Poster: SecureCells: A Secure Compartmentalized Architecture

Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li,
Siddharth Gupta, Andres Sanchez, Babak Falsafi, Mathias Payer
Ecocloud, EPFL

***Abstract*—Modern programs are monolithic, combining code of varied provenance without isolation, all the while running on network-connected devices. A vulnerability in any component may compromise code and data of all other components. Compartmentalization separates programs into fault domains with limited policy-defined permissions, following the Principle of Least Privilege, preventing arbitrary interactions between components. Unfortunately, existing compartmentalization mechanisms target weak attacker models, incur high overheads, or overfit to specific use cases, precluding their general adoption. The need of the hour is a secure, performant, and flexible mechanism on which developers can reliably implement an arsenal of compartmentalized software.**

We present SecureCells, a novel architecture for intra-address space compartmentalization. SecureCells enforces per-Virtual Memory Area (VMA) permissions for secure and scalable access control, and introduces new userspace instructions for secure and fast compartment switching with hardware-enforced call gates and zero-copy permission transfers. SecureCells enables novel software mechanisms for call stack maintenance and register context isolation. In microbenchmarks, SecureCells switches compartments in only 8 cycles on a 5-stage in-order processor, reducing cost by an order of magnitude compared to state-of-the-art. Consequently, SecureCells helps secure high-performance software such as an in-memory key-value store with negligible overhead of less than 3%.

1. Introduction

Modern software systems are complex but monolithic, comprising multiple interacting subsystems, incorporating third-party code like libraries, plugins, or interpreted code, while interacting over untrusted interfaces including networks, shared memory, file systems, or user input. The lack of isolation between the components of a monolithic program allows vulnerabilities to have far-reaching consequences. An attacker who exploits one component can corrupt other parts — for example, a buggy Linux driver can compromise core kernel data structures. The traditional process abstraction for running monolithic software violates the Principle of Least Privilege [1] which requires components to only have access to the data necessary for their operation. Instead, all code running within a process' address space has equal permissions to all data and code regions allowing attackers to subvert pre-defined interfaces between components. For example, calls between components can jump to an arbitrary address bypassing checks on function call arguments.

Intra-address space compartmentalization allows developers to *isolate components* of a program within compartments, only granting each compartment permissions to access their own data. When compromised, a buggy component cannot access another component's data. Conversely, a component is guaranteed integrity of its private data against other corrupted compartments. Compartmentalization is a key defense mechanism that leverages the inherent modularity of code to fortify the cloud [2]and desktop [3] sandboxed environments, programs with third-party libraries [4]and underpins the design of security-focused microkernel operating systems [5]. Compartmentalization constrains the negative effects of the myriad possible faults in software, including memory safety violations and logic errors, to compartment boundaries. For example, the Log4Shell exploit (CVE-2021-44228) which allowed attackers to exfiltrate secrets and inject arbitrary code in memory-safe programs can be mitigated by isolating the vulnerable Log4j framework in a separate compartment.

The compartmentalization mechanism enforcing the rules of access and communication between the program's components must be secure, performant and flexible. To be secure, the mechanism must enforce policy-dependent restrictions on memory accesses and inter-compartment calls in the face of powerful attackers. Particularly, the mechanism must prevent compromised compartments from escalating their memory access rights or from bypassing inter-compartment call gates. Developers for performance- and security-critical software such as operating systems constantly trade off the benefits of protection mechanisms against their overheads. The mechanism must implement low overhead checks and operations to support fine-grained compartmentalization for such programs. Faster compartment switching, for example, enables developers to refactor programs into smaller compartments with more frequent compartment switches, improving security while maintaining the same performance. Finally, a flexible mechanism which is able to support the wide variety of desired compartmentalization policies will bolster developer adoption.

Existing compartmentalization mechanisms lack one or more desirable features, often trading security for performance, or flexibility for backward compatibility or implementation simplicity. Traditional, process-based isola-

tion [6], [7]only permits costly, microsecond-scale compartment switches. On the other end of the spectrum, protection-key based mechanisms [8], [9] are performant, with nanosecond-scale switches, but fail to deter attackers with code-injection capabilities. Mechanisms co-locating permissions with page-based virtual memory [8], [9], [10] improve compatibility with existing page-tables but inherit the limited reach of modern Translation Lookaside Buffers (TLBs), incurring overheads for programs with large working sets. Finally, other mechanisms [11] target simpler policies, such as protecting a single trusted compartment from an untrusted compartment.

SecureCells achieves the trifecta of secure, flexible, and high-performance compartmentalization by embedding compartmentalization into the architectural virtual memory abstraction. SecureCells proposes *i*) *TCB-maintained VMA-scale* access control, and *ii*) unprivileged (i.e., userspace) instructions implementing *securely-bounded* compartmentalization primitives, with *iii*) software implementing call gates, call stacks, and context isolation. Related efforts towards languages, compilers and libraries for compartmentalization can extend these benefits to developers by using SecureCells as the underlying isolation mechanism.

For the first pillar, access control, SecureCells introduces the first VMA-granular permissions table consolidating permissions for all compartments into a single data structure designed for efficient permission lookups. In contrast, previous mechanisms use per-compartment permission tables with either duplicate VMA bounds information [12], duplicate per-page permissions within a VMA [8], [9], or both [10], [7]. Deduplicating VMA bounds accelerates compartment switching, eliminating the need to re-load bounds for the target compartment. VMA-scale permission tracking requires smaller VMA-based permission lookaside buffers while also overcoming TLB-reach limits.

For the second pillar, SecureCells accelerates common compartmentalization operations with novel, low-cost unprivileged instructions. Particularly, SecureCells is the first mechanism to allow generic, unprivileged permission transfer from userspace. SecureCells maintains the integrity of permissions by bounding the semantics of untrusted userspace operations to known-safe parameters — the hardware checks the compartment switch instruction to enforce call gates, and permission transfer instructions to prevent privilege escalation.

SecureCells' final pillar leverages the flexibility of software for operations where possible without compromising security or performance (context isolation, call gates and call stack maintenance). This paper shows the first software mechanism for restoring register context following a compartment switch, necessary for isolating compartment contexts, without trusting any general-purpose registers.

This poster builds on work accepted for publication at the 44th IEEE Symposium on Security and Privacy.

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. [Online]. Available: https://doi.org/10.1109/PROC.1975.9939

[2] Z. Bloom, "Cloud computing without containers," https://blog.cloudflare.com/cloud-computing-without-containers/, 2018.

[3] "Project fission," 2021, accessed: Nov 2021. [Online]. Available: https://wiki.mozilla.org/Project_Fission

[4] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: language-based restriction of untrusted libraries," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 255–267. [Online]. Available: https://doi.org/10.1145/3445814.3446728

[5] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf, "Policy/mechanism separation in HYDRA," in *Proceedings of the Fifth Symposium on Operating System Principles, SOSP 1975, The University of Texas at Austin, Austin, Texas, USA, November 19-21, 1975*. ACM, 1975, pp. 132–140. [Online]. Available: https://doi.org/10.1145/800213.806531

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009, pp. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[7] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An os abstraction for safety and performance," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[8] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[9] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys - efficient in-process isolation for RISC-V and x86," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[10] D. Du, Z. Hua, Y. Xia, B. Zang, and H. Chen, "XPC: architectural support for secure and efficient cross process call," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 671–684. [Online]. Available: https://doi.org/10.1145/3307650.3322218

[11] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi, "IMIX: in-process memory isolation extension," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 2018, pp. 83–97. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto

[12] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002*. ACM Press, 2002, pp. 304–316. [Online]. Available: https://doi.org/10.1145/605397.605429