Work in Progress: Securing a Critical Variable at Compile Time

Clara Bourgeais, Laure Gonnord, David Hély

Work in Progress: Securing a Critical Variable at Compile Time

Clara BOURGEAIS, Laure GONNORD, David HÉLY
Grenoble INP, LCIS - UGA
Valence, France
{clara.bourgeais, laure.gonnord, david.hely}@lcis.grenoble-inp.fr

***Abstract*—This article puts forward the idea of a co-design, between the hardware and the software of a computer system, combining hardware security features with code generation. We propose as a first contribution a countermeasure to a side channel attack that reduces the amount of critical information passing through vulnerable components, *i.e.* the bus and the cache. This solution consists of minimizing the spill of a critical data by integrating a pass, occurring between the instruction selection and the register allocation of a compiler. The method will be evaluated on RISC-V 32b processors.**

***Index Terms*—Physical attacks, Secure code generation, Hardware/Software co-design, RISC-V**

I. Introduction

Hardware systems, on which software runs, are vulnerable to physical attacks. Multiple hardware vulnerabilities are regularly discovered, necessitating new countermeasures, targeting only hardware or software level.

In order to secure end-to-end a whole system, we need to co-design the hardware and the software so that: 1) hardware security properties are expressed at code generation level; 2) available hardware countermeasures are exploited; 3) software countermeasures correct remaining hardware vulnerabilities. To address these issues, we will express and ensure security properties in both the compiler back-end and the hardware during its design. We plan to develop code generation techniques through LLVM compiler back-end passes to avoid hardware-specific vulnerabilities. We also plan to formally validate co-design methodologies [1]. These objectives are part of a thesis started in October 2024, sponsored by ARSENE, a PEPR Cybersécurité project [2].

This paper proposes a first contribution for this project. We present a work-in-progress aimed at securing programs by limiting the exposure of critical information on the cache and bus during execution on a RISC-V architecture.

The paper is structured as follows: 1) Section II examines the impact of register spilling on SCA, showing that its avoidance enhances security; 2) Section III explains our use of live ranges to prevent spilling; 3) Section IV proposes a future method to validate our work;

II. Impact of the spill on side-channel attacks

The "spill" is the compilation process of transferring a variable from a register to memory in order to release the register. This action enables the spilled register to hold a new data. During compilation, if all the registers are occupied, it becomes necessary to spill certain data (selected through heuristics) so that all the instructions can be executed properly at runtime. Spilling a variable involves moving its data through the cache or the bus, potentially exposing it to vulnerabilities, as depicted in Figure 1.

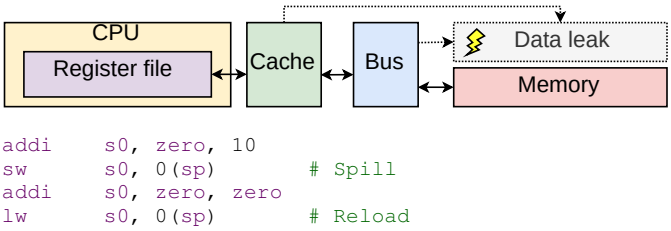

Figure 1. Illustration of a leaky register spill.

In fact, previous work has shown that both the bus and cache are vulnerable to Side-Channel Attacks (SCA) [3], [4], which are passive and non-invasive, enabling the retrieval of information (in our case, sensitive data such that -parts of- cryptographic keys), by observing auxiliary channels, like timing analysis or power consumption measurements. Such attacks can be done at relatively low-cost, with physical access to standard microcontrollers.

In this study, we thus consider that attackers can access any data passing through the bus and cache. In such cases, it is important to protect these sensitive data, or at least minimize their transactions between registers and memory. We then propose to modify the compiler spilling phase to prevent critical data for spilling.

III. Using live-ranges to eliminate leaky spilling

A. Observations

LLVM handles three main compilation stages: the front-end (which converts source code to intermediate representation), the middle-end (which optimizes the Intermediate Representation (IR)), and the back-end (which generates machine code). The LLVM back-end has two important passes which interests us the most [5]: 1) instruction selection, which maps IR instructions into target architecture instructions; 2) register allocation, which maps virtual registers to hardware specific physical registers, where spilling can occur if no register is available. The register allocator is based on early passes,

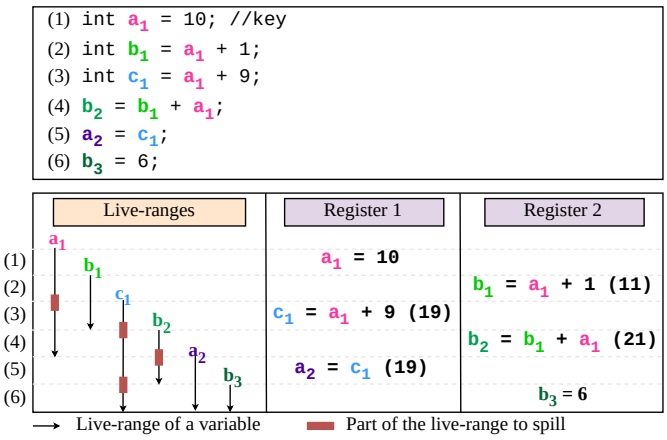

Figure 2. Illustration of live-ranges applying on a C code and impact on the register pressure.

particularly live-ranges analysis. A live-range represents the set of operations during which a variable remains available in a register, helping to determine the register pressure. Figure 2 illustrates how live-range analysis allows the compiler to identify cases where the number of available registers is not sufficient. For the sake of readability, the source code is still depicted in C-like syntax. It has been transformed into Single Static Assignment form [5], in which each variable is written only once, in order to ease the computation of the live ranges, depicted at the bottom left of the Figure. After line 3, a_1, b_1, c_1 are simultaneously*live*, requiring one to be spilled to memory. In this case a_1 is stored into memory (which is depicted by a brown rectangle).

However, a_1 being critical (as it contains a critical data), we should avoid as much as possible to prevent it for not being spilled, for all its live range.

B. Contribution

We have consequently implemented a compiler pass in the production LLVM pipeline, executed before register allocation. Since each register in an instruction is linked to a live-range, the pass iterates over IR instructions and marks selected live ranges as non-spillable using the `markNotSpillable()` function from the live-range object.

In addition, we must selectively decide which live-ranges to mark as critical. The difficulty here is that we only have this information at the source level; potentially marked as critical by the programmer, and tracing such an information from the source code to the register allocator would be a contribution per itself. Such a tracer is for the moment also work-in-progress [6]. We thus decide to use a "known-folklore" trick.

In this work, we use inline assembly volatile instructions, added after the declaration of a critical variable, in the C source code as shown in Listing 1. The `asm volatile` keyword prevents the compiler from optimizing away [7] the `key` variable. It therefore includes a command line that cannot be optimized away. The option `"r"(key)` indicates that our `key` will we stored in a register that we can retrieve.

```
uint64_t key = 0x2b7e151628aed2a6;
asm volatile ("# key %0":: "r"(key));
```

Listing 1. Illustration of an inline volatile assembly instruction in C.

IV. Limits and Evaluation

A non-spillable critical variable is protected, but if it depends on a spillable non-critical one, an attacker who knows the value of the latter at instruction time can infer the critical value. Therefore, all variables that influence a critical one must also be protected. Our approach should thus be combined with a technique similar to tainted flow analysis [8].

Moreover, if all variables stored in registers has to be spilled but are all critical, spilling couldn't occur anywhere else and compilation will remain impossible. We try to address this problem, by minimizing the total number of spill of these variables, *i.e.* modifying the register allocator strategy.

To check that our critical variables are not spilled and to assess the impact on the executable, we need to measure the number of spills in the assembly code and in our critical variable. A pass enabling this is under development. The aim is to be able to compare the spill rate without and with our countermeasure, and measure the impact on execution time.

We also intend to experimentally evaluate the SCA robustness of our solution on a 32-bit RISC-V microcontroller.

V. Conclusion

In this article, we present a method for protecting a critical variable from SCA. This method consists in preventing the spill of the variable, and is part of a wider ambition to take into account hardware specificities and existing countermeasures in a hardware system.

In the future, we would like to enhance our contribution with other data-flow analyses.

We also plan to track hardware specificities more precisely by integrating hardware/software contracts at the code generation level, in order to formally prove that a system is robust against fault attacks and SCA.

References

[1] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-Software Contracts for Secure Speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1868–1883, May 2021.
[2] "PEPR Cyber ARSENE." https://www.pepr-cyber-arsene.fr/.
[3] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, "An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, pp. 1010–1038, June 2021.
[4] E. B. Talaki, O. Savry, M. Bouvier Des Noes, and D. Hely, "A Memory Hierarchy Protected against Side-Channel Attacks," *Cryptography*, vol. 6, p. 19, June 2022.
[5] K. D. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, Aug. 2022.
[6] S. Michelland, "tracing-LLVM · GitLab." https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm/llvm, Apr. 2025.
[7] gcc.gnu.org, "Extended Asm - Using the GNU Compiler Collection (GCC)." https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Extended-Asm.html.
[8] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, pp. 236–243, May 1976.