xOS: The End Of The Process-Thread Duo Reign

Alain Tchana*
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG,
38000 Grenoble, France

Dorian Goepp
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG,
38000 Grenoble, France

Stella Bitchebe
McGill University, School of Computer Science,
Montreal, Canada

Renaud Lachaize
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG,
38000 Grenoble, France

ABSTRACT

Process and *Thread* are first-order abstractions of the operating system (OS), whose implementation is wired into the OS core. Several research works have shown the inadequacy of these two main abstractions for modern isolation needs, leading to the introduction of additional abstractions with new isolation and communication features. Despite their usefulness, these new proposals are introduced in a somewhat ad-hoc manner, compromising their broad and consensual adoption.

This position paper presents xOS, an OS design that does not introduce yet another first-class isolation abstraction but instead investigates how the OS can help application programmers, libraries, and OS developers integrate and easily use new abstractions. To our knowledge, xOS is the first work in this area. Similar to file system development built around a Virtual File System (VFS), xOS introduces the concept of Isolation Context (IC), which should be the unique first-class abstraction wired into the OS core. ICs can be realized in several pluggable Isolation Context Factories (ICFs) such as ProcessFactory (provides processes), ThreadFactory (provides threads), Docker Engine (provides Docker containers), KVM (provides KVM virtual machines), Wasp (provides virtines), etc. We discuss our plan to redesign a general-purpose OS from these foundations, the required APIs, and how to support new and legacy applications.

*Corresponding author: alain.tchana@grenoble-inp.fr

APSys '23, August 24–25, 2023, Seoul, Republic of Korea

ACM ISBN 979-8-4007-0305-8/23/08…$15.00
https://doi.org/10.1145/3609510.3609817

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Security and privacy** → **Operating systems security**;

KEYWORDS

Operating Systems, Security, Isolation, Concurrency

ACM Reference Format:
Alain Tchana, Dorian Goepp, Stella Bitchebe, and Renaud Lachaize. 2023. xOS: The End Of The Process-Thread Duo Reign. In *14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '23), August 24–25, 2023, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3609510.3609817

1 MOTIVATION

An OS should allow application developers to satisfy three fundamental needs:

- ($\mathbf{N}_1$) defining activities (called *Execution Flows*, EFs for short) that can execute concurrently and enable work distribution;
- ($\mathbf{N}_2$) defining isolation boundaries (code and data memory areas that EFs can access) to enforce safety and security guarantees;
- ($\mathbf{N}_3$) establishing communication between EFs.

The first two needs above ($\mathbf{N}_1$ and $\mathbf{N}_2$) constitute the two sides of the same coin, which is *the need for separation*: identifying distinct flows and delineating their respective protection domains. And the third need ($\mathbf{N}_3$) is a consequence of separation: facilities for synchronization, data exchange, and remote/cross-domain code invocation. Pursuing these goals has resulted in a large body of works that have proposed solutions and abstractions [6, 7, 15, 19–21, 24–26, 28] which, albeit innovative and ingenious, are mostly narrower to specific cases and, therefore, not generic. This led us to the following observations that motivate xOS.

Observation 1: First-class abstractions proliferate over the years. The notions of *process* and *thread* are the two first-class abstractions offered by contemporary OSes (to satisfy $\mathbf{N}_1$ and $\mathbf{N}_2$), accompanied by Inter-Process Communication (IPC) mechanisms (to satisfy $\mathbf{N}_3$). They were introduced in the early years of the systems design field and have been

Table 1: Examples of first-class abstractions for isolation and communication introduced in recent years.

Name	Isolation Units	Communication Mechanism	Hardware Features Used
Dune [6]	Dune processes	In-VM system calls	Intel VT-x
SeCage [20]	Secret compartments	VMFUNC	Intel VT-x & Intel VMFUNC
LwC [19]	Lightweight contexts	Memory mappings switching	-
Scone [4]	Secure containers	Shields & asynchronous syscalls	Intel SGX
Skybridge [21]	Client/server processes	EPTP switching VM function	Intel VMFUNC & Intel EPT
Donky [26]	Security Domains	Domain calls	Intel MPK
cubicleOS [25]	Cubicles	Cross-component function call	Intel MPK
cVM (cap-vm) [24]	cap VMs	Capability-based function call	CHERI
Virtines [28]	Virtines	Hypercalls[1]	Intel VT-x, AMD SVM
orbit [15]	Orbit tasks	PTEs copying[2]	-

intended for activities with precise modularity and isolation boundaries. These abstractions statically implement, throughout an EF's entire lifetime, a mapping between an EF and an isolation domain (e.g., a per-flow separate address space in the case of single-threaded processes). This rigidity of processes and threads has been regularly challenged for being unsuitable or insufficient, especially for new EF and isolation scenarios [15], which seek extensibility (e.g., user-defined functions, web browser extensions, kernel extensions, etc.), secure partitioning (e.g., session handlers, key signing, etc.), and observability (e.g., deadlock detection, garbage collection, checkpointing, etc.).

To fill this gap, numerous first-class abstractions [6, 7, 15, 19–21, 24–26, 28] and, consequently, communication mechanisms have been proposed in the scientific literature in a constant flow over the past 20 years — some examples are listed in Table 1.

Most of these abstractions take advantage of recent hardware features introduced by manufacturers for different purposes: virtualization (e.g., VT-x [11], EPT [14], VMFUNC [12], APICv [13], etc.), confidential computing (e.g., Intel SGX [10] and ARM TrustZone [1], AMD SEV [2] and Intel TDX, etc.), or safety and security (e.g., CHERI [29]). As a result, there are many ways to separate software components (and their respective execution flows) from each other and to let them communicate, at diverse levels of granularity.

All of these prior proposals in this area are somewhat ad hoc or "siloed": each one comes with its own base concepts, and, more importantly, its own set of specific integration hooks within the deepest layers of existing OSes. This complicates the introduction of new abstractions, as well as their bridging with existing ones. As a consequence, research in the domain is slowed and impeded.

Proposition 1: *We claim that it is possible to define a general principle and a standard methodology for developing new first-class isolation and concurrency abstractions to be integrated within the OS.*

Observation 2: Leaking design choices regarding tasks and isolation in OS APIs impedes application adaptation. The above-mentioned proliferation of low-level abstractions is not only problematic for kernel maintenance and evolution but also for application developers because abstractions are fully exposed to the developers and thus strongly impact the application design. Indeed, applications and/or libraries generally require intrusive modifications to be ported from one point in the design space to another. This is detrimental for application developers in many situations, e.g., to support different platforms or to explore different trade-offs between performance and security. As a result, application developers will generally settle for the "least common denominator," e.g., using pervasive and possibly coarse-grained abstractions such as processes or virtual machines.

Moreover, this problem also introduces limitations regarding the dynamic optimization of applications, as it is generally impossible to transparently reconfigure the task management, isolation, and communication abstractions they use. To illustrate this, let us consider an application developer that opts for multi-threading to benefit from efficient shared-memory communication. Consequently, her application cannot be distributed over several computers and can only scale within the limits of one physical machine. If, conversely, the developer chooses to rely on containers or virtual machines (VMs) to achieve independent scaling, she must use inter-machine communication channels such as TCP sockets. In this case, if the containers or VMs are (re)located on the same computer, they would still need to

[1]Through virtine hypervisor
[2]Between orbit tasks

use inter-machine communication, which is a suboptimal choice in the present situation. Developers, therefore, face a dilemma: either make the application horizontally scalable at the cost of performance or forfeit scalability for better performance. This dilemma comes from the developer being compelled to commit statically to the abstraction and the communication mechanism she will use while ignoring how the application will be distributed in the infrastructure.

Another negative consequence of this proliferation of compartmentalization interfaces exposed to application code is that they are difficult to design and use correctly, which results in a new and growing set of security vulnerabilities [17]. We believe that abstracting away the isolation primitives exposed to the applications could help reduce the number of such vulnerabilities and simplify their mitigation by containing them in more controlled parts of the OS.

Proposition 2: *We propose to decouple the need for the expressiveness of concurrency (flow separation) and communication between EFs from the choice of isolation abstraction and the underlying communication mechanism.*

2 TOWARDS A NEW OS DESIGN

xOS is a new OS design with two key differences from existing OSes: (1) it facilitates the integration of new first-class isolation abstractions, and (2) it arbitrates the choice of isolation abstractions (and communication channels) for parallel applications.

Fig. 1 summarizes the architecture of xOS. At the bottom level (yellow boxes), xOS manages the isolation abstractions, which can be implemented in kernel space or userspace (§ 2.1). At the top level (orange boxes), xOS exposes a parallel programming model to developers of end-user applications (§ 2.2). Unless the developer explicitly wants to choose the isolation abstraction to apply manually, xOS can embed a runtime that determines the most *appropriate* abstraction (§ 2.3) based on the application (its type, the context of its deployment, etc.). To do so, the developer specifies the applications workflow (i.e. the set of sequential and parallel tasks, as well as their communication graph), the isolation boundaries required for specific pieces of code and data, and indications regarding the desired security/performance trade-off. After that, xOS instantiates the isolation unit closest to the developer's needs, depending on available hardware features (EPT, SGX, etc.), and applies the corresponding communication mechanism (shared memory, sockets, etc.).

2.1 Virtual Isolation Contexts (VIC)

xOS is inspired by the way current OSes handle file systems. They provide a Virtual File System (VFS) interface that defines a set of standard file operations (e.g., `struct inode_operations` for Linux). Concrete file systems implement these operations and register themselves with the VFS.

Similarly, xOS includes the concept of *Virtual Isolation Context* (VIC) at its core, which provides a central foundation for abstracting the differences between various isolation abstractions. *Isolation Context Factories* (ICFs) register with the VIC layer and each ICF provides a distinct implementation of the VIC interface, which leverages one or several isolation techniques and one or several communication mechanisms (such as the ones described in Table 1). We use the term *Isolation Context* (IC) to refer to each instance of isolation domain created by an ICF. Note that ICFs can be implemented as kernel modules or as user-level components (see respectively the LwC and Scone examples in Figure 1).[3]

Below, we briefly describe the main operations of the VIC interface that every ICF must implement.

- `ic_init()`: first checks whether required (hardware) features are available (e.g., VMFUNC for SeCage [20]) and initializes any needed state/data.
- `get_ICFs ()`: returns the ids of registered ICFs. This can be leveraged by the runtime to discover the set of available ICFs on a given machine.
- `ic_create()`: creates an IC instance. In the case of traditional process and thread abstractions, the implementation of this function is similar to the `clone()` syscall in Linux.
- `ic_switch_in()` and `ic_switch_out()`: invoked by the OS scheduler[4] when an execution flow (associated to a given IC) must be scheduled in/out on a CPU. These functions provide the context switching mechanism that is suitable for the hardware/software isolation technique(s) employed by the IC (e.g., additional hardware registers to be saved and restored).
- `ic_dump()` and `ic_restore()`: respectively checkpoint and restore an IC. A checkpoint captures the full state (data and execution state) encapsulated in a given IC, which is useful for fault tolerance and also for migration.

The VIC interface defines a set of callbacks that ICFs can use to register functions implementing the relationships between distinct ICFs:

- `register_com`(id_1, id_2): registers a communication mechanism that is compatible for interactions between a pair of distinct IC types.

[3] In Figure 1, the `vicuser` driver and `libvicuser` components occupy roles that are analogous to the ones of the FUSE (*Filesystem in Userspace*) components in the VFS architecture: `fuse` driver and `libfuse`.

[4] Note that the design of xOS is also compatible with hierarchical scheduling. For example, the secure container abstraction provided by Scone [4] leverages M:N threading.

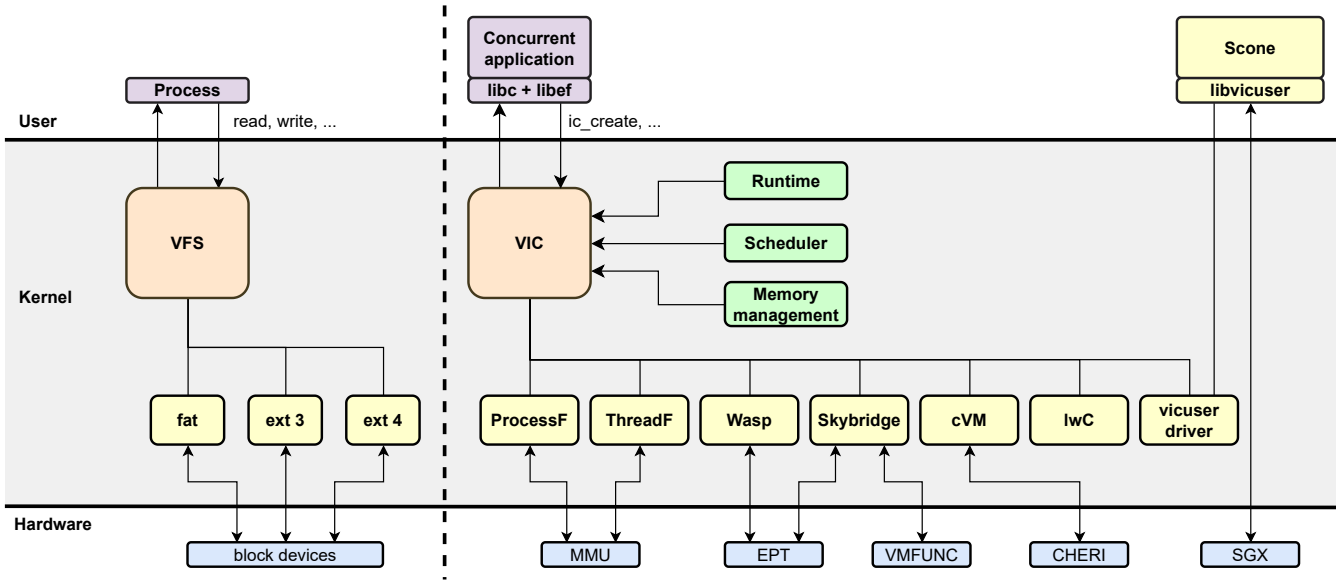

Figure 1: Overview of the xOS layer abstractions (on the right), compared with the existing Virtual File System design of Linux (on the left). ProcessF, ThreadF, Wasp, Scone, etc., are all isolation context factories.

- $\mathrm{mutate}(id_1, id_2)$: transforms an IC instance of (ICF) type id_1 into an an IC of type id_2. This function is used by the runtime (see §2.3). Since it is unrealistic to expect that system developers will provide an implementation of this function for every combination of source and destination ICF types, xOS uses a fallback strategy based on a "pivot" ICF type (the traditional process abstraction): each ICF must provide at least a pair of functions enabling to transform a IC into a process and vice versa.

Proposition 3: *xOS moves Process and Thread abstractions outside of the system core. They are instead registered in xOS as ICFs.*

It is worth noticing that the design of xOS supports *nesting*, i.e., a hierarchical grouping of EFs and their corresponding ICs. This principle allows combining several ICFs in order to simplify the construction and deployment of applications featuring multiple levels of isolation, e.g., using containers within a virtual machine — a common need on public cloud platforms. In the case of nesting, a control structure configured for each level of the hierarchy (akin to the VMCS [9], whose flags specify the causes of a guest virtual machine's exits) allows defining the events that require the intervention of the next (upper) level. However, we note that supporting nesting in a general way introduces many open challenges, including both functional and performance concerns, among which: (i) the identification of potential incompatibilities preventing the encapsulation of a given isolation abstraction within another one, and (ii) the limitations of many hardware-assisted isolation techniques with respect to multiple levels of nesting (and the overhead/complexity required to overcome them via emulation).

2.2 Programming Model

To help developers compartmentalize their applications, the programming model of xOS allows them to define *Execution Flows* (EF) and communication channels between them. An EF corresponds to a concurrent and preemptible task. Like a thread, it encapsulates an independent flow of control. Unlike a thread, it does not make any assumption about the degree of resource sharing (or lack thereof) with other flows it interacts with, including memory regions and system objects (e.g., open file descriptors).

The programming interface exposed by xOS to application developers for the manipulation of EFs is quite similar to a traditional threading API, yet with three notable exceptions, discussed below: (i) flow creation, (ii) flow migration, and (iii) inter-flow communication.

When instantiating an EF, a developer may indicate the set of acceptable ICFs that can be used to provide an environment to host and run this flow, or she can instead delegate this burden to xOS, which offers a high-level specification of the isolation and performance requirements (like with FlexOS [18]). In the latter case, the xOS runtime will choose

the most appropriate ICF according to the current deployment context (the workload, the availability of the ICF's hardware components, etc.).

An EF can be successively mapped to several IC types during its lifetime (thanks to the dump, restore and mutate facilities described in §2.1). This enables the dynamic reconfiguration of the isolation and communication techniques used by an EF, in-place or upon the migration of the flow to another location (e.g., another CPU or physical machine). For example, if an application encapsulated in a VM is migrated to an empty node (thus, with fewer risks of attacks), an OS policy may trigger a reconfiguration of the isolation technique (e.g., replacing the VM with a container) in order to improve performance.

The programming model of xOS imposes an explicit message-based communication between EFs, via unidirectional or bidirectional communication channels, which abstracts away differences between local and remote interactions. More precisely, xOS provides two communication APIs to manage interactions between EFs. The first one, partially inspired by Fable [27] is fairly low level; it aims at maximum efficiency (e.g., supporting optimized buffer allocation and ownership management) at the expense of some complexity. The second one is a socket-like interface, and is aimed at facilitating the porting of legacy applications.

The choice of a message-based interface may raise concerns about application performance and development complexity. Although we do not yet have sufficient experimental results to validate this important design choice, we believe that these potential issues can be significantly mitigated. First, regarding performance, our communication APIs do not preclude some IC implementations from internally leveraging shared memory flows (similarly to the optimizations introduced in some implementations of the multikernel model [5]), possibly with limited or zero data copies. Second, we observe that many multi-thread or multi-process applications leveraging shared-memory tend to rely on a limited number of high-level programming patterns (e.g., producer-consumer, remote procedure call, map-reduce, etc.) for which we can provide reusable and optimized implementations.

2.3 Runtime

The runtime chooses the appropriate type of IC for each EF based on four inputs: the developer indications, the hardware that currently hosts the EFs, the availability of migration functions for involved ICs, and the availability of communication functions between ICs. The runtime intervenes each time an EF is relocated/migrated (e.g., by a cloud orchestrator). The choice of the IC is guided by performance, while respecting the isolation requirements expressed by the developer/administrator.

2.4 Kernel Booting

When the kernel boots, it creates a set of kernel tasks, including the init one. xOS allows the execution of kernel tasks in different isolation envelopes (i.e., using different ICs), enabling compartmentalization within the kernel. We introduce a kernel isolation descriptor table (KIDT) to do so. KIDT allows for specifying the kernel tasks to be compartmentalized and the corresponding ICFs to be used for each compartment. This configuration can be coarse-grained (for a group of kernel tasks such as interrupt handlers) or fine-grained (for specific individual tasks such as kswapd).

At the end of the kernel boot, the first user task, init, starts. The list of usable ICFs for init is provided as a boot argument. The kernel opens any required communication channel between itself and the init EF.

init is in charge of starting the default user applications specified in the /etc/rc* directories in Linux. Say init creates a shell (e.g. a slightly modified version of bash or zsh), then any further EF will be started from this shell.

2.5 Launching User Applications

xOS keeps compatibility with legacy software while allowing new software to enjoy the full benefits of new abstractions, as explained below.

Legacy applications. They keep using the process and thread abstractions, which xOS also implements as ICFs. To run these without modification, we rely on LD_PRELOAD to shim an adaptor library that replaces functions such as fork(), exit(), etc., by their EF management counterpart. In particular, the shim library sets up an environment such that the execution flows of a legacy application can transparently interact via shared memory. Note however that legacy applications cannot benefit from the advanced features of xOS, i.e., transparent (static/dynamic) reconfiguration of isolation contexts and communication channels.

Applications based on new IC models. The shell creates new EFs as legacy applications. Only EFs created from the init shell can use a larger set of ICFs through explicit calls to EF_create(). Therefore, developers must write programs that first run in legacy mode, from which they start the rest of the application using other ICFs.

3 PRELIMINARY PROTOTYPE

xOS is still in an early stage of development. As a first milestone, we are focusing on the programming model. We currently only generate code that targets a vanilla Linux OS. The implementation of the OS kernel and the dynamic adaptation features will be tackled in the next phase.

Our current prototype consists in a toolchain (implemented atop Coccinelle [3, 16]) that takes as input a C program based on our generic IC and EF APIs. The output is a set variants of the applications, which only differ in the chosen isolation abstraction and/or inter-flow communication channel. The tool currently supports two ICFs (traditional thread and process abstractions) and four different communication mechanisms (Internet sockets, Unix domain sockets, pipes, and shared memory).

The current version of the toolchain imposes some constraints on the source code organization (e.g., the main routine for each EF must be implemented in its own file). In the next version, we intend to lift these restrictions and introduce support for additional types of isolation mechanisms (virtual machines, containers, and virtines [28]).

4 RELATED WORK

FlexOS [18] is a modular unikernel system allowing to postpone the decisions regarding the isolation strategy (i.e., the granularity of the isolation boundaries and the corresponding hardware/software mechanisms chosen to enforce them) until compilation/deployment time, instead of imposing a choice at design time. The FlexOS toolkit also includes an exploration technique to help users navigate the large set of possible configurations, with respect to the trade-offs between performance and safety/security. The xOS vision shares many motivations with FlexOS but has two main differences: (i) xOS aims at building a general-purpose host OS, supporting multiple concurrent applications and tenants; (ii) xOS aims at providing an extreme flexibility for the choice of isolation and communication mechanisms, until the launch time of an application, and possibly even with dynamic reconfiguration. Besides, we envision to reuse some high-level aspects of FlexOS for xOS, notably regarding the techniques for the exploration of the configuration space and the expression of high-level preferences/constraints (e.g., regarding the performance budget).

Smith et al. describe Fable [27], the blueprint of a system allowing to dynamically reconfigure the inter-process communication channels used by an application and automating the selection of the channel types, in order to improve performance according to the deployment constraints, the workload properties and the operating conditions. xOS aims at addressing a superset of the problem, which also encompasses the selection of adequate isolation techniques. We are currently studying how some aspects of the efficient communication interface proposed by the authors of Fable can be leveraged in the context of xOS.

Service Weaver (SW) [8] is a recent component-based framework (including a programming model and a runtime) for cloud applications. It allows decoupling the application development from the deployment concerns (e.g., choosing between a monolithic versus a microservice architecture), including the decision regarding the isolation and distribution boundaries between components. SW and xOS share a number of goals, yet at different and complementary levels of the software stack: the former focuses on high-level and distributed aspects (e.g., optimized protocols for data serialization and atomic rollouts) whereas the latter is targeting low-level questions (e.g., regarding OS task management and fine-grained isolation within each service).

Nu [23] and its successor Quicksand [22] are systems designed for *fungibility*, i.e., the ability to split a logical process into subparts (*proclets*) that can be dynamically distributed/migrated in a very granular way (in space and time) over the different servers of a datacenter, with the overall goals of improving elasticity and resource efficiency. The inter-flow communication interface used in xOS, which does not expose shared memory, has some similarities with the one of Nu. Besides, We believe that the core abstractions (EFs and ICs) proposed by xOS provide a good substrate to implement migratable and fungible microtasks like Nu's *proclets*.

5 CONCLUSION

In this paper, we make the observation that OS designs must evolve to accommodate the growing need for isolation in modern applications and platforms. This need is confirmed by the plethora of isolation and communication mechanisms proposed over the last few years. OSes should move away from tight coupling with the process and thread abstractions and welcome the addition of arbitrary isolation and communication mechanisms, as they do for drivers and file systems: this is the aim of xOS, which addresses the heterogeneity of workloads and architectures by giving the OS the ability to select the appropriate isolation layer for each application at runtime. However, this selection can still be done at the user's discretion if she wishes (e.g., an experimented developer who knows the internal mechanics of her application and exactly what level of isolation it requires). xOS is meant to support legacy applications based on the classical process and thread model, as well as software with more advanced compartmentalization needs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. Nel Navou and Dr. Thomas Messi (both from the University of Yaounde I, Cameroon) contributed to preliminary experiments. Maxime Just (ENS Lyon) contributed to an early version of the prototype. This work has been partially funded by: the CNRS MLNS2 International Research Project (IRP), the ANR Scalevisor ANR-18-CE25-0016 project, and a LIG "Emergence" grant.

REFERENCES

[1] [n. d.]. ARM Security Technology - Building a Secure System using TrustZone Technology. https://www.amd.com/en/processors/amd-secure-encrypted-virtualization. ([n. d.]).

[2] 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/en/processors/amd-secure-encrypted-virtualization. (2020).

[3] 2023. [Coccinelle Home Page]. https://coccinelle.gitlabpages.inria.fr/website/ (Accessed: 2023-06-01). (2023).

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[6] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, USA, 335–348.

[7] Jiahao Chen, Dingji Li, Zeyu Mi, Yuxuan Liu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. DuVisor: A User-level Hypervisor Through Delegated Virtualization. (Jan. 2022). arXiv:cs/2201.09652

[8] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards Modern Development of Cloud Applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23)*. Association for Computing Machinery, New York, NY, USA, 110–117. https://doi.org/10.1145/3593856.3595909

[9] Intel. 2022. *Intel 64 and IA-32 Architectures Software Developper's Manual*.

[10] Intel. April 2022. *Intel Software Developer's Manual - Chapter 34, Introduction to Intel Software Guard Extensions*. Vol. 3D.

[11] Intel. April 2022. *Intel Software Developer's Manual - Section 2-20: Introduction to Virtual Machine Extensions*. Vol. 1.

[12] Intel. April 2022. *Intel Software Developer's Manual - Vol. 3C - Section 25-5.6: VM Functions*.

[13] Intel. April 2022. *Intel Software Developer's Manual - Vol. 3C - Section 29-1: APIC Virtualization and Virtual Interrupts*.

[14] Intel. April 2022. *Intel Software Developer's Manual - Vol. 3C - Section 29-1: The Extended Page Table Mechanism (EPT)*.

[15] Yuzhuo Jing and Peng Huang. 2022. Operating System Support for Safe and Efficient Auxiliary Execution. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 633–648. https://www.usenix.org/conference/osdi22/presentation/jing

[16] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 601–614. https://www.usenix.org/conference/atc18/presentation/lawall

[17] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2022. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *Proceedings of 30th Network and Distributed System Security (NDSS'23) (NDSS'23)*. Internet Society, United States.

[18] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 467–482. https://doi.org/10.1145/3503222.3507759

[19] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[20] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[21] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3302424.3303946

[22] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. 2023. Unleashing True Utility Computing with Quicksand. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23)*. Association for Computing Machinery, New York, NY, USA, 196–205. https://doi.org/10.1145/3593856.3595893

[23] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

[24] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2022. {CAP-VMs}: {Capability-Based} Isolation and Sharing in the Cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 597–612.

[25] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 546–558. https://doi.org/10.1145/3445814.3446731

[26] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient in-Process Isolation for RISC-V and X86. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, 1677–1694.

[27] Steven Smith, Anil Madhavapeddy, Christopher Smowton, Malte Schwarzkopf, Richard Mortier, Robert M Watson, and Steven Hand. 2012. The Case for Reconfigurable I/O Channels. In *RESoLVE Workshop (Runtime Environments, Systems, Layering and Virtualized Environments)*.

[28] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. 2022. Isolating Functions at the Hardware Limit with Virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery,

New York, NY, USA, 644–662. https://doi.org/10.1145/3492321.3519553

[29] Jonathan Woodruff, Robert N.M. 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. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 457–468. https://doi.org/10.1145/2678373.2665740