The Case for Intra-Unikernel Isolation

Pierre Olivier[1], Antonio Barbalace[2], Binoy Ravindran[3]
[1]The University of Manchester, [2]The University of Edinburgh, [3]Virginia Tech
pierre.olivier@manchester.ac.uk, antonio.barbalace@ed.ac.uk, binoy@vt.edu

Abstract

The unikernel is an emerging operating system model offering lightweightness, security and performance benefits. In this paper we argue that a fundamental design principle of unikernels, the fact that one instance is viewed as a single unit of trust, is not suitable for the high security requirements of today's cloud applications. We advocate for the introduction of intra-unikernel isolation. We observe that some unikernel benefits derive from another fundamental design principle: the presence of a single address space. We investigate bringing intra-unikernel isolation without breaking that principle with the help of hardware technologies in the form of modern (Intel Memory Protection Keys) and future (hardware capabilities) Instruction Set Architecture extensions.

1 Introduction

The unikernel [18] is a new Operating System (OS) model in which an application and its dependency libraries run alongside a thin Library Operating System (LibOS) [7] on top of a hypervisor in a VM. Unikernels are a form of lightweight virtualization and provide various benefits including attack surface reduction, low resource usage leading to cost reduction/high consolidation, strong isolation, as well as improved system performance.

As a result, unikernel application domains are plentiful, encompassing cloud- and edge-deployed micro-services/SaaS/ software [2, 14, 19, 27], server applications [14, 17, 18, 27, 38], NFV [5, 18–20], IoT [5, 6], HPC [16], efficient VM introspection/malware analysis [37], and regular desktop applications [24, 29]. In particular, unikernels have an important role to play in the upcoming explosion of serverless computing/FaaS [8, 10, 13].

Regarding security, with this OS model one application runs per VM instance so the isolation between several unikernels (i.e., applications) running on the same host is considered strong [19], which is ideal from the cloud provider point of view. However, one of the fundamental design principles of unikernels is that application and LibOS code/data share a single and unprotected address space [18]: *there is no isolation within a unikernel instance, which is viewed as a single unit of trust.* This is concerning from the tenant standpoint, because with all of the current unikernel offers, a component of the application manipulating sensitive data cannot be isolated from less trusted components such as third party or memory-unsafe code. This lack of internal isolation has been noted as one of the main security issues of unikernels [21].

In this position paper, *we advocate for an evolution of this OS model to support the isolation of software components within a unikernel instance.* We observe that several performance benefits of unikernels result from the use of a single address space per instance, enabling for example fast context switches or low system call latencies. Thus, we propose to bring intra-unikernel isolation while keeping a single address space per VM. We investigate the use of modern and future security ISA extensions. More specifically, we report early results about the use of Intel Memory Protection Keys (MPK) and the associated challenges. We then look at a promising emerging technology, namely hardware capabilities, and reason about its application to the problem of intra-unikernel isolation. Because of the peculiarities of each technology, we expect the related isolation models to be quite different, in particular regarding isolation granularity. Furthermore, both technologies offer different characteristics in terms of security guarantees, scalability to high number of protection domains, as well as programmability/ease of applicability to existing unikernel codebases. Focusing on these technologies is motivated by the fact that they allow low-latency security domain switches within a single address space. Low-latency switches align with unikernel design principles and objectives. Finally, it is worth noting that some of the isolation ideas presented here may also apply to other OS models.

In the rest of this paper, we make the case for intra-unikernel isolation in Section 2. Next, we discuss the use of Intel MPK as well as hardware capabilities to implement compartmentalization in unikernels in Section 3.

2 Advocating for Intra-Unikernel Isolation

There are many situations in which the lack of intra-unikernel isolation raises serious security concerns. Modern applications are made of several components/libraries that have various degrees of trustworthiness (for example third-party vs. own code) and a variable potential for vulnerabilities (for example a formally verified cryptography library vs. a user-facing input parser) [1]. Isolating components within an application is useful from the security point of view, so as to avoid scenarios in which a subverted vulnerable component leads to the attacker taking over the entire system, including other components manipulating sensitive data. To bring such intra-application isolation, a Trusted Computing Base (TCB) has to be established to enforce an isolation policy. With current unikernels, the LibOS cannot play that role as it runs at the same level of privileges as the user code.

In order to bring intra-application isolation to unikernel environments, we envision two models for placing the TCB, depicted on Figure 1. A first model, present on the left side of the picture, has each application component running in a different VM instance, placing the TCB in the hypervisor. This is the model obtained when using for example the KylinX [38] unikernel, implementing support for `fork` by spawning a unikernel per process. We believe that this model conflicts with the lightweightness objectives of unikernels: indeed, the per-instance memory footprint is multiplied by the number of application components. Furthermore, this model breaks the single address space principle and as a consequence switching between security domains is costly (VMEXIT, EPT switch) and impacts performance. Thus, we advocate for a second model, pictured on the right of Figure 1. In this model, we establish another layer of trust on top of the hypervisor, within the unikernel LibOS, which

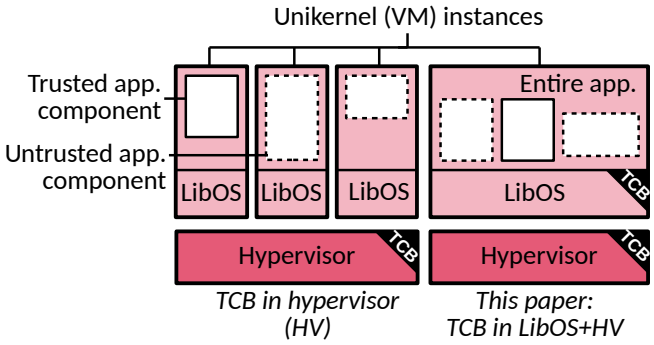

Figure 1: Trust models for isolation in unikernel environments: in-hypervisor (left) and in-LibOS + in-hypervisor (right) Trusted Computing Bases (TCBs). The TCB represents the code we trust to enforce isolation between application components.

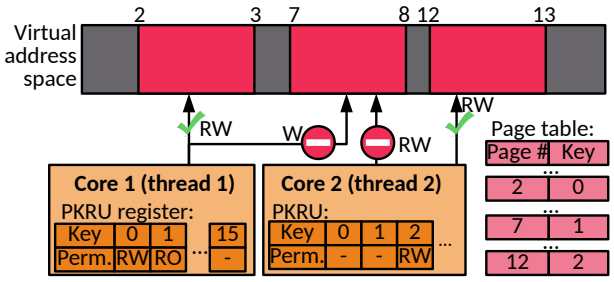

Figure 2: Intra-address space isolation with Intel MPK. PKRU registers and page table entries are setup in order to give different access permissions to threads 1 and 2 regarding the virtual pages 2, 7 and 12.

is now responsible for enforcing isolation within the instance. This preserves lightweightness and performance with a single instance and address space per application, but requires isolation between the application code and the LibOS, i.e., user/kernel separation in the guest.

A second security issue resulting from the lack of intra-unikernel isolation appears in unikernels LibOSes written with memory-safe languages [3, 15, 18, 34]. Although they offer stronger safety guarantees compared to LibOSes written in C/C++ [11, 12, 22], they all include a certain quantity of untrusted code necessary to perform the low-level operations an OS has to support. It can be either plain C code [3, 18, 34] or unsafe Rust code [15]. This can negate the security guarantees that come from using a memory safe language – an attacker may exploit a bug in an unsafe kernel component to take over the system. This further motivates the need for intra-unikernel isolation in the form of safe/unsafe kernel components separation.

3 Leveraging Modern and Future ISA Extensions

Unikernels have by design no user/kernel isolation, a deliberate choice made to cut the switching latency and enable cross kernel-user compiler optimizations. Therefore, we consider intra-unikernel isolation solutions that do not involve costly page table or classical user/supervisor switches. A historical way to provide protection among a fragmented address space is segmentation. However, it is unavailable/deprecated in most modern ISAs, including x86-64. Although there exist other memory protection techniques that may apply to unikernels, such as Mondrian protection [35], in this paper we focus on the ones that have real-world implementations or will have one in a near future. More precisely, we consider two recent/emerging technologies that satisfy our requirements: Intel MPK [4] and hardware capabilities [36].

3.1 Intel MPK

Memory Protection Keys is a technology available on recent Intel processors that allows memory protection between cores sharing a page table – i.e., within threads sharing a single address space. Each core (or more precisely, hardware thread) has a register named PKRU specifying the core's permission for each of the 16 available keys. These permissions are read-only, read-write, and no access. Each page of the address space can be tagged with a specific key using four bits of the corresponding page table entry. An example of MPK operation is described on Figure 2, where three protection keys are used to give different access permission to various memory areas to two threads sharing a single address space. Switching between protection domains on a core simply corresponds to updating the PKRU register, which is very fast [30]. Combined with the fact that it operates within a single address space, this makes MPK a very compelling technology to provide intra-unikernel isolation.

However, using MPK is not without limitations/challenges. First, the limited number of memory keys (16) is a problem and although it can be virtually increased [23], this will be at the cost of a performance impact. Second, for performance reasons the PKRU switch is an unprivileged operation, and isolation schemes must be complemented by static code analysis to validate all PKRU manipulation [30]. Indirect PKRU tampering through techniques such as Return-Oriented Programming [26] can be mitigated with Address Space Layout Randomization – an issue in some unikernels supporting only static binaries.

Unikernel Isolation Model. Due to the low number of protection keys, with MPK we advocate for a coarse-grained intra-unikernel isolation model. A simple example of such model would provide isolation between (1) user and kernel space; (2) trusted and untrusted kernel components; (3) trusted and untrusted user components. Realising (1) and (2) is a direct responsibility of the unikernel LibOS developer. (3) falls out of his/her control, however the LibOS should provide APIs in order to allow the creation of isolation domains in user space: a possible way to do so would be an extended version of `mprotect`).

We recently implemented user/kernel separation as well as safe/unsafe kernel components isolation [28] in the RustyHermit [15] unikernel. The work consisted in implementing MPK support for the LibOS, writing the protection domains switch code, and segregating in memory data that should be shared between domains from data that should not. For this last task we relied on custom ELF sections for static data as well as segregated heaps/stacks. Shared data is marked as such with simple attributes added to the code by the programmer. Results are encouraging as the isolation scheme

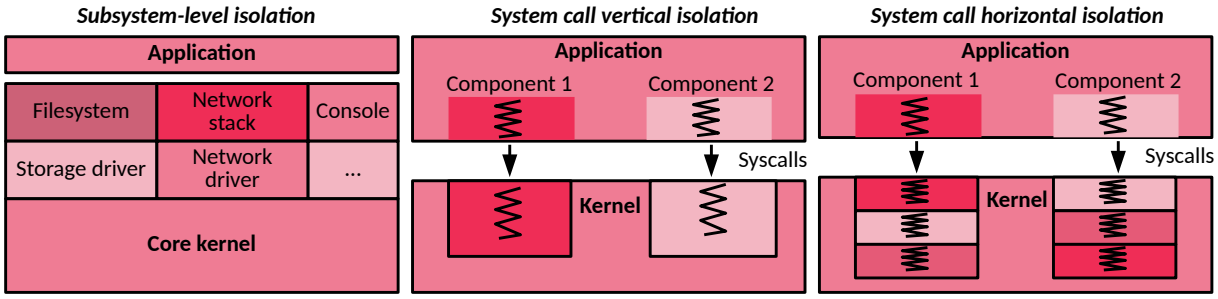

Figure 3: Various compartmentalization strategies offered by capabilities' fine-grain isolation properties: subsystem-level (left), system call vertical (center) and system call horizontal (right) compartmentalization.

provides security with a very low overhead. In other words, we are able to maintain unikernel lightweightness properties: system calls latency is 3*x* faster compared to Linux, due to the MPK security domain switch operation being much faster than a traditional user/kernel world switch. We also measured an average of 0.6% slowdown over a set of memory/compute intensive macro benchmarks compared to a non-isolated unikernel execution.

3.2 Hardware Capabilities

A rather old concept still getting attention today is hardware capabilities. It allows, among other benefits, the *compartmentalization* [33] of software at a very fine grain (byte level) in a single address space. It has been recently implemented by the CHERI project [36] on a MIPS architecture. CHERI is an extension of legacy ISAs to support capabilities and is currently ported to ARM [9] as well as RISC-V [32]. A real-world prototype board from ARM, one of the ISAs used in the data center [25], is expected for 2021 [9]. Hence, we foresee that in the near future hardware capabilities will be available on multiple CPUs of diverse ISA, some of which used in regular computers servers/desktop, but also others used as controllers for devices/smart devices.

OS and compiler code to support hardware capabilities has been developed before by the projects around CHERI [31], including support for FreeBSD and LLVM. The overhead reported by these pioneering projects for using hardware capabilities is very low, justifying usage atop a paged memory system. Moreover, CHERI type capabilities come in two main forms, what is called the hybrid capability and pure capability. Hybrid capabilities are similar to a segmented memory scheme, this may definitely be used to divide kernel code from application code as well as different parts of the kernel or the application between each other, with a low-overhead. As an alternatively, pure capabilities can be used as well, but they require compiler support, and memory is accessed via capabilities that maybe heavy, moreover, pointers are saved as capabilities thus using double the normal memory space.

Hardware capabilities are a compelling technology to implement intra-address space isolation for unikernels, but currently, it is not clear what will be their performance in production.

Unikernel Isolation Model. The benefits of hardware capabilities are the fine-grain level of protection and the potentially infinite number of security domains. Security domains boundaries can be set at the function call level while keeping good performance.

The unikernel LibOS could be decomposed in many different ways. Different isolation strategies are presented on Figure 3. Isolation can be achieved at the subsystem level (left of Figure 3), providing a micro kernel-like level of security while requiring modest changes to existing unikernel codebases. Another example would be to vertically compartmentalize LibOS services invocations made by untrusting user components (center of Figure 3). Horizontal compartmentalization can also be used in the kernel to isolate the different processing steps of untrusted inputs, coming either from the user (e.g., a file write operation) or from the hardware (e.g., an incoming network packet), as depicted on the right of Figure 3.

Regarding application code, it can either be legacy (capability-unaware) code, in which case the kernel should provide a system-call like API, or wrappers at another level such as the C library, for automated capability enabling tools to help bring its support for legacy code. Some applications will also be written with capabilities in mind. The interface the OS should provide to these can be a set of object-capabilities representing various system services such as memory, files, etc [33]. It should be noted that with unikernels the LibOS and application are compiled together and reside at runtime within the same address space, this provides the opportunity to compartmentalize application and OS altogether.

Capabilities should allow for a much more secure and scalable protection model compared to coarse-grained solutions such as MPK, however it is likely that these solutions would involve a significant redesign of existing unikernel codebases. The additional memory overhead brought by capabilities could also conflict with the minimal footprint objective of unikernels.

4 Conclusion

The unikernel is a promising OS model, however it lacks an internal isolation mechanism to offer sufficient security guarantees for modern applications. We propose and analyze the use of modern (Intel MPK) and future (hardware capabilities) ISA extensions in order to provide intra-unikernel isolation while still keeping the single-address space feature of this OS model so as to maintain their lightweightness characteristics.

Acknowledgements

This work was supported in part by the US Office of Naval Research under grants N00014-18-1-2022, N00014-16-1-2104, and N00014-16-1-2711.

References

[1] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E. Locasto, Jason Reeves, Sean W. Smith, and Anna Shubina. 2013. *ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection.* Technical Report TR2013-727. Dartmouth College, Computer Science, Hanover, NH. http://www.cs.dartmouth.edu/reports/TR2013-727.pdf

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

[3] Cloudozer LLP. 2017. LING/Erlang on Xen website. http://erlangonxen.org/. Online, accessed 11/20/2017.

[4] Jonathan Corbet. 2015. Memory protection keys. *Linux Weekly News* (2015). https://lwn.net/Articles/643797/.

[5] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet'17)*. ACM, 36–41.

[6] Bob Duncan, Andreas Happe, and Alfred Bratterud. 2016. Enterprise IoT security and scalability: how unikernels can improve the status Quo. In *IEEE/ACM 9th International Conference on Utility and Cloud Computing (UUC 2016)*. IEEE, 292–297.

[7] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. 1995. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 251–266.

[8] Henrique Fingler, Amogh Akshintala, and Christopher J Rossbach. 2019. USETL: Unikernels for serverless extract transform and load why should you settle for less?. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. 23–30.

[9] Richard Grisenthwaite. 2019. A Safer Digital Future, By Design. https://www.arm.com/blogs/blueprint/digital-security-by-design.

[10] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Popa, Ion Stoica, and David Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*.

[11] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[12] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[13] Michał Król and Ioannis Psaras. 2017. NFaaS: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM, 134–144.

[14] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'17)*. ACM, 15–29.

[15] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS'19)*. ACM, New York, NY, USA, 8–15. https://doi.org/10.1145/3365137.3365395

[16] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*. ACM.

[17] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 559–573.

[18] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud.. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 461–472.

[19] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[20] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[21] Spencer Michaels and Jeff Dileo. 2019. Assessing Unikernel Security.

[22] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*.

[23] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC19)*. 241–254.

[24] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1950365.1950399

[25] Amazon Web Services. 2019. AWS Graviton Processor. https://aws.amazon.com/ec2/graviton/.

[26] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[27] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox 2016)*. ACM, 44–49.

[28] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2019. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20)*.

[29] 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. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. ACM, 9.

[30] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys. *USENIX Security Symposium* (2019).

[31] Robert Watson. 2019. CHERI Software Stack. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-software.html.

[32] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W Moore, et al. 2017. *Capability hardware enhanced risc instructions: Cheri instruction-set architecture (version 6).* Technical Report. University of Cambridge, Computer Laboratory.

[33] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.

[34] Adam Wick. 2012,. The HaLVM: A Simple Platform for Simple Platforms. Xen Summit.

[35] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 304–316.

[36] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.

[37] Xen Website. 2018. Google Summer of Code Project, TinyVMI: Porting LibVMI to Mini-OS. https://blog.xenproject.org/2018/09/05/tinyvmi-porting-libvmi-to-mini-os-on-xen-project-hypervisor/, Online, accessed 10/30/2018.

[38] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*.