Compiler Support for Control-Flow Linearization Using Architectural Mimicry

Daan Vanoverloop
DistriNet
KU Leuven
daan.vanoverloop@kuleuven.be

Hans Winderix
DistriNet
KU Leuven
hans.winderix@kuleuven.be

Lesly-Ann Daniel
DistriNet
KU Leuven
lesly-ann.daniel@kuleuven.be

Frank Piessens
DistriNet
KU Leuven
frank.piessens@kuleuven.be

Abstract

Architectural Mimicry (AMi) is a novel ISA extension providing hardware support for hardening against software-based microachitectural attacks where secret branch conditions are leaked through control-flow. This work extends AMi programming models to support more control-flow linearization patterns, and implements compiler support in LLVM.

1 Introduction

Modern hardware heavily relies on optimizations to improve performance. Unfortunately, these optimizations often come at the expense of security. Hardware-software co-design is a promising solution to mitigate these attacks at an acceptable cost. However, research on hardware software co-design is challenging as it requires both hardware changes and (low-level) software support. Moreover, these defenses often come with new programming models that are not always trivial to enforce. It is therefore common for such hardware software co-designs to be evaluated on small, manually-written (assembly) programs [5, 12–14] and to rely on ad-hoc manual security verification. The lack of compiler support to transparently ensure compliance with the programming models required by hardware defenses is therefore a main hurdle to a realistic evaluation of these defenses and to their adoption for provable end-to-end security.

In this work, we seek to lift this hurdle for a recently proposed hardware-software co-design called *Architectural Mimicry* (AMi) [12], which provides support for efficient control-flow linearization and balancing. The core idea of AMi is a hardware feature called *mimic execution* that mimics the microarchitectural behavior of instructions: it executes instructions for their microarchitectural effects without committing their result to the architecture. AMi comes with 1) an ISA extension to control mimic execution in software and 2) *programming models* showing how to balance and linearize secret-dependent control flow.

Unfortunately, toolchain support for AMi is limited to an assembler; it does not include automatic control-flow balancing and linearization. Furthermore, the proposed programming models are only applicable to a subset of control-flow patterns that are encountered during compilation.

PriSC, January 20, 2024, London, United Kingdom
2024.

Concretely, in this work, we:

- Generalize the AMi programming models to support linearization of *reducible* control flow;
- Add compiler support for AMi to LLVM [7], a widely used compiler infrastructure;
- Evaluate the security and performance of our compiler on Proteus [2], a RISC-V core with an implementation of AMi provided by Winderix et al. [12];

2 Architectural Mimicry (AMi)

On the hardware side, AMi relies on a primitive, *mimic execution*, which imitates instructions in terms of their timing and microarchitectural behavior and a processor mode *mimicry mode*, in which the processor mimics the execution of instructions. On the software side, to control mimic execution, AMi extends the ISA with qualifiers (**`s`**, **`m`**, **`a`**, **`g`**, **`p`**) prefixing base instructions, which we illustrate below with an example. Notably, AMi provides ISA support to linearize a branch:

```
beqz c, label; [B]; label: [...]
```

by prefixing the branch instruction with the *activating qualifier*: **`a.beqz`**. An activating branch always falls through, but if `c = 0`, mimicry mode is enabled until `label` is reached, effectively *mimicking* the execution of the branch `B` instead of jumping over it.

Even if the core idea is simple, it is non-trivial to make sure that this linearization transformation is both *secure* and *correct*. Informally, the linearization pattern above is *secure* if (1) `B` does not leak secrets itself, and (2) `B` produces the same observations in mimicry mode and standard mode so that an attacker cannot infer whether `c = 0`. The pattern is *correct* if `B` has no effect on the live state when executed in mimicry mode: only then mimicking `B` is the same as jumping over `B`.

We illustrate how to securely and correctly linearize the branch in Listing 1a, resulting in Listing 1b. First, the branch instruction is turned into an activating branch, which always falls through but enables mimicry mode until the `then`-label is reached if the branch should have been taken. In mimicry mode, instructions prefixed by the *standard qualifier* **`s`** are mimicked: for instance, `v` is not modified at line 3.

For security, it is important to make sure that observations produced by the linearized branch do not depend on the processor mode. Hence, because the `store` instructions at line 6 leaks its address, the value of `a` should be independent of the processor mode. Hence, the computation of the address

```
1 beqz c, then
2   // c != 0
3   mul v, 2
4   add a, 4
5
6   store v, a
7 then: [...]
```

(a) Vulnerable code

```
a.beqz c, then // obs = {beqz}
// enter mimicry mode if c = 0
  s.mul v, 2   // obs = {mul}
  p.add a, 4   // obs = {add}
  g.load v, a  // obs = {load a}
  p.store v, a // obs = {store a}
then: [...] // a is not live
```

(b) AMi linearization, and leakage (obs).

Listing 1. Linearizing a secret-dependent branch with AMi.

at line 4 should be committed even if the processor is in mimicry mode. This can be achieved by prefixing the `add` at line 4 with the *persistent qualifier* **p**. Note that to be correct, this transformation requires that `a` is not live at line 7.

Finally, not all instructions can be mimicked. For instance mimicking `store` instructions would require support from the memory subsystem. Hence, `store` instructions are always persistent and, to preserve correctness, it is therefore important to make sure that their effect can be nullified when executed in mimicry mode. To do so, AMi provides the *ghost qualifier* **g** which, conversely to the standard qualifier, commits the result of an instruction to the architectural state only in mimicry mode and mimics it otherwise. The `load` at line 5 therefore nullifies the effect of the following `store` only in mimicry mode.

This small example illustrates how generating correct and secure linearizations can be non-trivial, and hence compiler support would be useful.

3 Compiler support for AMi

However, modifying a compiler to ensure security and correctness of generated code is not trivial. To enforce security, the compiler must be aware of secrets and leakage model of the target architecture (cf. Section 3.1). To preserve correctness, the compiler must be aware of the semantics of AMi and how mimic/persistent instructions affect the live state (cf. Sections 3.2 and 3.3).

3.1 Static taint tracking

In order to precisely identify secret-dependent control flow, we implemented static taint tracking in the RISC-V backend. First, we gradually lower source-level security annotations on function arguments and global variables across the different LLVM abstraction layers resulting in a set of tainted input registers for each function in the backend. We then identify secret-dependent branches by performing a forward dataflow analysis, following Kildall's method, using the usual high/low security domain.

3.2 Partial control flow linearization

To efficiently linearize reducible control flow, we adapt a method called *Partial Control Flow Linearization* (PCFL), first introduced by Moll and Hack [9], and recently applied in the context of side-channel hardening [11]. Instead of removing a branch and rewriting the instructions in the branch shadow using some expensive form of conditional execution (*e.g.,* [4, 10]), as done by the original PCFL algorithm, our approach simply replaces the branch instruction by an activating branch, and only inserts instructions in the branch shadow that are necessary to preserve correctness and security (making the linearization optimal). To preserve correctness, we insert ghost loads to nullify the side-effects of stores during mimic execution. Furthermore, to enforce security we ensure that all addresses of memory operations within the branch shadow are computed persistently.

3.3 Implementation

To increase security guarantees, hardening must be applied as late as possible in the compiler pipeline. However, since the linearized form uses more registers, our transformation cannot be easily applied after register allocation. As a result, some of the hardening starts before register allocation.

Unfortunately, the liveness constraints of AMi instructions are difficult to express within existing compiler infrastructure. For instance, two standard instructions belonging to two different sides of an activating branch can write to the same physical register, as they are not live at the same time. However, this does not hold for ghost and persistent instructions: executing them could overwrite results of an instruction in the other side of the branch. The lack of support to express such liveness constraints in LLVM makes it infeasible to apply register allocation on an AMi linearized control-flow graph. To address this challenge, we carefully constrain the register allocation by adding additional liveness constraints for ghost and persistent instruction.

4 Limitations and Future Work

In contrast to existing research [3, 11], our implementation does not provide data-flow linearization, and we do not support loops with a secret-dependent trip count.

Currently, our compiler support only supports *linearization* of secret dependent control-flow with AMi. In the future, we plan to extend our compiler to additionally support control-flow *balancing* using AMi. It would also be interesting to parameterize our compiler with a *leakage contract* [6], specifying what instructions can leak, in order to balance or linearize secret-dependent control-flow more efficiently and securely. More generally, we hope that our static taint-tracking in LLVM can be leveraged to provide compiler support for other hardware-software security co-designs [5, 14]. Finally, another interesting area for future work would be to add support for AMi in secure compilers like CompCert [8] or Jasmin [1] to achieve *provable end-to-end security.*

Acknowledgments

This research is partially funded by grants of the Research Foundation – Flanders (FWO), grant number 12B2A24N, by the CyberSecurity Initiative Flanders, and by the ORSHIN project, Horizon Europe grant agreement number 101070008.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. [n. d.]. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017-10-30) *(CCS '17)*. Association for Computing Machinery, 1807–1823. https://doi.org/10.1145/3133956.3134078

[2] Marton Bognar, Job Noorman, and Frank Piessens. [n. d.]. Proteus: An extensible RISC-V core for hardware extensions. In *RISC-V summit europe '23* (2023-06).

[3] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. [n. d.]. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021-11-12). ACM, 715–733. https://doi.org/10.1145/3460120.3484583

[4] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P*.

[5] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. [n. d.]. {ProSpeCT}: Provably Secure Speculation for the {Constant-Time } Policy. 7161–7178. https://www.usenix.org/conference/usenixsecurity23/presentation/daniel

[6] Marco Guarnieri and Marco Patrignani. [n. d.]. Contract-Aware Secure Compilation. abs/2012.14205 ([n. d.]). https://doi.org/10.48550/arXiv.2012.14205 arXiv:2012.14205 [cs]

[7] C. Lattner and V. Adve. [n. d.]. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004). IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[8] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE. http://xavierleroy.org/publi/erts2016_compcert.pdf

[9] Simon Moll and Sebastian Hack. [n. d.]. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018-06-11). ACM, 543–556. https://doi.org/10.1145/3192366.3192413

[10] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*.

[11] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. [n. d.]. Side-channel Elimination via Partial Control-flow Linearization. 45, 2 ([n. d.]), 13:1–13:43. https://doi.org/10.1145/3594736

[12] Hans Winderix, Marton Bognar, Job Noorman, Lesly-Ann Daniel, and Frank Piessens. 2024. Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs. In *IEEE Symposium on Security and Privacy (SP)*.

[13] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. [n. d.]. Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS &P)* (2021-09). IEEE, 667–682. https://doi.org/10.1109/EuroSP51992.2021.00050

[14] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W. Fletcher. [n. d.]. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *Proceedings 2019 Network and Distributed System Security Symposium* (2019). Internet Society. https://doi.org/10.14722/ndss.2019.23061