CubicleOS: A Library OS with Software Componentisation for Practical Isolation Extended Abstract

Vasily A. Sartakov, Lluís Vilanova, Peter Pietzuch
Imperial College London
{v.sartakov, vilanova, prp}@imperial.ac.uk

1. Motivation

In cloud environments, library OSs are gaining increasing traction when users want to make deployed applications self-contained in terms of OS functionality. They are used to deploy lightweight unikernels [9, 5, 7], make containers more efficient [12], and run shielded applications inside of trusted execution environments (TEEs) [15, 11].

Library OSs such as Graphene [14], IncludeOS [1] and Unikraft [7] are typically assembled from various independent library components (e.g., file system libraries, network stacks, and low-level drivers). To minimise the image size when linked with an application, the components required for a given application are selected at compile-time, and all OS and application components then execute as part of a single, unprotected address space.

This lack of compartmentalisation of library OS components raises security, robustness and reliability concerns. These issues are well-known deficiencies of monolithic designs, especially when complexity library OS components are exposed over the network [10]. For example, a vulnerability in a file system implementation may be exploited to compromise the library OS and application and then disclose encrypted keys from the TLS implementation [2].

We therefore argue that current library OS designs risk being a step backwards in terms of security, robustness and reliability. The research question that we explore in this paper is whether it is possible to design a modular and compartmentalised library OS with existing, third-party components, while enforcing practical isolation between these components.

2. Limitations of the State-of-the-Art

Existing library OSs often follow monolithic designs that simplify the integration of third-party components but lack compartmentalisation. In contrast, microkernel designs [6, 8, 4, 13] impose standard interfaces between kernel components, e.g., based on message passing or RPC-like calls, which can be used to enforce protection boundaries between components.

In a microkernel-style system, however, developers of isolated components must carefully design component interfaces to match the microkernel's message passing primitives. For example, this may affect the data structures used by components because data must be serialised and deserialised through the microkernel's primitives each time components interact. As developers try to reduce cross-component communication due to its well-known overheads, i.e., memory copies and context switches performed by the microkernel, it also affects the nature of interfaces that expose functionality across components.

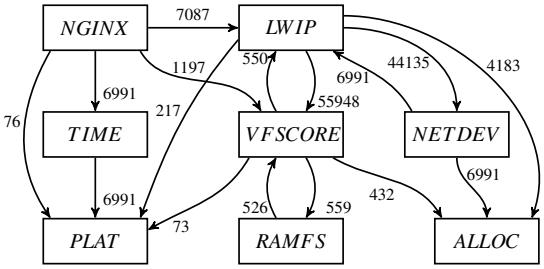

Figure 1: Interaction of various library OS components when executing a web serving benchmark with NGINX (Numbered arrows signify cross-component calls and their counts during the benchmark.)

When building a libraryOS, adjusting existing third-party components in these ways becomes cumbersome; the difficulty of this task becomes apparent if we look at the complex interactions between the OS and application components of a prototypical web server (NGINX), as shown in Figure 1. This explains the popularity of monolithic designs, which do not impose restrictions on interfaces and components. Especially in library OSs that strive for full POSIX compatibility, e.g., to execute current Linux applications and thus heavily rely on existing OS components, we observe a strong preference for monolithic designs. There seems to be a fundamental trade-off between compartmentalisation and development burden in library OSs.

3. Key Insights

Our goal is to retain the flexibility of arbitrary interfaces between components, as found in monolithic kernel designs, while compartmentalising the system to enforce spatial and temporal memory isolation with an acceptable performance overhead and without invasive source code changes.

We present a system – *CubicleOS* – that automatically isolates OS and application components in a rich library OS, while allowing them to share dynamically arbitrary data used across function calls. CubicleOS uses Intel's support for Memory Protection Keys (MPK) to efficiently compartmentalise the system, while at the same time allowing on-demand data sharing across compartment calls.

CubicleOS thus presents a system that conjoins existing monolithic library OS code bases with strong isolation guarantees without performing expensive data copies or interacting

with the privileged host OS on the critical path.

4. Main Artefacts

We implement CubicleOS on top of Unikraft [7], a feature-rich library OS that can execute existing POSIX-compatible applications, and runs on top of a host OS such as Linux. CubicleOS offers three core abstractions to component developers: (i) *cubicles*, which are isolated components; (ii) *windows*, which enable dynamic sharing across components; and (iii) *cross-cubicle calls*, which carry out control flow authorisation. Together, these abstractions provide spatial memory isolation, temporal memory isolation and control flow integrity, respectively.

CubicleOS enforces the isolation policies expressed by developers (via the abstractions introduced above) using four complementary methods: (1) a trusted *build tool* that automatically identifies components (cubicles) to isolate and generates additional trusted code for each exported function in a cubicle (cross-cubicle call trampolines); (2) a trusted binary cubicle *loader* that verifies no untrusted code uses instructions that would affect the integrity of the security mechanisms (necessary because Unikraft executes as a user-level process); (3) a small, *trusted run-time* that uses Intel MPK and the cross-cubicle call trampolines generated above to efficiently isolate cubicles without host OS intervention in the critical path; and (4) a trivial extension to Intel's MPK implementation that ensures control flow integrity across cubicles without resorting to multiplexing all cross-cubicle calls through a central authority.

5. Key Results and Contributions

We evaluate CubicleOS using an embedded database engine (SQLite) and a web server application (NGINX). We show that it is between 3× and 5× faster than a state-of-the-art microkernel with equivalent isolation guarantees (Genode [3]), and between 1.7× and 8× slower than a non-isolated Unikraft baseline. These results can be obtained with only moderate source code changes without impacting the design of existing OS and application component interfaces.

The core contribution of CubicleOS is the compartmentalisation of the monolithic architecture of an existing library OS, without fundamentally altering the design of its existing interfaces. We believe that this is a necessary step forward, and shows that the requisite compartmentalisation of library OSs is not at odds with maintaining their existing monolithic architectures.

6. Why ASPLOS

The paper provides system abstractions and mechanisms to integrate isolation with existing monolithic codebases in a library OS, using recent hardware support (Intel MPK) and automated code analysis and generation to enforce isolation without disruptive code changes.

7. Citation for Most Influential Paper Award

This work addresses the concern around the lack of compartmentalisation in existing library OSs. By emphasising the ability to compartmentalise monolithic codebases without disruptive changes to their design, the paper has had an enduring influence on improving the security, robustness and reliability of library OS implementations in a practical way.

8. Revisions

The paper has no previous revisions.

References

[1] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257. IEEE, 2015.

[2] CVE-2018-5410. Available from MITRE, CVE-ID CVE-2018-5410.

[3] Genode Operating System Framework. https://github.com/genodelabs/genode. Last accessed: Aug 1, 2020.

[4] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[5] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, 2014.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[7] Simon Kuenzer, Sharan Santhanam, Yuri Volchkov, Florian Schmidt, Felipe Huici, Joel Nider, Mike Rapoport, and Costin Lupu. Unleashing the power of unikernels with unikraft. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 195–195, 2019.

[8] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 175–188, 1993.

[9] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[10] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, 2014.

[11] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[12] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–135, 2019.

[13] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222, 2010.

[14] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[15] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.