Towards Register Spilling Security using LLVM and ARM Pointer Authentication

Towards Register Spilling Security using LLVM and ARM Pointer Authentication

Andrea Fanti*, Carlos Chinea Perez†, Remi Denis-Courmont†, Gianluca Roascio*‡, Jan-Erik Ekberg†
*Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy
†Huawei Technologies Oy Co. Ltd, Helsinki, Finland
‡ CINI Cybersecurity National Laboratory, Italy
andrea.fanti@studenti.polito.it, {carlos.chinea.perez, remi.denis.courmont, jan.erik.ekberg}@huawei.com
gianluca.roascio@polito.it

***Abstract*—Modern RISC processors are based on a load/store architecture, where all computations are performed on register operands. Compilers therefore allocate registers based on demand, and when occupancy is at maximum, register contents are spilled onto the stack and then retrieved later as data is needed. This phenomenon has security implications that cannot be ignored, as data on the stack is subject to well-known memory corruption attacks. Moreover, works presented so far are mainly targeting protection of pointers to code (e.g., return addresses), but are ineffective for protecting other context data in the stack.**

This paper presents a security solution for spilled registers, generalizing the use of ARM Pointer Authentication for this purpose. The protection is enforced by the LLVM compiler via additional compiler passes and modifications. The solution provides guarantees for both integrity and confidentiality protection, and also addressing reuse attack problems associated with PA usage. Experimental data collected demonstrates the effectiveness of the solution against corruption and eavesdropping. We test our solution using SPEC CPU 2017, which confirms the functional viability of our solution. Additionally, we expose real-world performance overhead metrics of our protection design on a ARM-PA enabled processor.

***Index Terms*—security, software security, memory safety, control-flow integrity, compiler, register spill, pointer authentication**

I. Introduction

When the source code of applications is compiled to binary representation, optimizing register allocation for performance is one of the most algorithmically complex tasks undertaken by the compiler. Computer registers are a limited resource, despite that many contemporary hardware architectures are stocked with plenty of general-purpose registers (e.g. ARM Aarch64 has 31). During program execution in a stored-program computer, and especially in *Reduced Instruction Set Computers* (RISC), registers serve as an on-chip buffer within which all arithmetic computation is carried out. Having the register bank not residing in memory is an important security distinction. In academic research, the presence and the power of memory attacks, aptly summarized in a paper of 2013 by Szekeres *et al.* [1], has received lots of attention in the last decade. A common denominator for all of this work is the insight that data in memory is insecure, and memory-based attacks can even be mounted within the executing code itself, without modifying it. A range of different memory-based attacks (such as Return-Oriented [2] and Data-Oriented Programming [3]) have been proposed in this setting. The literature is rich of software-based mitigations, such as canary protectors [4] or address randomization [5]. Different *Control-Flow Integrity* (CFI) [6] mitigations have been very visible in the academic research field in the last decade. Also, defenses based on hardware extensions such as stack replicas [7] or memory tagging [8] has been adopted by commercial chip vendors. This is the case of the ARM Pointer Authentication extension [9], introduced from the 8.3 version of the ISA, and used by Apple MacOS since 2019.

In most programming languages, there is no contract between programmer and compiler regarding which data stays "protected" in registers and which data can be stored in memory. This is likely due to the fact that languages largely predate the whole issue of memory protection. However, system languages like C often carry a keyword like `register` which, although not absolute, can be used by the programmer to suggest to the compiler his intent about the storage of a variable. New system languages like Rust promote a "safe" mode where static analysis guarantees the memory safety of the resulting program, but this happens in the abstract, on the language level, before any compiler optimization takes place.

In compiler operation, data movement between registers and memory is dictated by the hardware-specific *Application Binary Interface* (ABI), as well as the register allocation procedure. Depending on the code being compiled, the compiler may face *register pressure*, i.e., the size of the register bank is too small to hold all the needed values required for arithmetic at a given point in the executable code. When this happens, the compiler must resort to a process called *register spilling*, i.e., register contents are temporarily "spilled" to memory to retrieve space for values more urgently needed for computation at that specific location in the code. Spilled register values are later read back as they are needed.

To the authors' best knowledge, memory safety issues related to register spilling have been very much overlooked in the research community. A recent article by Panigrahi *et al.* [10] proves in formal terms both the insecurity of the register allocation process, assessing it in the field on the LLVM

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

compiler. They conclude that the security properties of the source code is not preserved downstream of the compilation process, and the reason is due precisely to register spilling, which taints possibly sensitive data. Also, register spilling easily affects security techniques negatively: for example, the research done in [11] by Huang *et al.* infers that stack canary reference values in volume end up in memory due to spilling, thereby defeating the protection effect of a canary.

This work focuses on the detrimental interdependence between register spilling and memory security, and proposes two hardware-assisted solutions to mitigate this issue. The work is also timely: as new hardware protections such as memory tagging or pointer authentication become available in *Commercial, off-the shelf* (CoTS) processors, the spilling security problem is exacerbated by the fact that memory protection designs leveraging these mechanisms may wrongly assume that register values are never spilled: e.g., relying on tag bits in pointers to remain immutable or that return pointers in leaf functions never leave the registers in which they are stored.

The present paper is written with the following intentions and contributions:

1) Sharing the insight that the issue of register spilling in compilers is a present and future security concern, especially in relation to memory safety;
2) Providing an analysis of how often (and in which context) register spilling occurs in a "typical" program, and discuss what vulnerabilities may follow from such an end result;
3) Providing an LLVM compiler solution using ARM Pointer Authentication, where compiler-spilled register data can be protected both for integrity and confidentiality;
4) Providing performance analysis for the application of this method, both when applied selectively and with full coverage.

The remainder of the paper is organized as follows. In Section II, background elements and data are provided on the register allocation and spilling processes, on the LLVM compiler, and on the ARM Pointer Authentication facilities. In Section III, the adversary model and the requirements that our solution aims to satisfy are presented. In Sections IV and V, details on how the protection is designed and implemented are offered. Section VI provides a qualitative and quantitative assessment about the security of our technique and its performance, measured on standard benchmarks taken from the SPEC CPU 2017 suite. Section VII is intended to place our work within the state of the art regarding related solutions. Finally, Section VIII concludes the paper.

II. Background

A. Register Allocation and Spilling

In compilers, *register allocation* is a complex optimization problem that can have a big impact on speed. In contemporary *Reduced Instruction Set Computers* (RISC), all arithmetic is done using registers. As memory loads and stores are very slow compared to arithmetic operations or intra-register moves, it is paramount to maximise register use. On the other hand, assigning variables and data contents in an optimal way to registers when a computer program is compiled into machine code is an NP-complete problem.

In programs, *register spilling* happens when register contents is temporarily *spilled* to memory to retrieve space in the register file. Spilling instructions are inserted by the compiler — the activity of spilling is not represented on the computer program level, and is not visible to the programmer. There are two general classes of register spilling: one occurs because of contract, i.e., spilling is defined to take place in a given way based on the agreed *Application Binary Interface* (ABI) for the underlying hardware. The second one is a consequence of register allocation, as sometimes the compiler needs to temporarily recover registers for computation when all available registers already carry content. This is referred to as the compiler having *register pressure*.

In ARMv8-A ABI, examples of the former spilling type happens in function calls with more than eight parameters, as this is the number of registers hosting parameters, while by contract further ones are passed via the stack, i.e., spilled. Register assignment within the function may also leverage local reuse of so-called *Callee-Saved Registers* (CSR) in a function, i.e., registers that by ABI maintain their value across function calls. However, if the compiler decides to reuse CSRs within the called function, it spills and restores their values at its convenience (Figure 1).

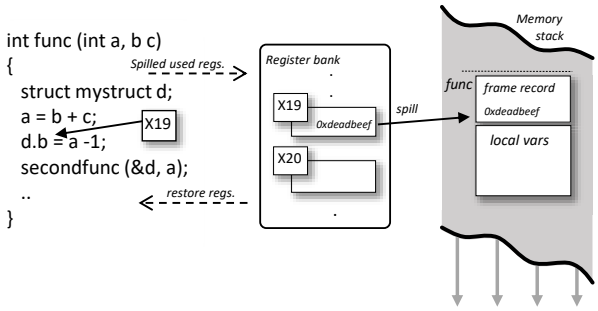


Fig. 1: CSR spills: the compiler assigns register `x19` to local computation. ABI dictates `x19` to be a CSR, so its contents is spilled to stack in the function prologue, and restored from memory in epilogue.

As part of register allocation, the compiler has many strategies by which register pressure can be released. Values of registers with a dedicated purpose (e.g., function arguments) are often swapped to/from other general-purpose registers. If a register is populated with a value that can be reconstructed based on other register contents and fixed values, it can be discarded and later *rematerialized* — this is often faster than caching its contents to memory. Only as a last resort, the compiler spills register contents to (and later restores them from) memory, such as to be able to execute some particularly complex calculation in highly nested loops.

Fig. 2: The 3-step LLVM compilation process.

B. The LLVM Compiler Infrastructure and interference graphs

LLVM[1] is a compilation framework designed for providing target-independent and language-independent optimization techniques that can be used through different compilation scenario as in a library fashion. LLVM defines an *Internal Representation* (IR), which can be seen as a low-level language with a high-level type system.

The LLVM process is outlined in Figure 2. The source code is first translated into IR by a language-specific *front-end*. Then, the IR goes through a set of optimization processes, all operating on the IR. Finally, in the compiler *back-end*, the code is translated into a final form which can be easily assembled to the machine code of the target architecture.

Within the IR, to ease optimization passes and trace information flows, each variable is assigned once and once only. This is called *Static Single Assignment* (SSA). For LLVM, register allocation is the step in which each IR variable is mapped onto a physical location. These may be physical registers, or *spill slots* may be allocated on the function stack frame. The Greedy Register Allocation algorithm is adopted, which follows the *graph coloring* concept [12]. 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 variables, while the available colors represent the available locations. The *edges* E of the graph represent interference between variables: when the liveness range of two of them crosses, ad edge connects these two nodes. G is then also called the *interference graph*.

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 (*coalescence of nodes*). 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.

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

[1]https://llvm.org

C. Register Spilling Case Study - Google Chromium

As a reference to illustrate the prevalence of register spilling in contemporary programs, we instrumented LLVM compiler version 12 with extra debug output, and added a tooling harness to report on different variants of register spilling in the program. We applied this toolset to an ARMAv8 compilation of the Google Chromium browser, since its commercial sibling (Google Chrome) is the default browser in Android mobile phones on 64-bit ARM hardware. With this argument, Chrome is one of the most used single applications in the ARM ecosystem, and also otherwise, Chrome accounts for over 60% of all web traffic in the world [13] independently of platform.

Our findings show register spilling occurring at a large scale. The Chromium baseline includes around 1 million functions, and when compiled, registers are spilled at 3.5 million code locations where the called function needs to use some registers that the ABI assumes it to maintain during function calls. Further, due to local register pressure in functions, around 100,000 additional memory locations (spill slots) are allocated in functions, and 100,000 register stores as well as 200,000 register reloads target these locations. Needless to say, register spilling is not a statistically rare event on the ARM platform. Another investigation on the prevalence of spills in binary code by Salgado *et al.* [14] counts the assembly overhead of register spilling in ARM Thumb code (a 16-bit embedded architecture variant), obtained after compilation of some embedded benchmarks by MiBench, which ends up in being average 12% for their tests.

D. ARM Pointer Authentication

The ARMv8.3-A (AArch64) hardware architecture support a feature for register integrity, called Pointer Authentication (PA) [9]. The idea behind this feature is that the 64-bit address space is unused in most of the AArch64 deployments, as the amount of addressable bytes with 64 bits is certainly out of needs for actual applications. This leaves the topmost bits of all pointers unused, and the intended use of PA is to repurpose these bits to store a cryptographic message authentication code (MAC). The MAC is from 16 to 32-bit long depending on the configuration, and calculated in hardware over the pointer value. When reloading this pointer from memory after having stored it, evidence of tampering can be easily discovered by checking the pointer signature. Signature creation and verification are performed using a hardware implementation of the QARMA algorithm [15], and are triggered by the explicit use of variants of two new instructions, `pac` and `aut`, respectively. The authentication code results in being computed as:

```
PAC = QARMA(key, pointer, context)
```

where `context` is a parameter chosen in relation to the execution context, for increasing uniqueness of the signature.

ARM PA also stocks a general purpose MAC primitive — the instruction variant `pacga Xd, Xn, Xm` which computes PAC for a value stored in register `Xn`, using a modifier stored in `Xm`, and a key, and stores the result in the upper 32 bits of `Xd`. In this work, we primarily leverage this PA instruction.

One advantage of using PA, e.g., instead of architecture-provided AES/SHA acceleration instructions, is that registers holding PA keys can be shielded in hardware from the privilege level using the PA instructions. Thereby key material theft is excluded as an attack vector. In our design, new keys are generated for a process by the Linux kernel as part of the `exec()` system call invocation [16].

Today, ARM PA technology is widely deployed in COTS processors like the Apple A12 (and later) cores, or provided as part of cloud service infrastructure using Amazon Gravitron 3 ARM cores.

III. Adversary model and Requirements

This work follows what is customary in memory protection literature, and assumes a powerful memory adversary that can achieve both memory reading and corruption via an attack. Additionally, it is assumed that the computer code, i.e., the `.text` segments, can be rendered immutable by means of common memory management features, such as W $\oplus$ X. Thus, we do not consider attacks that modify the code of the running application. Furthermore, in this work, we focus singularly on the protection of register spills, in the understanding that complementary compiler-assisted security mechanisms exist for, e.g., call-flow or data protection (see Section VII). In this respect, this work can form a basis for, e.g., a confidentiality solution - where some data is never written to memory in the clear - or an integrity solution, where memory modification of state variables or counters is unconditionally detected, or it can be used as a mechanism that corrects potential security oversights in earlier compiler-focused memory protection work, where the presence of register spilling as a threat has not been fully accounted for.

Against this background, we focus our security requirements explicitly on register spilling, i.e.:

1) The integrity of any value or reference stored in a register shall be guaranteed when it is temporarily emitted to memory due to compilation needs;
2) When so required, the confidentiality of data stored in a register shall be maintained when spilled to memory due to compiler-internal operation;
3) The protection of register spills shall only minimally interfere with or detract from any other memory security mechanism it is associated with.

As mentioned, we focus only on register spilling as a problem, although, in the absence of an associated memory protection scheme on the language or compiler level, theoretically our implementation could be applied for all data as well, at a huge performance cost. A similar argument can be had for applying confidentiality to spills – most spills will relate to program execution and call flow, and as such need only integrity protection. However, we cannot exclude that some spills may require confidentiality, say in the case of a key being input to an encryption function. So we implement and measure both features, and leave it for future work to decide what protection is needed in practice.

IV. Design

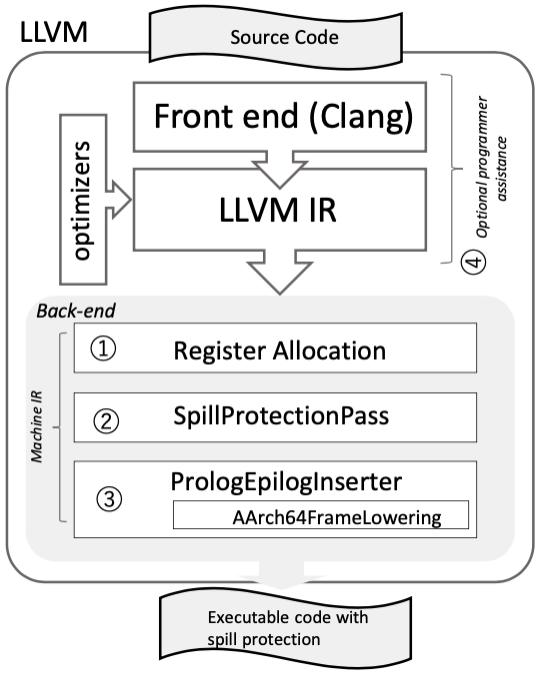


Fig. 3: Overview of compiler modifications.

Our instrumentation for securing register spills is implemented in the LLVM-12 compiler suite in the way illustrated in Figure 3. The instrumentation itself does not touch the front-end of the compiler, making the solution compatible with any programming language supported by LLVM. In the compiler back-end, the *Register Allocators* (1) releases register pressure by conditionally assigning variables to registers in the code. To handle these, our instrumentation adds a pseudo-instruction to the events for later processing. In our new *SpillProtection* (2) compiler pass, the compiler translates the pseudo-instruction to secure register spill machine IR code, adding integrity protection, or both integrity and confidentiality protection, as described below. CSR spills are fully handled in the *PrologEpilogInserter* pass (3), where they are written to our machine IR with the same properties as for the spills caused by register pressure. We also needed to extend the architecture-dependent *AArch64FrameLowering* module for CSR protection.

The solution requires the presence of ARM PA support to take advantage of some of the additional instructions there introduced. ARM PA is available, e.g., using the `armv8.3a` architecture selection.In this paper, we present our solution in

the context of *application / user-space* protection, where the PA keys are managed and only accessible by the OS kernel, and therefore we assume the kernel to remain uncompromized. However, with small modifications we can extend our design to any privilege level (including the kernel), and the PA keys can in the kernel context be protected e.g., by using MMU memory protection as outlined in [17].

Our solution shall not temporarily store any intermediate computations related to spill protection to memory, as that would trivially violate our security targets. We have chosen to reserve two registers, currently `x14` and `x15`, as temporary storage. Having two registers reserved for this purpose is sufficient for both integrity and confidentiality protection (as it is described in Section V), but removing two general-purpose registers from general compiler use implicitly leads to increased register pressure and more spills compared to the case when these registers are available. The resulting performance penalty is baked into the measurements in Section VI.

By using the temporary registers, we can take care that intermediary values are not passed through memory. E.g., for integrity, we simply use `pacga` as a message authentication code (MAC) function and compute:

$$\textit{Authenticate:} \quad reg_v \xrightarrow{PAC} \texttt{x15}\ ,\ \{reg_v, \texttt{x15}\} \rightarrow mem$$
$$\textit{Validate:} \quad mem \rightarrow \{reg_v, \texttt{x15}\ \}\ ,\ reg_v \xrightarrow{PAC} \texttt{x14},\ \texttt{x14 = x15}\ ?$$

whereas for confidentiality or combined protection, the register and PAC usage patterns are more complex and optimized — this is detailed in Section V.

As pointed out in several prior art, one problem of using PAC as part of a stack construct — both for integrity and as a keystream — is that a simple PAC modifier (like `sp` register) may repeat during the lifetime of the program (stack), and therefore the danger of replay attacks is present [18]. In such an attack, protected stack values are harvested, and later replaced in memory when the top of the stack reaches the same address. Our scheme is vulnerable to this attack. We next plan to implement the principle presented in Camouflage [17], where both the stack pointer and the program counter `pc` are used as a modifier — at the cost of one extra arithmetic instruction per spill. This will narrow the replay opportunity to values spilled in the same function at the same stack address at an earlier time. Even with this configuration, one could still envision, e.g., spilled loop counters within a function being replayable in this paradigm. An ultimate solution is to reserve yet an extra register for protection and deploy the principle of PACStack [19], which maintains a continuously changing (and protected) modifier for PA, but implementing this approach is being left for future work based on need.

V. Implementation

Our implementation provides protection against both spills introduced due to register pressure and spills required by the ABI, i.e., spilling CSRs. We control our protection solution using compiler flags. The self-descriptive flags `aarch64-enable-spill-protection`, `aarch64-integrity-protect-csr`, `aarch64-enable-spill-encryption` and `aarch64-encrypt-csr` configure the integrity and confidentiality individually for CSR and other spills, respectively.

```
// Mv  := original spill slot stack offset
// Mm  := MAC spill slot
// xv  := register whose value to spill

    // Storing registers
    str     xv, [sp, #Mv]
    pacga   x15, xv, sp
    lsr     x15, x15, #32
    str     w15, [sp, #Mm]
    ...
    // Loading registers
    ldr     xv, [sp, #Mv]
    ldr     x15, [sp, #Mm]
    pacga   x14, v, sp
    eor     x14, x14, x15
    cbnz    x14, .Lfail
```

Listing 1: Integrity protection of a register spilled due to pressure.

Even though these flags enable protection of spills selectively depending on the spilling cause (register pressure, CSR use in the function), the protection is applied for all spills in the program. While this is sufficient for this work, where we want to exercise our compiler solution as widely as possible and analyze its overall cost, there is certainly room for further improvement. For instance, confidentiality protection is likely best applied only for data indicated by the programmer. If static safety analysis is used (such as LLVM SafeStack pass) also integrity protection for spills can be applied selectively. These features we reserve for future work – the programmer assisted approach, where a language extension is provided for selective application of spill protection is illustrated by (4) in Figure 3.

The integrity protection of in-function spills is actuated by generating a distinct MAC for each spill using `pacga` with the Stack Pointer `sp` as modifier and storing the result alongside the spilled register in the stack frame. When variable size objects are present in the function, the Base Pointer `bp` (if available) or the Frame Pointer `fp` are used as modifier instead of `sp`. Pseudocode generated for spill integrity in case of register pressure is presented in Listing 1. The function `.Lfail` causes the program to fail in case the reloading and validation of a spilled register and its MAC causes an integrity error.

It is easy to visually verify from Listing 1 that all computation takes place among registers, both on stores and reloads. The same requirement is later fulfilled in the more complex protection patterns.

When encryption is activated, the register is masked with pseudo-random keystream generated with `pacga`, i.e., we essentially operate a register-based stream cipher for spill protection. This is done with only the two registers we reserve for the solution, and the pattern is observable in Listing 2. In the current implementation, the "keystream" generation is seeded with a number computed by the compiler hashing

```
// Mv  := original spill slot stack offset
// Mm  := MAC spill slot
// xv  := register whose value to spill
// d   := nonce to use, d = hash(function || Mm)

    // Storing registers
    mov     x15, #d
    pacga   x15, x15, sp
    eor     x14, xv, x15    // encrypt left word
    pacga   x15, x15, sp
    eor     x14, x14, x15, lsr #32
                            // encrypt right word
    str     x14, [sp, #Mv]  // store encrypted value
    pacga   x15, x14, sp    // generate MAC
    lsr     x15, x15, #32
    str     w15, [sp, #Mm]  // store MAC
    ...
    // Loading registers
    ldr     xv, [sp, #Mv]
    pacga   x15, xv, sp     // regenerate MAC
    ldr     w14, [sp, #Mm]  // reload MAC
    lsl     x14, x14, #32
    eor     x15, x14, x15   // compare MACs
    cbnz    x15, .Lfail     // fail if corrupted
    mov     x15, #d
    pacga   x15, x15, sp
    eor     xv, xv, x15     // decrypt left word
    pacga   x15, x15, sp
    eor     xv, xv, x15, lsr #32
                            // decrypt right word
```

Listing 2: Confidentiality and integrity protection used together. We apply integrity protection on the encrypted content.

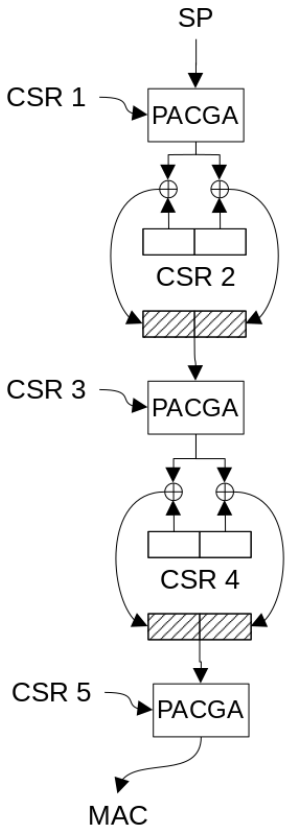


Fig. 4: CSR MAC generation algorithm for 5 registers.

together the function signature (function name, return type, and parameters name and type) and the MAC spill index within in the local function stack space to ensure different spills in the same function will have different keystream. When encryption and integrity protection are both activated, the former is performed before the latter, in an Encrypt-then-Authenticate (EtA) fashion.

With the objective of minimizing memory usage and speed up calculation, MAC calculation for the CSRs is performed in a more specialized way (also leveraging on the fact that CSRs are all spilled and reloaded all at once). The algorithm is represented visually in Figure 4 (in the case of 5 CSRs).

Finally, when we integrity protect callee-saved registers we can leverage on the fact that the compiler will for each laid-out function know which registers need to be spilled, and they are all spilled in one go into the function frame in the function prologue and later recovered in a batch at the end of the function, in the epilogue. Thus we deploy a chained MAC for CSRs by grouping together CSRs in pairs from the second CSR onwards as is depicted in Figure 4. The algorithm is as follows:

1) the intermediary MAC (e.g., of the 1st CSR, or 1st, 2nd and 3rd CSRs, or CSR 1 to 5, etc.) is expanded (i.e repeated twice) to 64 bits;
2) the expanded intermediary MAC is XORed with the next CSR to generate a modifier for the next `pacga`;
3) the new intermediary MAC is computed using `pacga` with the next CSR (e.g., the 3rd) and the modifier computed in step 2.

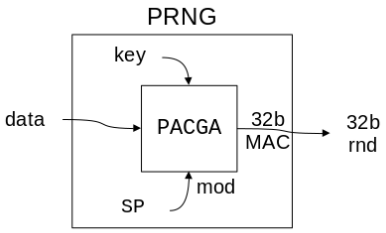
We continue with steps 1 to 3 until there is either 1 or 0 CSR left to protect. In the case where 1 CSR still remains, a final `pacga` is applied with the intermediary MAC as modifier. Otherwise the last computed intermediary MAC is considered to be final one. Where encryption is applied to protect CSRs it is performed in the same way as for individual spills shown in Listing 2, with the only difference that a single keystream is applied to cover all the encrypt CSRs in one go. The way this is done is visualized in Figure 5, and like the other protection schemes, this can still be implemented strictly using registers only.

VI. Evaluation

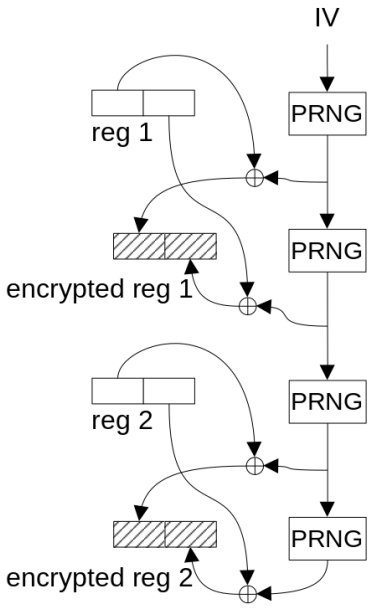
In this Section, we evaluate our solution from the perspective of functionality, performance and security, and argue for that our solution does meet requirements set forth in Section III.

A. Security Analysis

Our security guarantees focus on register spilling only, in the understanding that this feature complements some existing memory protection feature for call-flow or data protection, or is potentially used in a programmer-assisted way. All register spilling taking place in the compiled code is instrumented by our mechanisms. For integrity, we use the full 32-bit MAC



(a) Construction of the PRNG function.



(b) Encryption algorithm.

Fig. 5: Encryption of 2 CSR registers.

```
// x19, x29, x30 := CSRs used in the function

    // Function prologue, storing CSRs
    pacga   x15, x19, sp // compute temporary MAC
    eor     x15, x15, x15, lsr #32
    eor     x15, x29, x15 // mask 1st CSR with MAC
    pacga   x15, x30, x15 // compute overall MAC
    str     x15, [sp, #-32]! // store MAC, update SP
    stp     x29, x30, [sp, #8] // store other CSRs
    str     x19, [sp, #24]
    ...
    // Function epilogue, reloading CSRs
    ldr     x19, [sp, #24]
    ldp     x29, x30, [sp, #8]
    ldr     x15, [sp], #32 // reload MAC, restore SP
    pacga   x14, x19, sp   // recompute MAC in x14
    eor     x14, x14, x14, lsr #32
    eor     x14, x29, x14
    pacga   x14, x30, x14
    eor     x14, x14, x15  // compare MACs
    cbnz    x14, .Lfail    // fail if corrupted
    ret
```

Listing 3: Integrity protection of CSRs. Since these spills are clustered both when produced and recovered, we can apply a speed-optimized process for their protection (see Figure 4).

provided by PA, so brute-force attacks are not viable. We protect against replay attacks by instrumenting the modifier to be both location and function-dependent, and point to a mechanism by which modifier replays can be provably circumvented [19] if needed. The PA keys are set by the kernel at a higher privilege level, and not attackable by the running code. All MAC computation and validation is done within the register bank, and insecure memory is never used during this process. With these arguments, we ensure Requirement 1. The chained MAC generation for CSRs in Figure 4 is a speed optimization – it is also done within the register bank and ultimately just includes more input to the generated MAC.

The spill encryption feature is constructed as a stream cipher, where the keystream is generated from a cryptographically secure pseudorandom number generator (CSRPNG), built using PAC as the cryptograhically secure hash function. For this feature to fully protect spills, the MAC seeding needs to be unconditionally unique, as can be achieved, e.g., using [19]. However, it is not known whether MACs are always suitable for CSPRNG construction [20], and furthermore the PAC algorithm is allowed to be manufacturer-specific, i.e., it can vary between ARM processors. This cryptographic validation is left for further work, but based on the rest of the arguments listed for integrity protection, we can conclude that Requirement 2 is also satisfied.

In terms of secure design, the spilling protection only relies on the ability to generate non-repeating seeds for the integrity and confidentiality protection – beyond this, the design is fully self-sufficient in terms of stack layout and operation. It does not impose any requirements on the PA key, as long as its secrecy is maintained. With these arguments, we stipulate that this mechanism is easy to integrate with most other memory protection solutions on ARM-A versions 8 or 9, thereby satisfying Requirement 3.

For a more pragmatic security evaluation, we also run relevant parts of the Juliet test suite v.1.3 [21] on C/C++ code instrumented with our spilling protection. Juliet is a large collection of vulnerable programs with associated tests that exploit known vulnerabilitites in them, based on *Common Weakness Enumeration* (CWE) by Mitre. These tests have been used, e.g., to evaluate the efficiency of scanning tools [22]. However, since a portion of Juliet tests represent real-world vulnerabilities in the stack, it is expected that some of these attacks will be mitigated by our solution. The Juliet suite includes 67 stack-based buffer overflow tests. All of them run successfully with our instrumentation when no attack is mounted (functional test, false-positive check). Our spilling protection catches 23 of the associated attack tests. We consider this to be a good result, as our system is not intended to provide memory safety only by itself. However, this testing indirectly confirms the prevalence of spilling as discussed in Section II-C, and our protection efficiency is largely explained by the fact that our CSR protection will end up serving as an authenticated stack canary, akin to the methods presented in [23].

TABLE I: Spill protection binary size overhead measured on SPEC CPU 2017 benchmarks.

Benchmark	Binary size (baseline)	Binary size (with integrity)	Overhead	Binary size (with encryption)	Overhead
perlbench	2860392 B	3204456 B	12.03%	3851624 B	34.65%
gcc	9816144 B	11360336 B	15.73%	14506064 B	47.78%
mcf	616888 B	629176 B	1.99%	641464 B	3.98%
omnetpp	2006968 B	2420672 B	20.61%	3362752 B	67.55%
xalancbmk	4521936 B	5316560 B	17.57%	7098320 B	56.98%
x264	1166640 B	1269040 B	8.78%	1424688 B	22.12%
deepsjeng	92440 B	104736 B	13.30%	125216 B	35.46%
leela	164952 B	189536 B	14.90%	246880 B	49.67%
xz	702768 B	731440 B	4.08%	788784 B	12.24%

TABLE II: Spill protection execution time overhead measured on SPEC CPU 2017 benchmarks.

Benchmark	Baseline	With integrity	Overhead	With encryption	Overhead
perlbench	62.39 s	125.96 s	101.89%	397.83 s	537.64%
gcc	31.02 s	46.89 s	51.18%	120.64 s	288.92%
mcf	231.70 s	242.06 s	4.47%	306.44 s	32.26%
omnetpp	267.62 s	305.57 s	14.18%	477.70 s	78.50%
xalancbmk	209.34 s	230.60 s	10.16%	329.35 s	57.33%
x264	62.88 s	81.49 s	29.59%	197.29 s	213.75%
deepsjeng	249.76 s	285.75 s	14.41%	549.63 s	120.06%
leela	333.92 s	395.62 s	18.48%	695.16 s	108.18%
xz	87.90 s	91.23 s	3.80%	107.54 s	22.34%

TABLE III: Spill protection stack size and spill count overhead measured on SPEC CPU 2017 benchmarks.

Benchmark	Spills (no instr.)	Spills (instr.)	Spill overhead	Stack size (no instr.)	Stack size (instr.)	Stack size overhead
perlbench	16779	16801	0.13%	244176 B	264240 B	8,22%
gcc	79723	79769	0,06%	847904 B	963904 B	13.68%
mcf	326	335	2.76%	67568 B	68080 B	0.76%
omnetpp	27478	27483	0,02%	439408 B	479040 B	9.02%
xalancbmk	58651	58652	0.00%	880512 B	964144 B	9.50%
x264	9327	9583	2.74%	313056 B	326704 B	4.36%
deepsjeng	711	713	0.28%	25488 B	26256 B	3.01%
leela	1780	1789	0.51%	197472 B	199536 B	1.05%
xz	1671	1703	1.92%	72784 B	75584 B	3.85%

B. Performance Evaluation

For the evaluation of our protection mechanism, we used an Apple MacMini system featuring an Apple M1 processor which supports ARM Pointer Authentication. We installed Debian Bookworn [24] with Linux kernel 5.16 from the Asahi project [25] with PA support enabled.

Earlier work on ARM PA [18] has mostly evaluated PA performance using a software equivalence w.o. real hardware access, and with an estimation that the PAC generation / validation overhead is 4 cycles. As this work evaluates performance on actual ARMA-v8.3 hardware, we have conducted some specific testing to validate the accuracy of the earlier estimations: by executing multiple dependent ALU instructions interleaved with `pacga` instructions in a single-processor thread, we can determine that the `pacga` instructions constitute the bottleneck for execution time if there are no more than 4 ALU instructions per `pacga` instruction. With 5 or more ALU instructions, the ALU processing becomes the dominant factor. Thus, we surmise that a single `pacga` instruction takes between 4 and 5 times as long to retire as an exclusive-or ALU instruction on the M1 processor. It also appears that the processor supports multi-issue of pointer authentication instructions with other operations such as arithmetic or loads/stores, enabling parallel execution and reducing the impact of the extra pointer authentications on execution time. Even though these metrics are collected only from software (without deeper information of processor hardware or pipeline structure being available to us), it does appear that prior PA overhead estimates were quite accurate, at least with respect to the Apple M1 core.

For functional and performance testing, we compiled all C/C++ integer rate base suite tests of the SPEC CPU 2017 benchmarks [26] with our instrumentation, and executed them. All the listed tests run to completion with an LLVM image including our compiler modifications. This was done in all categories (baseline, integrity, integrity+encryption), i.e., from this we can conclude the functional aspect of testing. In addition, we have run both Juliet tests and Nbench[2] benchmarks to the same effect.

Our protection scheme affects performance primarily in two aspects, which we evaluate here. First, code size increases due to the added run-time instrumentation, but also indirectly due to the higher register pressure caused by the fact that we reserve two registers for our design. Table I shows the increase in code size — we can also observe a wide variation in the size increases between programs, going from 9.96% for integrity to 28.42% for integrity+confidentiality in geometric mean: this of course reflects that different tests expose different spilling profiles to begin with, based on their respective source code

[2]https://www.math.utah.edu/~mayer/linux/bmark.html

and algorithmic complexity.

Secondly, the execution times of the programs increase due to the instrumentation and especially the added pointer authentication operations, that based on our analysis cost around 4 cycles each. Table II shows the execution overhead of different tests. The geometric means of the performance overhead land at 16.69% for integrity and 103.80% for integrity+confidentiality, but as can be seen from the table, means are hardly a relevant consideration in this case, as the overhead of register spilling is so very dependent on the code being protected and its "inclination" for register spills.

Finally, we also report on spill count increase and stack memory consumption increase in Table III. Recall that encryption does not increase stack size if it is featured together with integrity, as the amount of stored data does not change. This overhead, together with the binary overhead, translates to extra memory consumption of a running application instrumented with our compiler design. The spill overhead presented averages to 0.2% in geometric mean. This result is certainly promising in absolute terms, but it needs to be evaluated as a whole with the other features of the solution: even though spilling instructions increase by a small amount, there is still a larger performance hit due to the fact that the spilling procedure in itself becomes a more performance-hungry operation. Also, recall that our spill protector reserves 2 registers to store intermediate values while performing its duties, causing an inevitable increase in register pressure.

Stack sizes have been obtained using the LLVM back-end flags `-mllvm -stats` for each object file of a program, and then summed up. Linux `time` command has simply been used for measuring the execution times. Reported timings are averages over 10 repeated execution runs. The spill count of each benchmark has been obtained through elaboration of LLVM-generated statistics, by summing together the number of in-function spill instances (which is is less than or equal to the number of reload instances) and the number of CSRs of each object file pertaining to the benchmark.

VII. Related work

In the literature, engineering investigation on register allocation within compilers has historically turned predominantly on optimization problems [27] [28]. From this perspective, the use of spill slots to allocate variables outside of registers is seen as something to be minimized in relation to the memory access cost that each of the spill and fill operations might have. Since the work of D'Silva *et al.* in 2015 [29], reflections on the level of trust afforded to compilers began to come up in the literature. Work by Panigrahi *et al.* [10] formalizes this problem, giving definitions for *information leakage* and *relative security* between source and machine code along the compilation process, and demonstrating with numbers their poor level within LLVM.

Several studies have focused on how to enhance compilers to introduce protections for data types considered as potentially more dangerous when released in memory, i.e., code pointers. PointGuard [30] is an early example of theorizing on and implementing encryption of code pointers during compilation, which is done by XORing with a key from a random source. StackGhost [31] takes advantage of the presence of the windowed register file in the SPARC architecture to minimize the number of spills, and when necessary, explicitly invokes the kernel to save the contents of the register file safely away from the stack. These techniques suffer from cryptographic weakness, due to the lack of adequate hardware support for both encryption and key management. The landscape has definitely changed due to the introduction of facilities for fast and/or lightweight encryption of data on the fly.

An example appears in a 2015 paper, CCFI (Cryptographic Control-Flow Integrity), in which the AES-NI extension of the x86-64 architecture is used to sign 4 different types of pointers: return addresses, pointers to functions, pointers to virtual methods of a class, and exception handlers. Specifically, a 128-bit key is generated at bootstrap and continuously maintained within the extended register bank (`XMM5-XMM15`). This key is used to encrypt a pointer identifier consisting of its 48 least significant bits, plus another 80 bits indicating its class. Such a signature is placed in memory next to the pointer every time it needs to be stored. Similarly, authentication is done each time the pointer is loaded.

As for the ARM family, the Pointer Authentication (PA) facilities introduced in the architecture few years ago provide support for a large number of techniques recently presented in the literature. PATTER (Pointer AuThenTication for kERnel) [32] inserts instrumentation for protecting arguments of indirect branches. All pointers used for forward branches are signed and then stored in memory, and when they need to be used, they are loaded from memory and then authenticated. As for backward edges, prologues to sign the return value are put at the start of each function, and the return is executed through an authenticate-and-branch atomic instruction. A similar paper of the same year, PARTS (Pointer Authentication Run-Time Safety) [18], extends the same protection concept to data pointers in addition to code pointers, and questions the use of simple modifiers, highlighting replay attack risks. For this, PARTS uses a modifier based on the *ElementType* of the pointer, as defined in LLVM. Return address signing uses a 48-bit function unique identifier and the 16 most-significant bits of the `sp` value. Camouflage [17] is another work based on pointer signatures, that also tends to avoid the replay attack problems affecting the original ARM PA implementation: the modifier is built by concatenating the low-order 32 bits of `sp` with the low-order 32 bits of the address of the function. PACStack [19] creates an authenticated call stack which revisits the original concept of the shadow call stack [33] without requiring new hardware-protected memory. Here, instead of using the stack pointer as a modifier for the signature, the previously-authenticated return address is used. The latest authenticated address is stored in a compiler-reserved register.

Despite their effectiveness in protecting pointers, none of these works have an explicit vocation for register spilling security issues. A recent publication on the topic is RegGuard [34]. Its idea is to keep as much sensitive data as possible in registers, ranking them on the basis of a *security score* assigned according to the type of the variable: pointers are the most sensitive, followed by user-defined and condition

variables. When spilling is the only option, the CSR block is MACed using HMAC-SHA256 before being saved in memory. Two registers are reserved by the compiler for storing the key and the hash, respectively.

VIII. Conclusions and Future Work

In this paper, we present a solution for securing the register spilling process through modifications to the LLVM compiler and based on the use of ARM Pointer Authentication facilities in ARMA-v8.3 and later cores. By introducing our compiler design, we primarily intend to raise the awareness of register spilling as a memory security issue, and to provide a comprehensive design by which this issue can be mitigated. Protection is implemented as an additional step within the compilation process, by which both registers spilled on the stack due to register pressure and callee-saved registers within functions are protected when stored in memory.

Measurements made on the entire pool of the integer benchmarks from the SPEC CPU 2017 suite shows that the impact on stack size for integrity protection is around 4% (geometric mean), with no additional cost for encryption, due to the design choices made, while it goes from around 9% to 28% in binary code size depending on whether the protection is for integrity only or integrity+encryption. For execution time overhead, integrity protection comes at a cost of 16.69% whereas adding confidentiality protection sets the overhead to 103.80%. Although this last result may seem penalizing, it should be noted that it is the result of an overhead on the encryption of *all* register spills in the program, including callee-saved registers. Especially for spill encryption, a partial protection, based on function criticality or programmer marking of critical data can be the path for higher efficiency.

As for future phases of this work, we see three next steps. First, performance speed-up for encryption might be gained by combining the use of PACGA with ARM cryptographic Extensions (AES), where PAC provides key material for the latter primitive. Second is to either apply our design to a compiler memory protection solution that will benefit (as overlooked) the dangers of register spilling. Third, our solution can be used to complement safety solutions in languages such as Rust, where the mixing of safe and unsafe code is supported, but where the security analysis of unsafe code is done at language level without accounting for such multi-layer security issues as spilling (which will take place "underneath" both safe and unsafe code compilation). All directions may allow the resulting design to only selectively apply spilling protection where needed, which can bring down the performance overhead in such setups to more practical levels.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, IEEE, 2013.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[3] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 969–986, 2016.

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, , and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," vol. 98, pp. 5–5, 01 1998.

[5] S. Bhatkar, D. D. C, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.," in *USENIX Security Symposium*, vol. 12, pp. 291–301, 2003.

[6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[7] A. De, A. Basu, S. Ghosh, and T. Jaeger, "FIXER: Flow Integrity Extensions for Embedded RISC-V," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 348–353, March 2019.

[8] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pp. 378–387, June 2005.

[9] Q. Technologies, "Pointer Authentication on ARMv8.3." https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017. [Online; Accessed March 18th, 2022].

[10] P. Panigrahi, V. Sahithya, C. Karfa, and P. Mishra, "Secure Register Allocation for Trusted Code Generation," *IEEE Embedded Systems Letters*, pp. 1–1, 2022.

[11] K. Huang and K. Luo, "A Study of the Evolution of Defences in Linux Software and Vulnerable Register Spilling," 2016.

[12] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 98–101, 1982.

[13] Statcounter Global Stats, "Browser Market Share Worldwide." https://gs.statcounter.com/browser-market-share, 2022. [Online; Accessed March 18th, 2022].

[14] M. Salgado and R. Ragel, "Register spilling for specific application domains in ASIPs," in *7th International Conference on Information and Automation for Sustainability*, pp. 1–5, IEEE, 2014.

[15] R. 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," *IACR Transactions on Symmetric Cryptology*, pp. 4–44, 2017.

[16] M. Rutland, "Pointer authentication in AArch64 Linux." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/arm64/pointer-authentication.txt?id=fbedc599e9b891a6756b1c9bc2eead02b02cce77, 2017. [Online; Accessed March 21th, 2022].

[17] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J.-E. Ekberg, "Camouflage: Hardware-assisted CFI for the ARM Linux kernel," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[18] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "{PAC} it up: Towards pointer integrity using {ARM} pointer authentication," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 177–194, 2019.

[19] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "{PACStack}: an Authenticated Call Stack," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 357–374, 2021.

[20] A. Young and M. Yung, *Malicious cryptography: Exposing cryptovirology*. John Wiley & Sons, 2004.

[21] T. Boland and P. E. Black, "Juliet 1. 1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.

[22] A. Wagner and J. Sametinger, "Using the Juliet test suite to compare static security scanners," in *2014 11th International Conference on Security and Cryptography (SECRYPT)*, pp. 1–9, IEEE, 2014.

[23] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with PACed canaries," in *Proceedings of the 4th Workshop on System Software for Trusted Execution*, pp. 1–6, 2019.

[24] "Debian." https://www.debian.org/releases/bookworm/. [Online; Accessed April 4th, 2022].

[25] "Asahi Linux." https://asahilinux.org/. [Online; Accessed April 4th, 2022].

[26] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[27] J. Eisl, S. Marr, T. Würthinger, and H. Mössenböck, "Trace Register Allocation Policies: Compile-time vs. Performance Trade-offs," in *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, pp. 92–104, 2017.

[28] R. C. Lozano and C. Schulte, "Survey on combinatorial register allocation and instruction scheduling," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–50, 2019.
[29] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *2015 IEEE Security and Privacy Workshops*, pp. 73–87, 2015.
[30] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "{PointGuard™}: Protecting Pointers from Buffer Overflow Vulnerabilities," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.
[31] M. Frantzen and M. Shuey, "{StackGhost}: Hardware Facilitated Stack Protection," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.
[32] Y. Yang, S. Zhu, W. Shen, Y. Zhou, J. Sun, and K. Ren, "ARM Pointer Authentication based Forward-Edge and Backward-Edge Control Flow Integrity for Kernels," *arXiv preprint arXiv:1912.10666*, 2019.
[33] C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer, "Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems," *IEEE Embedded Systems Letters*, vol. 10, pp. 87–90, Sep. 2018.
[34] M. Geden and K. Rasmussen, "RegGuard: Leveraging CPU Registers for Mitigation of Control-and Data-Oriented Attacks," *arXiv preprint arXiv:2110.10769*, 2021.