or take over a data pointer first to overwrite it (i.e. absolute address attack). Registers are immune to both as they are not addressable.

However, to use CPU registers as a protection mechanism, we have to solve a couple of challenges. First, we must use them for security while still allowing them to serve their primary purpose, namely as a fast storage mechanism for data in use to reduce execution time. Second, we have to find a way to leverage the limited capacity of the registers to protect all the relevant variables. Simply using CPU registers as program-wide (interprocedural) storage would put a hard limit on the number of variables allocated, whereas register states that are saved to the stack during function calls void their immunity against potential corruptions. Hence, we need a global (function-level) allocation scheme that can employ the same registers for each call without undermining security. With such an integrity assurance, CPU registers can provide enough storage to secure critical control and data objects on the entire stack.

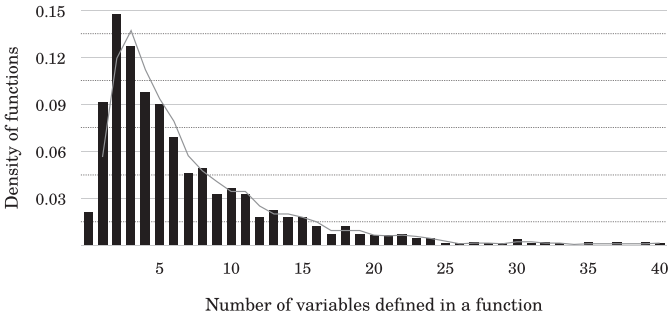

Fig. 1. Probability distributions of variable counts.

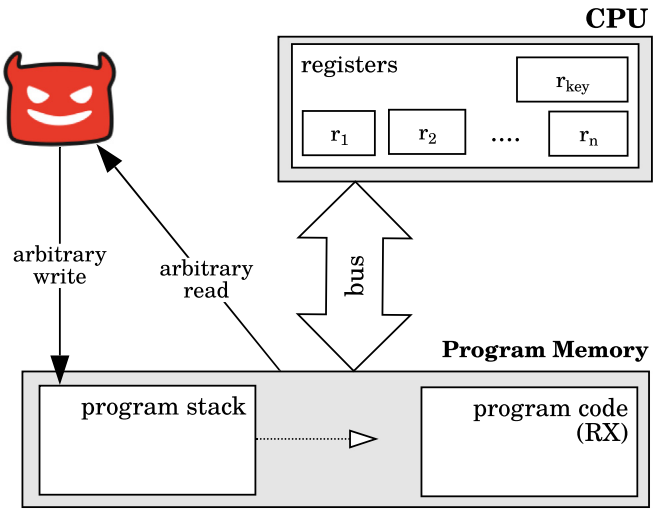

Fig. 2. Overview of the system components.

To provide insight into the coverage such protection can provide, Fig. 1 shows the distribution of variable counts found per function, which we extracted using more than a thousand functions of the benchmark programs used for performance evaluation. The results have shown that 93% of functions have less than 16 variables, and 99% have less than 32 variables. Considering the average number of variables (6.9) and arguments (2.6) found per function, most modern CPUs contain enough registers (with 16/32 GPRs and 32 FPRs) to accommodate them. Note also that these counts represent all reference and primitive variables found in a function at any point, and do not take the live ranges into account, so the number of concurrently live ones would be smaller. In Section 4.1, we show how it is still possible to deal with the rare event that this number exceeds the number of available registers.

3.2. System and adversary model

In our model, the CPU is trusted and provides limited but secure register storage. Regarding the program memory, the system (see Fig. 2) ensures code integrity via non-writable (RX) code addresses, which can be provided by some flash memory or page-level protections, depending on the setting. The CPU has n registers available (r_1-r_n) for the scheme. We dedicate a specific register (r_{key}) to store the key, for instance, a single FPR that is never saved to the program memory. We deliberately avoid making assumptions about the device type and its architecture. It can be a single-threaded bare-metal environment as well as a multi-threaded one with a rich operating system (OS), the kernel space of which is trusted by the user processes. As long as the system has the necessary CPU registers and ensures the integrity of program code,

our scheme is applicable in different settings. We only assume that software stack running on the device can be recompiled and modified as required, without asking for any changes in the hardware.

The adversary's goal is to manipulate the program runtime by modifying critical control and data objects on the stack, although program termination does not constitute a successful attack. For instance, he can target a code pointer such as a return address or a function pointer to hijack the program's control flow. Alternatively, he can overwrite a non-control data object, for example, a condition variable to manipulate the control flow indirectly. We assume a powerful adversary that has full read access to any part of the memory at all times, as well as arbitrary write access to stack addresses. We are not going to explore how such read and write capabilities can be achieved in practice, we just grant the adversary this power. We do assume that the adversary cannot intervene in the compilation process and cannot modify the program code, which includswpart of it.

This model captures both control- and data-oriented attacks exploiting the stack, including scenarios that can bypass DEP and CFI. This model also covers a wide range of scenarios on how the adversary can interact with the program memory. In contrast to protections relying on random values (e.g. stack canary) or addresses (e.g. safe stack, ASLR), this model covers integrated attacks (Göktaş et al., 2016) that exploit memory disclosure bugs first.

4. Design of RegGuard

During the compilation process from source code down to machine code, the compiler has to map variables to either memory addresses or CPU registers. Since registers are safe from memory corruption and can be accessed very fast, we would prefer to put all variables in registers. However, this might not be always possible as there can be more (simultaneously live) variables than available registers (i.e. register pressure), especially for CPU architectures suffering from register scarcity. Therefore, we must first ensure that the compiler prioritises a variable that is more likely to be attacked for registers. Second, even if all function variables are assigned to registers, their values will be saved to the memory during a function call, to make the registers available to the new function. Because these values can be overwritten on the stack, we must guarantee their integrity.

4.1. Security-oriented allocations

Similar to the conventional spill cost that estimates the performance burden of a variable left in the memory, we suggest assigning a *security score* to each variable to ensure a register is primarily allocated to a variable that is more likely to be attacked. In contrast, a security score is a compile-time estimate of how critical a function variable is for the program's runtime integrity. Variables with higher security scores are thus prioritised for register allocation and are included in the integrity checks designated for register values saved to the stack during a function call, as explained in detail in Section 4.2.

4.1.1. Security scores

RegGuard considers variables listed below as primary attack targets that must be prioritised during register allocations. It assigns a security score to each according to the given order (i.e. the first in the list has a higher score).

1. code pointers, e.g., function pointers,
2. data pointers, i.e., variable addresses,
3. programmer-defined variables, e.g., `is_admin=1`,
4. condition variables, e.g., `if(authenticated)`.

```
1  ... high register pressure...
2  int (*func_ptr)(const char*,...) = &printf;/*code pointer*/
3  int is_valid=0; /*decides on control flow*/
4  int drop_stats=0; /*no critical use*/;
5  int max_trial=read();/*user defined data*/
6  char data[SIZE]; /*buffer with untrusted environment data*/
7  /*the user has a legitimate control over the loop iterations*/
8  while (--max_trial>=0){
9      /*vulnerable function*/
10     read_buffer(data);
11     if (check(data){
12         is_valid=1;
13         break;
14     }
15     drop_stats++;
16 }
17 if (is_valid==1) /*decides on control flow*/
18     do_process(data); /*critical task*/
19 (*func_ptr)("attempts:%d", drop_stats);/*print stats*/
20 ... high register pressure...
```

Fig. 3. Code under register pressure for the given scope. For security, registers are allocated to func_ptr and is_valid first instead of less critical `max_trial` and `drop_stats`.

Pointers have the highest scores as they are the most common attack vector for powerful attacks. If not caught, the corruption of a code pointer such as an indirect jump or a call target can result in arbitrary execution, while a data pointer can be used to access or modify other memory addresses (i.e. absolute-address attack). Next comes the variables whose values are set from the code and condition variables used for branch decisions. We remind that programmer-defined variables are different from the constants evaluated and eliminated at compile time by the optimisations. A programmer-defined variable whose all possible values are hard-coded actually represents the programmer's intention. Although those are generally used as condition variables, they allocated first compared to ones defined from unknown origins. This is because an attacker would not benefit from corrupting a data object that is already controlled or defined by the user or environment (Geden and Rasmussen, 2020). Return addresses, return values, and function arguments are also assigned to registers. But they are excluded from this scoring and selection process because the calling convention in place already dedicates registers for them.

Figure 3 exemplifies how our *security scores* differ from conventional spills costs. This code depicts a high register pressure for the given scope. Normally, a conventional scheme would allocate registers to `drop_stats` or `max_trial` variables first for better execution times as they will be accessed by each loop iteration. However, RegGuard considers that `func_ptr` and `is_valid` should be given registers primarily. Alteration of `func_ptr` as a code pointer can result in illegitimate execution of sensitive system functions, whereas modifying `is_valid` flag, which is both a programmer-defined and a condition variable, would manipulate branch decisions as a data-oriented attack. On the other hand, `max_trial` which is defined externally (e.g. the user) or `drop_stats` that does not affect control-flow are not identified as critical.

In contrast to spill costs based on the use densities of variables, security scores are designed as a fast intraprocedural static approximation based on type, value agents and use purposes. Hence, a security score must be associated uniformly with different live ranges of a variable. In other words, the scores should not be localised. Algorithm 1 shows how those security scores are calculated to rank register candidates in an order that would maximise the security by taking those properties into account.

Algorithm 1 Pseudocode of security score calculations.

```
function SecurityScore(var)
    var.score ← 1
    if var.type is a pointer type then
        var.score ← var.score + 4
        if var.uselist has a branch instance then
            var.score ← var.score + 1
        end if
    else if var.type is an integral type then
        if var.deflist has an immediate assignment then
            var.score ← var.score + 2
        end if
        if var.uselist has a comparison instance then
            var.score ← var.score + 1
        end if
    end if
end function
```

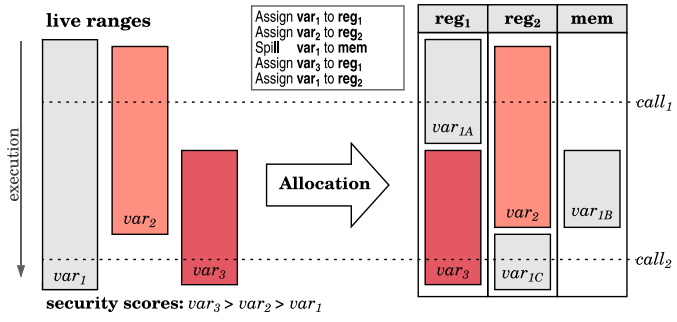

Fig. 4. Allocations under register pressure.

4.1.2. Allocation process

As a global allocation scheme, RegGuard works at the function level to reuse the same register file repeatedly for each call and to accommodate more critical data. The allocation technique to be used (see Section 2.2 for different options) should be chosen based on the features of the compiler. For instance, the compilers using single static assignment (SSA) forms (e.g., LLVM Xavier et al., 2012) generally prefer linear scan techniques (Mössenböck and Pfeiffer, 2002; Wimmer and Franz, 2010) to have faster compilation times, whereas other compilers can adopt graph-colouring (Matz, 2003). We highlight that the choice of allocation method, which some compilers can provide as a configurable option (e.g., `-fira-algorithm`), is an orthogonal issue. And it does not have any impact on the applicability of our scheme as long as conventional spill costs are replaced by the security scores proposed. Any global allocation technique provided by the compiler can thus be preferred.

We recall that registers are actually allocated to live ranges of variables. A *live range* describes the scope of the instruction or basic block scope ranging from the definition of a variable value to all its uses for the same definition. A variable can have multiple live ranges with potential gaps in between, where each starts with a new definition. The variable does not have to occupy a register during these gaps. Hence, the allocation schemes generally utilise those for more optimal allocations (Traub et al., 1998). Such utilisation can also benefit our scheme without undermining its security promises since the attacker cannot benefit from overwriting a variable value that will be later redefined before its use. The attack surface thus gets smaller as the registers are utilised better. This can be meaningful for architectures suffering from register scarcity.

Figure 4 depicts how RegGuard should allocate available registers to the variables using security scores; so decides which variables to be protected. This example considers a scope under high register pressure with two available registers reg_1 and reg_2, and three variables, the live ranges of which interfere as shown. The security scores are represented by colour tones; var_3 is the most critical target, followed by var_2, whereas var_1 has the lowest score. Using security scores, the scheme priorities two registers to var_3 and var_2 and spills var_1 when required. However, the allocation method can still utilise gaps (i.e., instructions that var_3 and var_2 do not interfere), where a register becomes temporarily available for var_1. Those splits not only enhance the performance but also provide a better reduction of the attack surface. For instance, regardless of its criticality, var_1 in the example can thus enjoy both performance and security promises, even if for a short time. And it will be safe during the execution of function calls, $call_1$, $call_2$ depicted as potential attack vectors. Suppose such a case occurs while a critical variable range is left in the memory. In that case, the compiler could display a warning message to guide the programmer to review the code.

Table 1
Variance of register saves during the callee function.

Target Type	Variance
Variables (Not Addressed)	Static
Variables (Locally Addressed)	Static
Variables (Called by Reference)	Dynamic
Temporaries	Static
Arguments	Static
Return Addresses	Static
Frame Pointers	Static
Return Values	Static

4.2. Integrity of saved register values

The program can save a register to the stack for one of two reasons. The first one is to free up a register for a more critical variable within the same function. These register spills rarely and only happen under high register pressure, and the decision of evicting a register in use for another variable is guided by the security scores described in Section 4.1. The second more common reason, which we should take care of, is a new function call that triggers the eviction of registers for the callee function. Those registers that belong to the caller's execution are saved to the stack either by the caller at call sites or by the callee as part of its prologue code. The decision of which registers must be saved/restored by the caller and the callee is mainly described by the calling convention. Regardless of the calling convention in use, any register state saved to the stack during a function call becomes vulnerable to memory corruption. Therefore, RegGuard implements integrity checks on those to ensure that they are restored back to the registers without any corruption.

4.2.1. Invariance of saved states

Integrity assurance covers saved register states that must not change during the execution of a callee function. Table 1 presents an overview of those as potential targets. The only exceptions are the values that can be legitimately modified by the callee, usually an updateable value passed as a call by reference argument. Otherwise, RegGuard protects local variables, return addresses, frame pointers, temporaries, function arguments, and return values, all of which can be targeted for an attack. With a fine-grained (e.g. flow-sensitive) pointer analysis (Hardekopf and Lin, 2011; Kuderski et al., 2019) that distinguishes local pointers from call by reference arguments, where the latter must be destroyed following the call instruction, RegGuard can ensure the integrity of locally addressed variables whose values must not change during the callee's execution.

4.2.2. Integrity checks

We recall that the data in use on registers are already safe from attacks, and the only attack surfaces left are register values saved to the stack. Therefore, RegGuard employs a cryptographic keyed hash (MAC) to guarantee that those saved register values have not been modified while they were at rest on the stack. Prior to the execution of a function body, our scheme computes a reference tag from register objects being saved to the stack. This tag value is also kept on a specific register unless the callee function makes another call. Upon completion of the function body, a new tag is generated from actual objects being restored to the registers. This tag is compared to the reference tag previously generated from saved objects, any corruptions on those can thus be revealed. For a function call consisting of both caller- and callee-saved registers, this is a two-step process connected. The first tag generation/verification of caller-saved registers is managed at the tails of call sites, while the following tag digesting callee-saved registers is created/checked as part of function prologues/epilogues.

Function-wise, RegGuard digests each call frame using a single tag value. Program-wide, because we save the tag register to the stack with other registers and include it in the next tag calculation, our scheme actually creates a chain of tags that provides (almost) a complete stack image on a single register. Although we still rely on the key for integrity checks, this chain prevents the attacker from replaying a (standalone) call frame and its corresponding tag for a different call context. Thanks to the control over the compilation process of the software stack, we remind that the key register is never saved to the same program/process memory, which is adequate to authenticate any tag restored from the memory that serves as the integrity proof of restored objects. With a single key kept secret on a dedicated register and MAC calculations that are part of non-writable program code, RegGuard enables the use of register file as an integrity-guaranteed storage for each function call.

Figure 5 depicts an overview of a call stack tied with tags. RegGuard creates a tag for each callee- and caller-saved region, where the tag of a caller-saved region also contains the previous tag of a callee-saved region or vice versa. This helps us to bind frames to each other with a tight representation of the whole program stack. Equations (1) and (2) below express what each tag created for caller- and callee-saved regions contains.

$$\text{tag}_i = \text{MAC}_{\text{sk}}(\text{tag}_{i-1} \parallel \text{arg}1_{i-1} \parallel \ldots \parallel \text{tmp}n_{i-1}) \tag{1}$$

$$\text{tag}_{i+1} = \text{MAC}_{\text{sk}}(\text{tag}_i \parallel \text{ret}_i \parallel \text{bp}_i \parallel \text{var}1_i \parallel \ldots \parallel \text{var}n_i) \tag{2}$$

Although the details can vary depending on the calling convention and the architecture, we consider that the caller is responsible for saving and restoring its arguments (*arg*) and temporaries (*tmp*) at call sites while its return address (*ret*), base/frame pointer (*bp*) and local variables (*var*) on registers are saved by the callee function. Even if the architecture (e.g. x86) does not use a link (return) register and creates return addresses directly on the stack, because the return addresses are static and not used during the function body unlike other objects, they are also included in the tag generated for callee-saved regions.

To reveal the corruption of a saved object, RegGuard injects two groups of instructions. The first group generates a reference tag for register values being saved at function prologues and call sites. The second group checks whether this reference tag matches with the one calculated from restored values. Both tag calculations directly align with the existing register operations to avoid any additional memory accesses. With a few scratch registers, RegGuard can compute tags from directly register values. In order to make this possible, the compiler should rearrange register restores in the same order they are pushed, instead of pop instructions working in the reverse order.

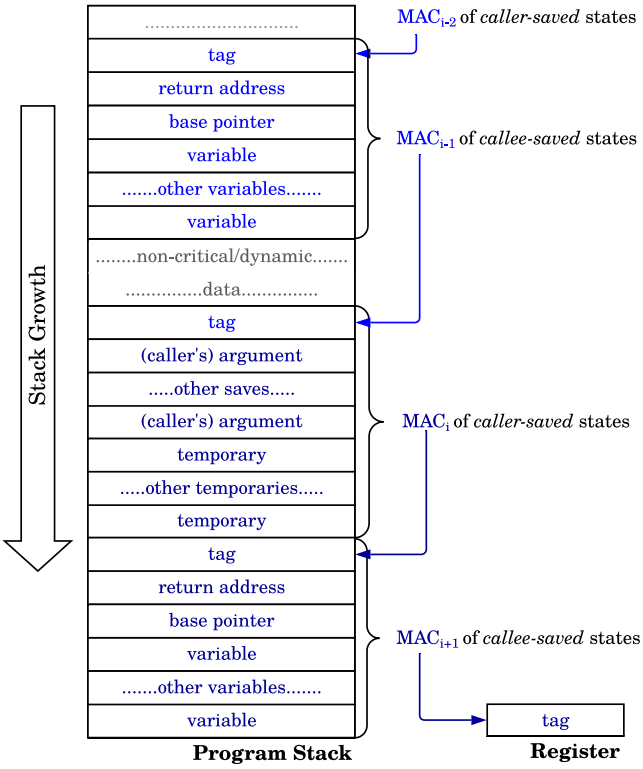

Fig. 5. Securing saved register objects using a keyed hash.

4.2.3. Bootstrapping and key management

Regarding the bootstrapping of the system, the tag generation starts with the first call made by the software in question. For a simple setting with no process or privilege separation, such as a bare-metal or a RTOS environment, a single key to be shared by all tasks is generated at boot time using software or hardware RNGs available on the system. This key is assigned to an FPR dedicated as the key register. We note that this register is not saved to the memory by the scheduler or interrupt handler, thanks to the control over the software stack. If there is a hardware context switching in use, those instances also usually do not save FPRs. Otherwise, in the case of a general-purpose OS, a fresh key is generated at each process start. Only the kernel space can host the key, which is trusted by the user processes. User-managed threads share the same key and do not save the key register during a context switch.

4.2.4. Choice of MAC

The MAC function to be used should be chosen based on available features of the CPU architecture. If the ISA provides relevant vector and cryptographic extensions, we recommend using HMAC-SHA256 (Hansen, 2011) with hardware acceleration. Otherwise, we suggest using SipHash (Aumasson and Bernstein, 2012) as an architecture-agnostic option for CPUs that lack cryptographic extensions. SipHash is a keyed hash primarily designed to be fast, even for short inputs, with a performance that can compete with non-cryptographic functions used by hash tables. Thanks to its performance benefits, SipHash is highly practical and deployable on different architectures.

Figure 6 sketches how RegGuard can align its MAC calculations with register operations at function prologues and epilogues using SipHash. Both sections start by initialising internal states (on scratch registers) generated from the key and constants. Next, it applies compression rounds on those with message blocks (values)

```
read_buffer:                          #callee function
    .....prologue: register saves.....
    INIT(key)                         #initialise states (v1-4) with r_key
    store   r_tag,    [sp]
    COMPRESS(m1)                      #m1=r_tag
    store   r_ret,    [sp-8]
    COMPRESS(m1,m2)                   #m2=r_ret
    store   r_bp,     [sp-16]
    COMPRESS(m2,m3)                   #m3=r_bp
    store   r_var1,   [sp-24]
    COMPRESS(m3,m4)                   #m4=r_var1
    store   r_var2,   [sp-32]
    COMPRESS(m4,m5)                   #m5=r_var2
    r_tag = FINALIZE(m5)
    sub     sp,     40
    .....body instructions.....
    .....epilogue: register restores.....
    mov     r_tmp1, r_tag             #copy tag to a scratch register
    INIT(key)                         #initialise states (v1-4) with r_key
    load    r_tag,    [sp+32]
    COMPRESS(m1)                      #m1=r_tag
    load    r_ret,    [sp+24]
    COMPRESS(m1,m2)                   #m2=r_ret
    load    r_bp,     [sp+16]
    COMPRESS(m2,m3)                   #m3=r_bp
    load    r_var1,   [sp+8]
    COMPRESS(m3,m4)                   #m4=r_var1
    load    r_var2,   [sp]
    COMPRESS(m4,m5)                   #m5=r_var2
    r_tmp2 = FINALIZE(m6)
    CHECK(r_tmp1,r_tmp2)              #check whether tags match
    add     sp,     40
    ret
example():                            #code in Figure 2
    mov     r_var1, &printf           #int (*func_ptr)...; line 2
    mov     r_var2, 0                 #int is_valid=0; line 3
    .....instructions.....
    call    read_buffer
    .....instructions.....
    mov     r_var2, 1                 #is_valid=1; line 12
    .....instructions.....
    cmp     r_var2, 1                 #if (is_valid==1) line 17
    .....instructions.....
    call    r_var1                    #(*func_ptr)(...); line 19
```

Fig. 6. MAC calculations aligned with register operations for the slice of func_ptr and is_valid variables.

already on registers. Lastly, it completes tag generation with the final message block (register). The reference tag is not pushed to the stack unless the function calls another function. Prior to the epilogue, this reference value is moved to a scratch register; the epilogue can thus restore the previous tag to the dedicated register as a part of the restoring process. The reference tag moved to a temporary register will be later compared against the actual tag generated from restored registers at the end before return. Any mismatch of two tags implies an attack because saved register objects cannot be changed unless the control is returned back to the caller function.

4.2.5. Attack coverage

ROP attacks that exploit return addresses are prevented by RegGuard, regardless of whether the architecture has a link register or not as in x86. In contrast to other variable objects, return addresses are always static and must have a single definition (call) and single use (return) located at our instrumentation sites, so they are always included in the MACs and protected. Further JOP scenarios that corrupt other code pointers on the stack given by function pointers and switch statements are also mitigated as those are either securely updated (e.g. pointer arithmetic) within the CPU or checked against any corruptions before they are restored back from the stack. Thanks to the integrity guarantees on data pointers, absolute-address (non-linear) attacks that can use them to access/corrupt other memory sections are also avoided. In addition, RegGuard mitigates relative-address (linear) attacks such that a stack array is overflown onto an adjacent condition variable as a DOP attack. We exclude scenarios that might alter composite data values, such as strings for practicality. However, those strings typically host untrusted inputs and their corruption can be only meaningful as a data-only attack in case the given string has a critical use in bulk following a sanitisation check, with a timely bug located between the sanitation and critical use. Otherwise, the sanitisation (comparison) outcome of those inputs that affects the control flow would be already transferred to a condition variable that will be safe on a register (i.e., control-flow bending attacks).

4.3. Security analysis

As previously described, the adversary's goal is to manipulate the program execution by corrupting the control and data objects on the stack. For the corruption to stay undetected, the adversary has to either skip the integrity checks or make those checks pass. We will look at each of these options in turn.

In order to skip checks, the adversary must modify the binary or its execution to void the instrumentation. The former is not possible in our model because the code segment is non-writable. The latter which requires altering code pointers is also infeasible as the scheme protects those in the first place. For the adversary to pass integrity checks, he has to forge a valid tag or reuse a previously recorded one. Forging a valid keyed hash for an attack state either requires finding the second preimage of the legitimate state or access to the key. Since the key is protected on a register that is never saved to the same address space (including user-managed context switches and setjmp/longjmp instances), and therefore unavailable to the attacker, if the MAC-function is secure (i.e. existentially unforgeable, and second preimage resistant), forging a valid tag without the key is only possible with negligible probability. We remind the reader that our system model assumes that the software code executing within the same process/program space, including user libraries are recompiled, or can be touched through binary-level mechanisms. This is to guarantee that no instruction operates on the register (i.e., FPR) reserved for the key, except bootstrapping code responsible for key generation and placement. This protects the key even under the presence of a powerful attacker that has arbitrary access primitives to the same program memory. In the case of a multi-threaded environment, the key register is allowed to be managed (i.e., context switches) by only trusted software components, such as the kernel.

The adversary might attempt to replay a seen tag for a different call of the same or a different function. However, even with the same variable and argument values, replaying will not work. This is because each tag containing return address, base pointer and more importantly former tags (representing previous call frames) provides a very tight representation of the whole stack, where the (most recent) tag digesting all context is also safe on a register. Besides, replaying a tag for a different process in rich OS environments or a different execution time in the embedded systems is not an option since a fresh key is generated at each process or device start.

Table 2
The details of calling convention used.

Register	Type	Purpose
x0-x7	Caller-saved	Arguments
x9-x15	Caller/e-saved	Temporaries
x19	Callee-saved	Tag
x20-28	Callee-saved	Local variables
x29	Callee-saved	Frame/base pointers
x30	Callee-saved	Link/return addresses
q31 (FPR)	Reserved/not saved	MAC key

5. Implementation on ARM64

We have implemented a proof-of-concept[1] of RegGuard on ARM64 (AArch64) to mainly evaluate its performance impact. RegGuard can be adapted to different architectures such as x86, SPARC, MIPS, and RISC-V. We have chosen ARM64 for demonstration purposes. In addition to being the dominant architecture of the mobile and embedded landscape with a rapidly increasing market share in the PC domain, ARM64 provides enough registers to secure more variables than expected to be found per function (i.e., 10 callee-saved GPRs (x19-28) compared to 6.9 variables on average) without having to modify the standard calling convention (ABI) of the underlying software components. Furthermore, the registers reserved for arguments (x0-x7) and temporaries (x8-x15) can help not only to secure other potential data targets but also to avoid register pressure in general. If needed, FPRs can be used for the same purpose. More importantly, the ISA equipped with cryptographic extensions allows us to evaluate the hardware-accelerated SHA256 option.

For the implementation, we have used the LLVM compiler, which is configured to dedicate a single FPR (128-bit) for the key and a GPR (64-bit) for tag values. We first modified the basic register allocation pass provided as a custom linear allocation technique that use priority queues. Since our benchmark programs have not encountered register pressure, our modified pass simply ensures that registers are not spilled for performance reasons. Then, we have mainly worked on the components responsible for the generation of prologue and epilogue code. For the proof-of-concept, integrity checks are placed for only callee-saved registers that are primarily assigned to local variables by the allocator. But the registers known as caller-saved can also be included in tag calculations using the same instrumentation, thanks to the compilation flags available (e.g. `-fcall-saved-x9`). Table 2 summarises the highlights of the calling convention used during our experiments.

For simplicity, we have encapsulated hash calculations with two functions added to the C library (musl).[2] The first one (`_register_mac`) is injected to the end of the prologue and generates a reference tag from saved register values. The second one (`_register_check`), which is placed at the beginning of the epilogue, creates another tag from the values to be restored and compares it against the reference value. In the case of unmatched values, which means an attack, it terminates the program. Both wrapper functions take the starting address and the size of the region where registers are pushed as their arguments. The latter function additionally requires the reference tag for comparison. The instrumentation also handles the preservation of original arguments required by the actual callee function and the return values upon its completion at call sites of the wrapper functions. For optimisation purposes, we have avoided injecting these two functions into the leaf functions, as their frames cannot be modified in practice without having another function call. Differently from the ideal design proposed in Section 4.2, those wrapper functions calculate MACs from the register values awaiting on the stack instead of directly using values already on registers. We remind the reader that as a proof-of-concept implementation avoiding the complexity, these functions introduce additional memory and cache accesses. Hence, our performance discussion should be seen as an over-approximation, whereas the implementation based on the proposed design would have less performance overhead.

For MAC, we have explored two keyed hash options. The first one is SHA256, backed by hardware acceleration. The second one is SipHash-2-4, as a fast, practical, and deployable option for different architectures lacking vector and cryptographic extensions.

[1] https://github.com/msgeden/llvm-project
[2] https://github.com/msgeden/musl

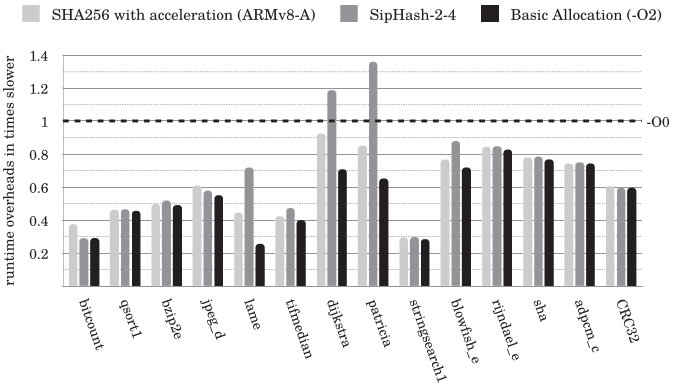

Fig. 7. Runtime overheads of RegGuard.

6. Evaluation

This section evaluates the performance overhead of our proof-of-concept implementation and presents some real-world vulnerabilities it mitigates.

6.1. Performance

For performance evaluation, we have used *cBench* (Fursin, 2009), a popular open-source uniprocessor benchmark suite that is based on earlier MiBench (Guthaus et al., 2001) suite. The experiments were performed with a collection of 14 C programs from various categories that aim a realistic benchmarking and research. We have run those programs on a Linux system running on an Apple M1 chip that is equipped with the ISA features we need, such as SHA extensions. We have run benchmark programs with both non-instrumented and instrumented versions of the C library (i.e. musl libc). The latter version aims to provide a better understanding of the costs for extended guarantees in the case that the *libc* library is not considered as part of TCB and can be exploited to corrupt the program's objects. We have experimented with both SHA256 (using ISA acceleration) and SipHash-2-4 for integrity checks.

For program-only instrumentation (see Fig. 7), SHA extensions have produced only *13%* runtime overhead, whereas SipHash has yielded *23%* overhead, compared to the programs compiled with basic register allocation without any instrumentation (`-O2`). In contrast to unoptimised versions, where no register allocation takes place, both MAC options have yielded better execution times. We have observed higher performance costs for programs linked to an instrumented C library as expected (see Fig. 8). Compared to the basic allocations bundled with `-O2` optimisations, SHA256 and SipHash instrumentation have introduced *33%* and *59%* runtime overheads, respectively. Considering the binary sizes, instrumented C library with wrapper functions is only *14%* higher than the non-instrumented library file.

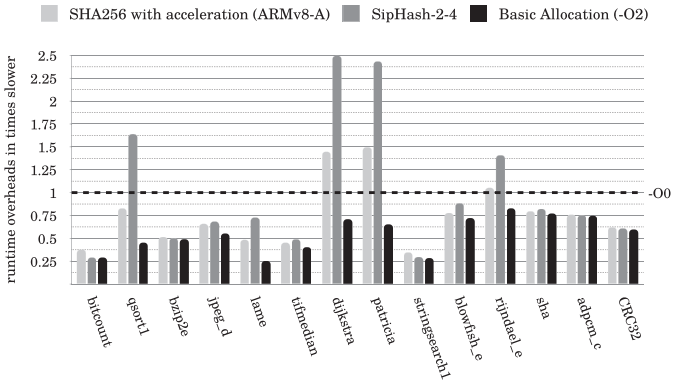

Fig. 8. Runtime overheads with libc instrumentation.

We have used `-O2` as the baseline to evaluate the overhead costs of additional integrity checks. On the other hand, comparisons with unoptimised and non-instrumented programs aim to evaluate the performance compensation by the register allocations. We note that there are other optimisations included contributing to the performance compensation. For instance, inlining some functions not only avoids branching costs but also reduces tag calculations. This is due to the fact that the caller can aggregate register operations of the inlined function. Overall, SipHash, with its reasonable overheads, proves to be a practical option for different CPU architectures without asking for any hardware change or acceleration. If available, using native SHA instructions that provide around *7x* speed-up would be a faster and more convenient option. Depending on the CPU features, both options can thus be practically used to ensure the integrity of register data on the stack since the overheads are within very small fractions of optimised times (`-O2`) for most programs.

6.2. Real-world cases

We have tested our PoC implementation using buffer overflow cases extracted from open source model programs (e.g. BIND, Sendmail, WU-FTP) made available as a SARD test suite (88) by NIST. For a sound evaluation, we have first attempted to compile available 14 cases with clang and `-O2` flag instead of the default gcc and `-O0` configurations given by the suite. Due to the optimisations changing the memory layouts and architectural differences, solely 6 cases have remained compiled and exploitable. From those, our implementation has successfully captured 5 out of 6 cases (1285/CVE-1999-0368, 1287/CVE-1999-0878, 1289/CA-2001-01, 1299/CVE-1999-0131, 1303/CVE-1999-0047). Only one case (1307/CVE-2001-0653) exploiting a sign cast bug to underflow a global array with negative index values has been undetected. Although such scenarios are not within the scope of this work, Section 8 discusses how MAC checks can be extended to mitigate similar corruptions.

7. Related work

This section reviews relevant work previously proposed and discusses how RegGuard differs from them.

7.1. Software-based mitigations

There have been different proposals to mitigate control-oriented attacks. Control-flow integrity (CFI) techniques (Abadi et al., 2005; Davi et al., 2012; Niu and Tan, 2014) validate code pointer addresses according the control-flow graph (CFG) and do not bother with how the corruption occurs. Although a shadow stack can assist for a fully precise backward-edge CFI (i.e. return addresses), forward-edge targets can only be approximated depending on what is decidable and computable at compile-time. In contrast, code-pointer integrity/separation (CPI/CPS) (Kuznetsov et al., 2014) focuses on the integrity of code pointers instead and provides more precise protection. This approach requires a safe stack, the location of which is generally randomised within the same memory space without special hardware support. However, integrated attacks that can disclose the location of the stack easily circumvent made promises (Evans et al., 2015). RegGuard does not need to worry about those attack scenarios as it does not rely on isolation or hiding of any data on the memory.

Control-flow protections are generally prone to attacks that do not touch any code pointers (i.e. DOP) and still lack a practical mitigation deployed in the wild. Miguel et al. (2006) have proposed data-flow integrity (DFI) protection against those attacks. DFI checks whether any data object used at runtime is defined by an expected instruction given by flow-sensitive static reaching definitions analysis. DFI requires excessive instrumentation of almost every memory access to protect both program and instrumentation data. A more coarse-grained technique with better performance in return for the loss of precision, write integrity testing (WIT) (Akritidis et al., 2008) instruments only write instructions to prevent them from modifying objects that are not in the set of flow-insensitive points-to analysis. Two relevant studies PointGuard (Cowan et al., 2003) and data space randomisation (DSR) (Bhatkar and Sekar, 2008) mask data objects with random values and unmask them prior to their use. The main goal is to make corrupted values useless for an attacker that does not know masking values. Although masking pointers can harden to find a useful target address, branch decisions relying on boolean or value range comparisons would not have much protection by masking. Differently, RegGuard detects the corruption of critical data objects under stronger adversary assumptions (e.g. memory disclosure), regardless of whether they are useful or not to the attacker.

7.2. Hardware-assisted protections

Regardless of their coverage, software based techniques must first ensure the integrity of their instrumentation data (e.g. shadow stack). But this is a challenging task without special hardware assistance. Hardware-assisted schemes can provide better performance and protection against both control (Christoulakis et al., 2016; Davi et al., 2015) and data (Nyman et al., 2019; Song et al., 2016) attacks. However, those academic proposals are not usually adopted in practice as they require changes in CPU architectures, and the manufacturers do not implement them due to various reasons. Furthermore, already available features provided specific CPUs to protect instrumentation data, such as Intel MPX and MPK, are shown to have high instrumentation or switch overheads despite their strong security promises (Burow et al., 2019). In contrast, RegGuard promises the same strength of assurance as an instrumentation-only solution using very basic hardware primitives that are available in any CPU. This makes our scheme applicable to both legacy and modern architectures for a broad spectrum of devices, from high-end processors to low-end embedded systems.

7.3. Cryptographic approaches

MACs are first used by CCFI (Mashtizadeh et al., 2015) to mitigate control attacks on x64 architectures. A CBC-MAC is computed and placed alongside each control object on the memory. But instead of leveraging registers for practical protection of control data in use, CCFI uses floating-point registers to only store

AES keys that occupy most of them (i.e. 11 out of 16 XMM registers). The authors benefit from Intel's AES-NI extensions to speed up MAC calculations. A similar work (Liljestrand et al., 2019) use new pointer authentication (PAC) features provided by ARMv8.3-A. PAC tags are calculated from effective bits (39-bit) of pointer addresses and squeezed into the (unused) upper part (24-bit) of the address word, which makes them susceptible to brute-force scenarios due to the short tag size. PAC associates return addresses with the stack pointer to harden replay (pointer substitute) attacks. PAC does not provide any mechanism to detect corruption of a primitive variable, for instance, a condition variable overflown by an adjacent buffer. Similar to CCFI, PAC authenticates pointers in a standalone way with a separate MAC tag for each, in contrast to our work that digests many control and data objects using a single tag. Furthermore, both idea is only applicable to specific CPU models. More recent schemes, ZipperStack (Li et al., 2020) and PACStack (Liljestrand et al., 2021) (as an implementation of a similar approach using PAC) create a chain of tags to protect return addresses. These schemes protect only return addresses and does not cover other (forward-edge) control or data targets. Similar to PAC, ZipperStack stores MACs on the upper (24-bit) space of word, which provides weaker protection. Apart from their limited coverage, none of those cryptographic solutions leverages the security and performance features of CPU registers as means for protecting critical objects in use.

Palit et al. (2019) suggest encrypting sensitive data resident in the memory to address memory disclosure attacks. The proposed work ensures that the sensitive data is always kept encrypted in the memory and is decrypted only while being loaded into CPU registers, for confidentiality. It leverages AES extensions and XMM registers available in the x86 ISAs. A very recent work (Fanti et al., 2022), which also cites the early version of this study, similarly suggests the protection of register spills via cryptography, but using an architecture-specific primitive i.e., ARM64's PAC instructions. Differently, the proposed work does not provide a mechanism to favour critical variables during register allocations. Another recent work RegVault (Xu et al., 2022) suggests the protection of kernel data through value masking, by adopting a similar approach to our work. The scope of the work is confidentiality and integrity assurance of register- and interrupt-based context data in the kernel space. Similar to PAC, RegVault uses an FPGA-based acceleration of the QARMA scheme implemented for RISC-V.

8. Discussion

In this section, we present a discussion of certain design decisions of RegGuard, including further extensions and future CPU design features that would complement our scheme.

8.1. Chained vs. independent frames

Given that RegGuard uses a keyed hash, it is not a strict requirement to include the previous tag in the tag of the next frame. In other words, we could have chosen to independently secure each call frame, rather than chaining them together. This section will briefly look at the reasons for and against this design desertion. For a program with a regular call stack strictly following LIFO, we could have relied solely on a single (unkeyed) hash for the stack *integrity* by chaining frames. This is because such a program can ensure that any CPU state restored from the stack complies with the hash register first. However, there are many legitimate cases where the register hosting the head of the chain has to be saved to/restored from program memory without our instrumentation, for example, setjmp/longjmp, exception handling and threading managed within the same address space. They all oblige us to rely on the MAC key instead of a single hash.

Despite its redundancy for integrity assurance, we have chosen the chained approach over independent frames to prevent *replay attack* scenarios. With independent frames, the attacker could simply replay a call frame (and its aligned tag) for a different function call or context. However, with a chained approach, replaying for a different call context will not work since the tag register provides a very tight representation of the execution context, including all functions calls waiting to be returned.

8.2. Primitive devices and register scarcity

RegGuard uses security scores to distinguish critical variables and prioritise them for available registers under register pressure scenarios. However, it is difficult to observe such cases in a modern CPU setting that provides a register file consisting of 48–64 (16/32 GPRs and 32 FPRs) registers with sizes up to 2 kB. Hence, our selection process actually serves more primitive architectures suffering from register scarcity (e.g. 6–8 GPRs with no FPRs). In such a case, our security scores aim to accommodate at least critical stack objects in registers. But if there is a critical object still left in the memory, the compiler could display a warning; so the programmer can review the code. Despite being ignored by some compilers, the programmer can use the `register` keyword in C to annotate which variables to protect. We have designed RegGuard as an architecture-agnostic solution to make it applicable to a wide range of systems, even with the most resource-constrained devices in mind; for example, a 16/32-bit MCU with no security at all, but might be still prevalent in critical systems. By just relying on a flash program memory and a few registers, we can reduce the attack surface significantly against less strong adversaries (i.e. weaker checksums).

8.3. Future CPU architectures

Although RegGuard is designed to fit existing CPU architectures, we would like to see CPU manufacturers incorporate some of these ideas into their designs in the future. If the next generation of CPUs were to include the necessary registers and maybe even hardware acceleration of a suitable MAC function, RegGuard could be implemented at the hardware level through a single instruction. A bit vector-like operand can be given to describe which registers to include in the MAC, and the new instruction can then run all the necessary calculations within the CPU. Such instruction would enable us to create a standalone tag for registers spills happening within the function, without having to worry about the performance overheads.

Furthermore, similar to discontinued Itanium (IA-64) architecture with 128 GPRs and 128 FPRs, CPU manufacturers can consider expanding their register files as trusted storage and adopting *register windows* to zero out the performance costs in return for space overhead within the CPU. Register windows, which are designed to avoid the cost of register spills on each call by making only a portion of registers visible to the program, can actually benefit our scheme more than its original purpose by eliminating cryptographic calculations. For example, with a window size of 32 (from 128 registers), RegGuard would not incur any overheads for a program that has no call down deeper than four calls, where some of these could also be reserved for protection of global variables.

8.4. Further extensions

RegGuard covers attack scenarios that target stack objects. Due to the integrity assurance of the pointers on the stack, most illegitimate accesses to other memory sections would also be mitigated. However, the attacker might still have options not covered here, such as overflowing a global array or a heap buffer to target

an adjacent variable whose modification can result in a successful attack. But in the case of hardware acceleration for MAC, we can extend our scheme to address those. For example, we can allocate a tag address next to each global variable or composite data that will host a digest of them. We can update this tag at each legitimate (re)definition of those variables and verify when used.

9. Conclusion

This paper presents RegGuard, a novel and practical scheme that leverages CPU registers to mitigate control- and data-oriented attacks targeting stack objects, for instance, return addresses, function pointers and condition variables. Our protection relies on the immunity of registers from memory corruptions as unaddressable storage units. Despite their heavy use by compiler optimisations, general-purpose registers have not been systemically used as secure storage because of their limited capacity and voided immunity of saved values on the stack.

RegGuard addresses these with a two-step proposal. First, during register allocations, it prioritises program variables that are expected to be targeted; so they stay safe while in use. Second, when those registers are saved to the stack because of a function call, we compute a keyed hash to ensure they are restored without any corruption. Those integrity checks enable the reuse of the same register file as secure storage repeatedly for each function call, without having to occupy registers across function boundaries.

Although RegGuard is designed as a software-based approach to be practical, it makes strong promises using a very basic hardware primitive, i.e., CPU registers. This makes our scheme applicable to a broad range of devices from high-end to low-end without any special hardware features. Our experiments on ARM64 have shown that registers can also enhance program security with a surplus within the range of 13% (with SHA extensions) to 23% (SipHash) on average compared to purely performance-based optimisations. Therefore, RegGuard provides a practical protection designed upon on very basic building blocks of computers, such as code integrity, registers and MAC calculations that can be expressed by any CPU ISA.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Munir Geden: Conceptualization, Methodology, Software, Validation, Visualization, Writing – original draft. **Kasper Rasmussen:** Supervision, Project administration, Writing – review & editing.

Data availability

PoC is given as a github link in the paper.

Acknowledgement

We thank the Ministry of National Education of the Republic of Turkey because of their generous support for Munir Geden's doctoral studies during the preparation of this paper.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.cose.2023.103213.

References

Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J., 2005. Control-flow integrity. In: Proceedings of the 12th ACM conference on Computer and communications security - CCS '05. ACM Press, New York, New York, USA, p. 340. doi:10.1145/1102120.1102165.

Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M., 2008. Preventing memory error exploits with WIT. In: 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, Oakland, CA, USA, pp. 263–277. doi:10.1109/SP.2008.30. http://ieeexplore.ieee.org/document/4531158/

Aumasson, J.-P., Bernstein, D.J., 2012. SipHash: a fast short-input PRF. In: 13th International Conference on Cryptology in India (INDOCRYPT 2012), Vol. 7668 LNCS. Springer Berlin Heidelberg, Kolkata, India, pp. 489–508. doi:10.1007/978-3-642-34931-7_28.

Bhatkar, S., Sekar, R., 2008. Data space randomization. In: 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). Springer, Paris, France, pp. 1–22.

Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z., 2011. jump-oriented programming : a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11. ACM Press, New York, New York, USA, p. 30. doi:10.1145/1966913.1966919.

Briggs, P., Cooper, K.D., Torczon, L., 1994. Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. 16 (3), 428–455. doi:10.1145/177492.177575.

Burow, N., Zhang, X., Payer, M., 2019. SoK: shining light on shadow stacks. In: Proceedings - IEEE Symposium on Security and Privacy. IEEE, pp. 985–999. doi:10.1109/SP.2019.00076.

Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R., 2015. Control-flow bending: on the effectiveness of control-flow integrity. In: USENIX Security Symposium. USENIX Association, Washington, D.C, USA, pp. 161–176.

Chaitin, G.J., 1982. Register allocation and spilling via graph coloring. ACM SIGPLAN Not. 17 (6), 98–101. doi:10.1145/872726.806984.

Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W., 1981. Register allocation via coloring. Comput. Lang. 6 (1), 47–57. doi:10.1016/0096-0551(81)90048-5.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., Winandy, M., 2010. Return-oriented programming without returns. In: Proceedings of the 17th ACM conference on Computer and communications security - CCS '10. ACM Press, New York, New York, USA, p. 559. doi:10.1145/1866307.1866370.

Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K., 2005. Non-control-data attacks are realistic threats. USENIX Security Symposium. USENIX Association, Baltimore, MD.

Chow, F., Hcnnessy, J., 1984. Register allocation by priority-based coloring. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1984, 19, pp. 222–232. doi:10.1145/502874.502896.

Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S., 2016. HCFI: hardware-enforced control-flow integrity. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 38–49. doi:10.1145/2857705.2857722.

Cooper, K.D., Taylor Simpson, L., 1998. Live range splitting in a graph coloring register allocator. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 1383, pp. 174–187. doi:10.1007/BFb0026430.

Cowan, C., Beattie, S., Johansen, J., Wagle, P., 2003. PointGuardTM: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium. USENIX Association, Washington, D.C., pp. 91–104.

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Grier, A., Wagle, P., Zhang, Q., Attacks, B.-o., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., 1998. StackGuard : automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium. USENIX Association, San Antonio, TX, pp. 63–78.

Davi, L., Dmitrienko, A., Egele, M., Thomas, F., Holz, T., Hund, R., Nurnberger, S., Sadeghi, A.-R., 2012. MoCFI: a framework to mitigate control-flow attacks on smartphones. In: NDSS 2012 (19th Network and Distributed System Security Symposium), Vol. 26, pp. 27–40.

Davi, L., Hanreich, M., Paul, D., Sadeghi, A.-r., Koeberl, P., Sullivan, D., Arias, O., Jin, Y., 2015. HAFIX: hardware-assisted flow integrity extension. In: Proceedings of the 52nd Annual Design Automation Conference. ACM, New York, NY, USA, pp. 1–6. doi:10.1145/2744769.2744847.

Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., Okhravi, H., 2015. Missing the point(er): on the effectiveness of code pointer integrity. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp. 781–796. doi:10.1109/SP.2015.53. https://ieeexplore.ieee.org/document/7163060/

Fanti, A., Chinea Perez, C., Denis-Courmont, R., Roascio, G., Ekberg, J.E., 2022. Toward register spilling security using LLVM and ARM pointer authentication. IEEE Trans. Computer-Aided Des. Integr. Circuits Syst. 41 (11), 3757–3766. doi:10.1109/TCAD.2022.3197511.

Fursin, G. Collective benchmark (cBench): collection of open-source programs and multiple datasets from the community. https://www.sourceforge.net/projects/cbenchmark/files/cBench/V1.1/

Geden, M., Rasmussen, K., 2020. TRUVIN: lightweight detection of data oriented attacks through trusted value integrity. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, Guangzhou, China, pp. 174–181. doi:10.1109/TrustCom50675.2020.00035. https://www.ieeexplore.ieee.org/document/9343134/

Göktaş, E., Economopoulos, A., Gawlik, R., Kollenda, B., Athanasopoulos, E., Portokalidis, G., Giuffrida, C., Bos, H., 2016. Bypassing clang's SafeStack for fun and profit. https://www.blackhat.com/docs/eu-16/materials/eu-16-Goktas-Bypassing-Clangs-SafeStack.pdf.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R., 2001. MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). IEEE, Austin, TX, USA, pp. 3–14. doi:10.1109/WWC.2001.990739. http://www.ieeexplore.ieee.org/document/990739/

Hansen, T., 2011. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). Technical Report. RFC Editor. https://www.rfc-editor.org/rfc/rfc6234.txt

Hardekopf, B., Lin, C., 2011. Flow-sensitive pointer analysis for millions of lines of code. In: International Symposium on Code Generation and Optimization (CGO 2011). IEEE, pp. 289–298. doi:10.1109/CGO.2011.5764696. http://ieeexplore.ieee.org/document/5764696/

Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z., 2016. Data-oriented programming: on the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, San Jose, CA, USA, pp. 969–986. doi:10.1109/SP.2016.62.

Ispoglou, K.K., AlBassam, B., Jaeger, T., Payer, M., 2018. Block oriented programming: automating data-only attacks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 1868–1882. doi:10.1145/3243734.3243739.

Kuderski, J., Navas, J.A., Gurfinkel, A., 2019. Unification-based pointer analysis without oversharing. In: Proceedings of the 19th Conference on Formal Methods in Computer-Aided Design, FMCAD 2019, pp. 37–45. http://arxiv.org/abs/1906.01706

Kuznetsov, V., Szekeres, L., Payer, M., 2014. Code-pointer integrity. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, Broomfield, CO, pp. 147–163.

L. P. Horwitz, Karp, M.R., Miller, R.E., Winograd, S., 1966. Index register allocation. J. Assoc. Comput. Mach. 13 (1), 43–61.

Li, J., Chen, L., Xu, Q., Tian, L., Shi, G., Chen, K., Meng, D., 2020. Zipper stack: shadow stacks without shadow. In: European Symposium on Research in Computer Security. Springer, Guildford, UK, pp. 338–358. doi:10.1007/978-3-030-58951-6_17.

Liljestrand, H., Nyman, T., Gunn, L.J., Ekberg, J.E., Asokan, N., 2021. PACStack: an authenticated call stack. In: Proceedings of the 30th USENIX Security Symposium, pp. 357–374.

Liljestrand, H., Perez, C.C., Nyman, T., Ekberg, J.E., Wang, K., Asokan, N., 2019. PAC it up: towards pointer integrity using ARM pointer authentication. In: Proceedings of the 28th USENIX Security Symposium. USENIX Association, Santa Clara, CA, USA, pp. 177–194.

Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D., 2015. CCFI: cryptographically enforced control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 941–951. doi:10.1145/2810103.2813676.

Matz, M., 2003. Design and implementation of a graph coloring register allocator for GCC. In: GCC Developers Summit, pp. 151–170. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.9896&rep=rep1&type=pdf-page=151

Miguel, C., Costa, M., Harris, T., 2006. Securing software by enforcing data-flow integrity. In: Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, pp. 147–160.

Mössenböck, H., Pfeiffer, M., 2002. Linear scan register allocation in the context of SSA form and register constraints. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2304, pp. 229–246. doi:10.1007/3-540-45937-5_17/COVER. https://www.link.springer.com/chapter/10.1007/3-540-45937-5_17

Niu, B., Tan, G., 2014. Modular control-flow integrity. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 577–587. doi:10.1145/2594291.2594295.

Nyman, T., Dessouky, G., Zeitouni, S., Lehikoinen, A., Paverd, A., Asokan, N., Sadeghi, A.-R., 2019. HardScope: hardening embedded systems against data-oriented attacks. In: 2019 56th ACM/IEEE Design Automation Conference (DAC). IEEE, Las Vegas, NV, USA, pp. 1–6. doi:10.1145/3316781.3317836.

Palit, T., Monrose, F., Polychronakis, M., 2019. Mitigating data leakage by protecting memory-resident sensitive data. In: Proceedings of the 35th Annual Computer Security Applications Conference. ACM, New York, NY, USA, pp. 598–611. doi:10.1145/3359789.3359815.

Poletto, M., Sarkar, V., 1999. Linear scan register allocation. ACM Trans. Program. Lang. Syst. 21 (5), 895–913. doi:10.1145/330249.330250.

Santhanam, V., Odnert, D., 1990. Register allocation across procedure and module boundaries. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation - PLDI '90. ACM Press, New York, New York, USA, pp. 28–39. doi:10.1145/93542.93551.

Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., Paek, Y., 2016. HDFI: hardware-assisted data-flow isolation. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, San Jose, CA, USA, pp. 1–17. doi:10.1109/SP.2016.9. http://www.ieeexplore.ieee.org/document/7546472/

Traub, O., Holloway, G., Smith, M.D., 1998. Quality and speed in linear-scan register allocation. SIGPLAN Not. (ACM Special Interest Group on Programming Languages) 33 (5), 142–151.

Wimmer, C., Franz, M., 2010. Linear scan register allocation on SSA form. In: Proceedings of the 8th annual IEEE/ ACM international symposium on Code generation and optimization - CGO '10. ACM Press, New York, New York, USA, p. 170. doi:10.1145/1772954.1772979. http://www.portal.acm.org/citation.cfm?doid=1772954.1772979

Wimmer, C., Mössenböck, H., 2005. Optimized interval splitting in a linear scan register allocator. In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments - VEE '05. ACM Press, New York, New York, USA, p. 132. doi:10.1145/1064979.1064998. http://portal.acm.org/citation.cfm?doid=1064979.1064998

de Souza Xavier, T.C., Oliveira, G.S., de Lima, E.D., de Silva, A.F., 2012. A detailed analysis of the LLVM's register allocators. In: 2012 31st International Conference of the Chilean Computer Science Society. IEEE, pp. 190–198. doi:10.1109/SCCC.2012.29. http://ieeexplore.ieee.org/document/6694089/

Xu, J., Lin, H., Yuan, Z., Shen, W., Zhou, Y., Chang, R., Wu, L., Ren, K., 2022. RegVault: hardware-assisted selective data randomization for operating system kernels. In: Proceedings of the 59th ACM/IEEE Design Automation Conference. ACM, New York, NY, USA, pp. 715–720. doi:10.1145/3489517.3530549. https://dl.acm.org/doi/10.1145/3489517.3530549

Munir Geden is a D.Phil. student in Cyber Security at the University of Oxford. He completed his masters degree in Software Systems Engineering from University College London, in 2015, with a dissertation on malware analysis. His doctoral research explores runtime verification of software programs in different contexts. His main areas of interest are software security, malware analysis, compilers and trusted computing.

Kasper Rasmussen received his Ph.D. degree under the guidance of Prof. S. Capkun at the Department of Computer Science, ETH Zurich, where he worked on security issues related to secure time synchronization and localization, with a particular focus on distance bounding. He was a Post-Doctoral Fellow with the University of California, Irvine, for two years. In 2013, he joined the Department of Computer Science, University of Oxford, where he is a Full Professor. He was awarded a University Research Fellowship from the Royal Society of London in 2015. His thesis won the "ETH Medal" for outstanding dissertation from the Swiss Federal Institute of Technology, and he was additionally awarded the Swiss National Science Foundation Fellowship for prospective researchers.