An Architecture Design for Expressive Security

Jason Zhijingcheng Yu*, Prateek Saxena

School of Computing, National University of Singapore

Abstract

Today's computer systems face many security challenges such as memory safety violations, pressing need for fine-grained isolation, and growing demand for support of non-traditional trust models (e.g., confidential computing). The prevalent approach to those challenges at the architecture level is security extensions *designed to address individual security requirements. This* patchwork *of special-purpose security extensions has two main problems. Firstly, it is difficult for software to rely on them because of varying support in hardware implementations. The hardware vendors can also modify or even deprecate security extensions at any time. Secondly, when software desires multiple security goals, it has to compose multiple security extensions, which is often difficult or even impossible. In light of this, we create* Capstone*, a new architecture design that provides a* unified foundation *for achieving expressive security goals.* Capstone *adopts a capability-based model and extends it further to support exclusive memory ownership, revocable access delegation, and extensible privilege hierarchies. We show that* Capstone *enables various use cases such as two-way isolation as required by confidential computing that is arbitrarily nestable, full memory safety, and detection of Rust principle violations, all through a single uniform capability-based abstraction. We also present our design of a matching modular software system with finely isolated components as well as our hardware implementation of a concrete RISC-V-based version of* Capstone*.*

Introduction

Computer systems face many security challenges today. For example, despite decades of mitigation efforts, memory safety violations continue to account for about 70% of reported software security vulnerabilities [1]. Increasingly complex software such as web browsers and monolithic OS kernels calls for fine-grained isolation. New application scenarios such as confidential computing have also emerged. They abandon the traditional trust model and demand protection from privileged software traditionally assumed trustworthy.

Those issues are challenging to the computer system world, in large part due to the limitations of mainstream architecture designs. In response, the prevalent approach in the industry to those challenges is security extensions, which are minor changes or additions to an existing architecture to in effect *patch* its design. Each security extension typically targets a specific security goal. For example, Intel MPK targets page-level intra-process isolation, ARM RME targets confidential virtual machines. Such an approach has led to a *patchwork* of security extensions with different goals, threat models, and abstractions.

This patchwork of security extensions has two main problems. Firstly, it is difficult for software to rely on specific security extensions. Hardware implementations support different subsets of them, and hardware vendors are free to change or even deprecate a security extension at any time. This has for example happened to Intel MPX and Intel SGX (on consumer hardware), which Intel deprecated in 2021. Secondly, when software needs to achieve multiple security goals, e.g., both fine-grained intra-process isolation and confidential computing, they must compose multiple security extensions. However, in many cases, such composition is difficult if not impossible. For example, SGXLock [2] and SGXJail [3] propose using Intel MPK to protect a host process from SGX enclaves running inside the same virtual address space. This requires them to devise *ad hoc* mechanisms for banning enclave software from modifying PKRU registers. Not only are those mechanisms *ad hoc*, but they also trade off the possibility of using MPK inside enclaves.

Our proposal. Instead of relying on a patchwork of separately designed security extensions, we propose rethinking the architecture abstraction with fundamental security considerations. We present Capstone, a new architecture design which supports expressive security goals with a simple uniform architecture abstraction based on capabilities. It extends the basic capability-based model to further provide necessary security requirements including exclusive memory ownership, revocable access delegation, and extensible privilege hierarchies. We show how a concrete RISC-V-based version of Capstone enables various program safety and isolation use cases, including full memory safety and nestable two-way isolation.

Design Overview

Desired Properties

We summarise four basic properties that are essential to achieving our goal of a uniform architectural ab-

*Corresponding author: yu.zhi@comp.nus.edu.sg

straction for expressive security. We refer to each unit of isolated software component a *domain*.

Exclusive memory ownership. The abstraction should guarantee exclusive ownership to specific memory regions by a domain. Such guarantee is needed not only in one-way or hierarchical scenarios (e.g., how an OS kernel maintains exclusive ownership over kernel memory), but also in two-way scenarios (e.g., confidential virtual machines or enclaves).

Revocable access delegation. A domain should be allowed to delegate access to its memory resources to another domain. Such delegation needs to be revocable: the delegator should be allowed to invalidate the delegated access at any time. This corresponds to mechanisms such as virtual memory as means for privileged software to delegate memory to less privileged software in the hierarchical case, and memory sharing in the more general case.

Extensible privilege hierarchy. Traditional architectures have privilege hierarchies of fixed levels (e.g., U-, S-, and M-modes). The demand for containerisation, fine-grained isolation, and nested virtualisation has revealed the limitation of such a design. It is thus desirable to allow such privilege hierarchies to be extensible rather than fixed. Any domain wishing to further isolate part of itself while still remaining in control of managing of its resources can choose to extend this hierarchy further.

Secure domain switching. Preemptive domain switching is necessary to prevent denial-of-service against the whole system. When preemption takes place, all the above properties should still be upheld.

Capability-based Model in Capstone

Capstone is based on capabilities. A capability is an unforgeable token which conveys authority to access memory resources. It encodes information including memory bounds and access permissions. Software uses capabilities instead of raw pointers to access memory, upon which the hardware performs checks on the bounds and permissions. A basic capability-based model does not provide the above properties. Capstone accordingly extends the basic model to provide those properties. Central to the Capstone model are distinct capability types. *Linear capabilities* guaranteedly do not overlap with other capabilities and provides exclusive memory ownership. Capstone allows capabilities to be directly passed across domains to delegate memory access. Capabilities that have been passed out can later be reclaimed at any time with a corresponding *revocation capability*.

Case Studies

We have designed and implemented Capstone based on RISC-V. We show that it supports various security goals with a uniform abstraction.

Nestable two-way isolation. We present a modular software stack design using Capstone capabilities for isolation. The isolation is *two-way*: neither side of a software interface needs to trust the other for integrity and confidentiality of its private data. The isolation still allows system software to manage memory resources and reclaim allocated memory at any time, but without being able to access already allocated memory regions. Moreover, such revocation mechanism defines a privilege hierarchy that is extensible. A software component can allocate memory to an isolated subcomponent and retain the privilege to revoke it later, regardless of what the latter does.

Safety of mixed Rust code. Rust programs follow strict disciplines to guarantee their safety. In Rust code mixed with unsafe Rust, FFI, and inline assembly, however, the compiler is unable to statically enforce such disciplines. The Capstone abstraction provides a way to detect some violations of those disciplines at the binary level in mixed Rust code during run-time, in addition to providing full memory safety. This use case also reveals some common principles that both Rust and Capstone follow in their designs, despite their belonging to different abstraction levels and being apparently unrelated.

References

[1] Matt Miller. *MSRC-Security-Research/Presentations/2019_02_BlueHatIL/2019_01 - BlueHatIL - Trends, Challenge, and Shifts in Software Vulnerability Mitigation.Pdf at Master · Microsoft/MSRC-Security-Research*. GitHub. 2019. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf (visited on 01/17/2025).

[2] Yuan Chen et al. "SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX". In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Ed. by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pp. 4129–4146. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yuan.

[3] Samuel Weiser et al. "SGXJail: Defeating Enclave Malware via Confinement". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 2019, pp. 353–366. URL: https://www.usenix.org/conference/raid2019/presentation/weiser.