Creating Trust by Abolishing Hierarchies

Charly Castes*
EPFL, Switzerland

Adrien Ghosn*
Microsoft, UK

Neelu S. Kalani
EPFL, Switzerland

Yuchen Qian
EPFL, Switzerland

Marios Kogias
Imperial College London &
Microsoft, UK

Mathias Payer
EPFL, Switzerland

Edouard Bugnion
EPFL, Switzerland

ABSTRACT

Software is going through a trust crisis. Privileged code is no longer trusted and processes insufficiently protect user code from unverified libraries. While usually treated separately, confidential computing and program compartmentalization are both symptoms of the same problem, deeply rooted in hierarchical commodity systems: privileged software's monopoly over isolation.

This paper proposes a separation of powers: to decouple trust and isolation from privilege hierarchies. It introduces an *isolation monitor*, which delivers verifiable isolation, confidentiality, and integrity to all software, independent of existing system abstractions and privilege hierarchies. Tyche, our prototype isolation monitor, runs on commodity hardware without relying on complex and emerging hardware security extensions. It enables any software component to create, compose, and nest isolation abstractions, including user and kernel sandboxes, enclaves, as well as confidential virtual machines.

ACM Reference Format:
Charly Castes, Adrien Ghosn, Neelu S. Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. 2023. Creating Trust by Abolishing Hierarchies. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595900

*co-equal first author.

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595900

1 INTRODUCTION

Trust is gone. In 1984, Ken Thompson warned "You can't trust code you did not totally create yourself" [50]. Today, developers only create a tiny fraction of the software stack. User applications use thousands of unverified libraries [43, 44] and give them unrestricted access to their address space. Applications further have no choice but to trust commodity systems, *i.e.,* operating systems and hypervisors written by thousands of contributors, with an ever-increasing code base [31, 36], and executing in the Cloud under the control of third parties.

Modern application software needs to minimize its trusted computing base (TCB) and control how its sensitive data and resources are exposed to other software components. For example, an application should be able to isolate libraries coming from untrusted third parties; a kernel should be able to isolate untrusted device drivers; applications need isolation from potentially compromised operating systems or hypervisors.

Modern software addresses these challenges through a combination of compartmentalization, *i.e.,* intra-program isolation, and confidential computing, *i.e.,* isolation from more privileged code. Industry, including hardware manufacturers, and Academia have so far treated them separately. They provide point solutions (1) tied to privilege levels or system abstractions (2) that are hard to combine [7] or do not nest [46]. Compartmentalization mechanisms focus on either user [10, 19, 22, 39, 53, 55] or kernel [16, 37, 49] code, and are often incompatible with confidential computing abstractions tied to either processes [5, 8, 11–13, 17, 20, 23, 30, 35, 38, 40, 57] or virtual machines [3, 4, 29, 32, 58]. As software components combine and nest, so must isolation mechanisms.

Previous work relies on separate, incompatible mechanisms to implement compartmentalization and confidential computing. The heterogeneous nature of those mechanisms and their ties to privilege levels make them hard to nest or combine. An ideal solution should provide configurable policies that can be recursively applied, and enforced by a

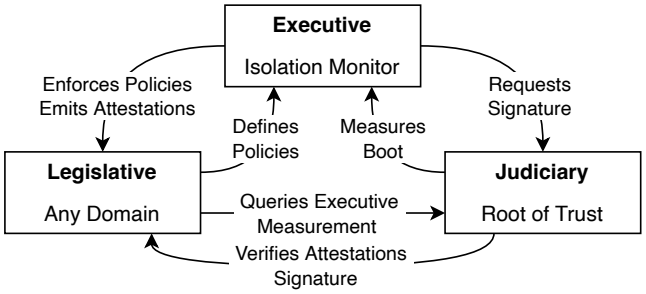

Figure 1: The separation of powers

mechanism supporting recursive-nested isolation, in a system that can be fully attested and inspected for correctness.

This paper argues that challenges in compartmentalization and confidential computing stem from a common system design limitation: commodity systems' current monopoly over isolation. The system is the sole provider of isolation and does so through processes and virtual machines. These abstractions are coarse-grained, conflate trust and privilege, and are thus unfit for modern software isolation.

This monopoly can be expressed as an analogy with politics: commodity systems concentrate (1) legislative power, by restricting isolation policies based on privileges, (2) executive power, by having exclusive access to hardware enforcement mechanisms [14, 28, 45, 52], and (3) have no judiciary oversight, *i.e.,* provide no verifiable guarantees that isolation is correctly enforced.

Inspired by French philosopher Montesquieu [41], we propose in Figure 1 a separation of powers to abolish commodity systems' monopoly on isolation. We believe that a complete separation of powers would address the software trust crisis by providing flexible and verifiable isolation, orthogonal to privilege levels and existing system abstractions.

We introduce an *isolation monitor* to enforce (executive) policies defined by any software (legislative) and guarantee system-wide, remotely-verifiable invariants (judiciary). This separation of powers unifies compartmentalization and confidential computing, supports more complex trust models, and can be retrofitted into commodity systems with minimal disruption.

We present Tyche, a prototype implementation of an isolation monitor, designed to be formally verifiable, with a unified isolation API that allows to create, combine, and nest various isolation abstractions, including sandboxes, enclaves, confidential virtual machines, and more. With an early prototype on x86_64 and RISC-V, Tyche demonstrates that our solution is not only possible on commodity hardware, without using complex hardware security extensions [3, 4, 7, 29, 30], but also allows for the exploration of isolation policies beyond the ones provided by hardware manufacturers and previous work.

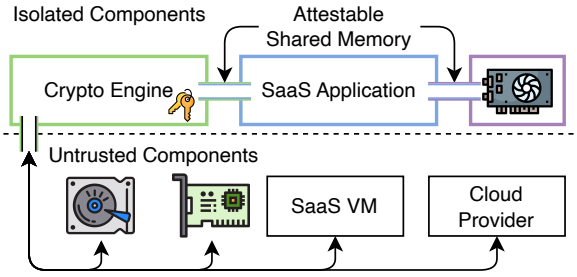

Figure 2: Confidential processing of data through an untrusted SaaS application. Isolated components (enclaves and GPU) are restricted to solely access their own (confidential) and explicitly shared memory.

2 THE PRIVILEGE HIERARCHY

Verifiable isolation of a software component is achieved by the combination of three elements: *policies*, *enforcement*, and *attestation*. This paper draws a parallel between these three elements and the three political powers: legislative, executive, and judiciary. This analogy explains how commodity systems create rigid hierarchies that conflate trust and privileges, but also suggests how to provide flexible verifiable isolation on commodity hardware.

2.1 The Three Powers

In computer systems, the *legislative power* defines the isolation policies for a given piece of software: (1) the resources, *i.e.,* memory, CPU cores, and PCI devices, available to the software, (2) their corresponding access rights, but also (3) controlled sharing: isolation conditions for resources to be shared with other components.

Figure 2 illustrates how crucial controlled sharing is to allow mutually distrustful entities to collaborate. The customer is only willing to process sensitive data through the SaaS application under the guarantee that its data cannot be leaked. For this purpose, the SaaS application is isolated from the rest of the system with the exception of shared memory regions with a similarly isolated GPU and a confidential crypto engine, *i.e.,* an encryption enclave. After attesting the identity of the crypto engine and the proper isolation and shared memory configuration of both the SaaS application and GPU, the customer can provision the crypto engine with its key to ensure encryption of all outgoing traffic.

The *executive power* enforces isolation: it restricts access to the machine's resources. On commodity hardware, enforcement is the prerogative of privileged code that derives its power from *privileged hardware instructions* [45]. Privileged instructions configure *access control mechanisms*, such as segments [14, 45], page-tables [28, 52], and I/O-MMU [28] for memory, to restrict unprivileged user code's access to the machine's resources.

The *judiciary power* ensures that policies are correctly enforced. It provides oversight of the executive power, preventing privileged code from arbitrarily violating isolation policies. Judiciary power can take different forms, but usually comprises a root of trust and a way to derive a chain of trust. In Figure 2, the judiciary power provides attestable verifiable assurance that the SaaS application and GPU are completely isolated and only share memory with the crypto engine throughout the entire execution.

2.2 The Monopoly on Isolation

Commodity systems have a *monopoly* on isolation: they concentrate legislative (policies) and executive (enforcement) powers in a single entity without judiciary oversight. This concentration of powers creates a rigid trust hierarchy tied to privileges and impedes their ability to satisfy modern software's need for isolation.

First, the lack of attestable verifiable guarantees leaves no choice but to blindly trust *all* privileged code which allows arbitrary modifications to access control mechanisms by *any* privileged code. As such systems often comprise millions of lines of code [31, 36] and are trusted based on reputation rather than verifiable correctness, they cannot provably provide confidential guarantees or compartmentalize privileged code, even when attested.

Second, commodity systems overly restrict the set of supported isolation policies, violating the principle of separation between mechanisms and policies [34]. They only provide processes and virtual machines, two coarse-grain abstractions with rigid trust models.

Processes mirror hardware's privilege hierarchy, *i.e.,* they protect privileged code from user code, but not the other way around. Processes are meant to isolate full programs, not individual libraries. As a result, developers must either extend their trust to thousands of unverified libraries [43, 44] or isolate them in separate processes, with all associated overheads in creation, synchronization, and management. Meanwhile, privileged code can easily bypass process isolation and thus has no choice but to run untrusted drivers in user mode at the cost of extra context switches for privileged operations.

Virtual machines duplicate hardware privilege levels and grant full control to virtual-privileged code over virtual-user software [45]. This creates a rigid trust hierarchy that forces software to blindly trust all intermediate privileged levels and leads to an uncontrolled explosion of the TCB.

Third, commodity systems isolation abstractions do not support verifiable controlled sharing. They are meant to virtualize and multiplex physical resources and as such, do not provide full visibility into how these are shared among software components, and more importantly, do not provide proof that they are not shared with unauthorized ones.

2.3 Breaking the Monopoly

Montesquieu suggests a separation of the three powers into independent branches so that no single entity can abuse its power [41]. Solutions in both compartmentalization and confidential computing can be interpreted as partial attempts to reduce commodity systems' monopoly on isolation through a limited redistribution of powers.

Compartmentalization: Compartmentalization is the ability to let any piece of software divide itself into isolated subcomponents with different access rights to the program's resources. It relies on program-specific knowledge to derive isolation policies following the principle of least privilege, *e.g.,* to enable the protection of applications from untrusted libraries [19, 22, 53] or operating systems from kernel drivers [15, 49].

Compartmentalization solutions typically involve two privilege levels: the software that needs to be compartmentalized dictates policies and relies on more privileged code to enforce them. Policies are expressed via new system abstractions [10, 24, 39], compiler instrumentation [21], or programming constructs [9, 19, 22, 53] and enforced by more privileged software with existing or new hardware mechanisms [27]. For example, operating system code relies on a hypervisor to isolate untrusted system libraries [16, 37], and user code relies on OS abstractions, such as LWC [10, 39] or SMV [24], to create the desired isolated compartments.

Compartmentalization solutions attenuate commodity systems' monopoly on isolation. They implement a partial separation of legislative (policies) and executive (enforcement) powers. Unfortunately, they duplicate mechanisms by handling kernel and user compartmentalization separately, and preserve the conflation of trust and privileges.

Confidential computing: Confidential computing is the ability to construct attestable trusted execution environments (TEEs) that preserve a program's confidentiality and integrity, even in the presence of compromised privileged software. This allows a drastic reduction of a program's TCB.

Confidential computing solutions build a root of trust that (1) prevents illegal access to TEEs and (2) provides a remotely-verifiable attestation that the TEE is correctly configured and isolated from the rest of the system. The root of trust is a third party, separate from the untrusted OS or hypervisor, implemented in hardware [3, 4, 29, 30] or software [13, 40, 57]. Hardware solutions use a mix of firmware and silicon to prevent illegal access to TEEs, while software solutions introduce a security monitor [6, 35, 40, 57] below the untrusted privileged software. Trust in the security monitor is derived from a combination of hardware measurement, *e.g.,* with a TPM [25, 51], and formal verification [17, 42] or

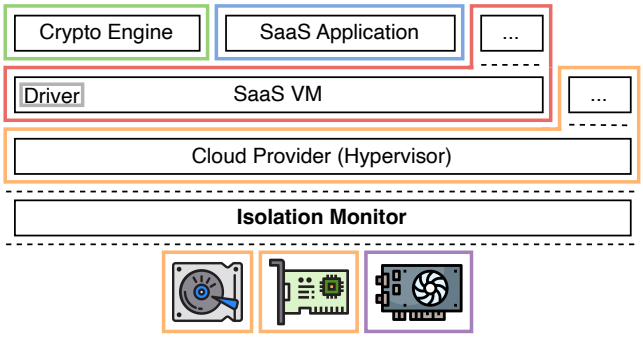

Figure 3: Deployment of Figure 2 on an isolation monitor. Black boxes denote traditional system abstractions (hypervisor, VM, and processes), while colored boxes represent trust domains.

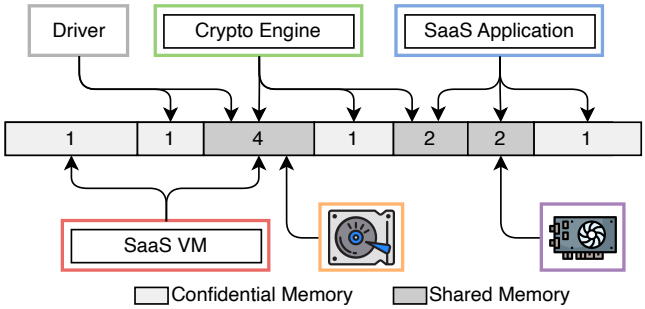

Figure 4: View of a subset of the physical memory for the deployment in Figure 3, with domain-to-regions mappings and regions reference counts.

code inspection [13, 23, 40, 57] of the monitor's implementation.

Confidential computing solutions implement judiciary oversight of isolation on traditional systems. They provide a measurement of isolation policies that apply to a TEE and prevent their violation by privileged code. Similar to compartmentalization, they however provide distinct complex [7] solutions tied to existing system abstractions, *i.e.,* enclaves and confidential VMs, that seem hard to combine and nest.

3 THE ISOLATION MONITOR

We introduce an *isolation monitor*, a security monitor that implements a *complete* separation of powers to democratize isolation on commodity hardware.

Figure 1 shows how the isolation monitor enables a complete separation of powers. The isolation monitor itself is only the executive branch (§3.3), exposing isolation via a simple yet expressive API for all software to specify their policies (legislative, §3.2), orthogonal to existing privilege levels and system abstractions. A third-party root-of-trust provides verifiable oversight of the system through attestable system-wide invariants (judiciary, §3.4).

3.1 Domains and resources

The isolation monitor provides a new abstraction to uniquely identify security contexts: *trust domains*. A trust domain is an identity associated with a set of access rights to physical resources, *i.e.,* memory, CPU cores, and PCI devices. Domains can be *sealed*, so that their resources cannot be extended or further shared with others. The monitor enforces the desired isolation, confidentiality, and integrity properties for each trust domain, without considering how protection is further implemented inside the domain itself.

Figure 3 depicts trust domains, independent of existing privilege levels and system abstractions, that provide the desired isolation guarantees (§2.1). They can be as large as a full confidential VMs (SaaS VM), as small as a program library or driver, or I/O domains running on devices with restricted access to main memory (*e.g.,* GPU), thus allowing fine-grained control over the TCB.

The isolation monitor enforces discretionary access control over resources. The monitor maintains a per-domain resource configuration that associates each resource with access rights, *i.e.,* a set of valid operations, and exposes a system-wide reference count, as shown in Figure 4. The reference count is maintained by the monitor to reflect the number of domains with access to the resource. It ensures attestable controlled sharing of resources. In Figure 3, the customer only shares the decrypted data with the SaaS application and GPU if their resources are either shared among themselves (ref. count 2) or exclusively owned (ref. count 1), *i.e.,* if they are sealed and do not share resources (*e.g.,* memory or registers) with the rest of the system.

The monitor mediates and validates all control transfers between domains. Domains have a fixed entry point and are only allowed to run on CPU cores or I/O devices that are part of their resource configuration.

3.2 Expressing policies via the API

Software running in any trust domain can access the isolation monitor API. The API is made of two parts: one part to create new trust domains, seal them, perform domain transitions, as well as enumerate and attest a domain's resources; another part for resource management policies.

The monitor draws inspiration from microkernels [33, 37], in that it tries to be minimal and exposes a flexible configuration of access policies via a narrow API. Unlike a microkernel, policies operate on physical name spaces (*e.g.,* memory, CPU cores), which (1) permit reasoning about sharing and exclusive ownership without having to consider aliasing, and (2) favors the implementation of higher-level abstractions, similar to the exokernel [16].

The monitor API provides operations for sharing or granting exclusive control over resources with a specified revocation policy. Both sharing and granting of resources are revocable operations, *i.e.,* the resource can be taken back from the domain. This keeps management code (OS or hypervisor) in control despite making policy configuration available to all software on the machine. A revocation policy specifies a "clean-up" operation, *e.g.,* zeroing-out memory or flushing CPU cache, that is guaranteed to execute upon revocation. Additionally, the monitor can produce a hash of domain configurations and selected initial resources (*e.g.,* memory content) for attestation purposes.

3.3 Hardware-based policy enforcement

The isolation monitor enforces policies. This entails (1) a direct (secured) communication channel between a domain and the monitor to configure policies and (2) the monitor's oversight of access control mechanisms to enforce them. Both requirements can be satisfied on major modern architectures (x86_64, ARM, RISC-V), either through virtualization or leveraging higher privilege modes, and without complex [7] new hardware security extensions [3, 4, 29, 30].

Previous work [40, 57] implemented security monitors on top of hardware support for virtualization [1, 52] or in programmable machine mode [35]. Memory virtualization provides a second level of page tables to enforce memory access control at page granularity while other resources, *e.g.,* devices, can be partitioned using SR-IOV and isolated using I/O-MMUs. Intel VT-x's VMCall instruction provides a direct communication channel to the monitor. On RISC-V, "machine mode" is the most privileged programmable execution level and can protect physical memory segments through PMP [47].

3.4 Trust through remote attestation

The isolation monitor needs (1) to be attested by a root-of-trust, (2) trusted to correctly enforce policies, and (3) supply verifiable attestations of system-wide invariants to third parties. The monitor should not accept invalid policies, arbitrarily reconfigure access control mechanisms, or misreport domain configurations, and must guarantee the ability to revoke resources while preventing information leakage.

First, a hardware root-of-trust, such as an industry-standard TPM [25, 51], measures the machine's boot process and provides a signed remotely-verifiable attestation that the machine is under the complete control of a specific monitor implementation.

Second, trust in the monitor is derived from the attestation by comparing the measurement to a known expected value that corresponds to an implementation that is either provided by a trusted supplier, is open-source and can be inspected and tested, or, ideally, provides strong correctness guarantees through model-checking or formal verification.

Third, the monitor, now trusted, implements a two-tier attestation protocol [40] to attest individual domains. A domain's attestation, signed by the monitor, enumerates its physical resources, their reference counts, and the measurement of selected memory regions. Resource enumeration and reference counts make sharing and communication paths between domains *explicit* and enables remotely-attestable controlled sharing, confidentiality, and integrity guarantees. For example, exclusive access to a resource (*i.e.,* a reference count of 1) coupled with an obfuscating revocation policy guarantees integrity (while in use) and confidentiality.

3.5 Comparison with existing designs

The design of the isolation monitor borrows from previous work [16, 18, 32, 35, 37, 40, 58]. It differs from monolithic operating systems and hypervisors in that it focuses on isolation and avoids resource management, device emulation, and does not expose high-level abstractions. The monitor does not choose resources to allocate to a domain, but rather validates allocation, *i.e.,* sharing, granting, and revocation operations.

The isolation monitor's design is very close to the microkernel one as it strives to be minimal and provides a tight but flexible API for resource management. An isolation monitor or microkernel is expected to be orders of magnitude smaller, *e.g.,* thousands of lines of code instead of millions, than a typical monolithic kernel or hypervisor. However, unlike microkernels, the isolation monitor's goal is not to replace monolithic kernels or hypervisors, but rather to execute at a higher privilege level, only manage physical names, and provide an isolation mechanism orthogonal to the ones provided by the OS. For example, the OS still provides the process abstraction, while the monitor transparently allows sub-compartments within a process.

Previous security monitor implementations [17, 35, 40], as well as hardware extensions for either confidential computing or compartmentalization, only provide a single high-level abstraction, tied to pre-existing systems one, *e.g.,* an enclave within a user process or a confidential virtual machine. The isolation monitor takes a holistic approach to isolation: it provides a single, unified API for both compartmentalization and confidential computing that supports arbitrary nesting.

4 THE TYCHE ISOLATION MONITOR

Tyche is our prototype isolation monitor, written in Rust, designed to be minimal (<10K LOC), formally verifiable, remotely attestable, and thus trusted. Tyche boots on bare metal and runs an unmodified Ubuntu distribution and Linux kernel as an initial domain.

Tyche comprises: (1) a platform-independent capability model that implements § 3.2's API, and (2) a platform-specific backend. The capability model allows software libraries to implement higher-level programming abstractions for isolation, such as sandboxes, enclaves, and confidential VMs, while the backend configures commodity hardware mechanisms (§ 3.3) to enforce the desired policies.

On Intel x86_64, Tyche is attested using TXT [25] and isolates domains with Intel VT-x [52] and I/O-MMUs. On RISC-V, it runs in machine mode and demonstrates the generality of our approach by relying on a more limited mechanism than virtualization: PMP [47]. PMP only supports a fixed number of segments, which requires a careful memory layout of trust domains and validation by the monitor. A more flexible segmentation mechanism could however appear on open platforms in the future [54], potentially in the form of CHERI [56] capabilities for physical memory and under exclusive control of machine mode.

4.1 Tyche's capabilities

The platform-independent capability model, which implements Tyche's API, is written in safe Rust [48], and meant to be formally verified. It defines a capability model for which *grant*, *share*, and *revoke* operations modify a tree structure that represents a capability's lineage, maintains per-resource reference counts, and facilitates cascading revocations, even in the presence of circular sharing.

Operations on capabilities are validated and translated into platform-specific hardware configurations by Tyche's backend. At the moment, Tyche's capabilities handle physical memory, PCI devices, and CPU cores. They define the access rights and revocation policies associated with the corresponding resource.

Tyche's capabilities safely expose hardware and additionally enrich it with new access rights and revocation policies. For example, transitions between domains are enabled by operations on domain and CPU capabilities. This allows domains to implement policies that mitigate side-channel attacks, *e.g.,* by ensuring exclusive access to a CPU core or revocation policies that flush micro-architectural state (caches) during a transition. We are also exploring how to extend capabilities to provide scheduling guarantees, cross-domain interrupt routing, and expose denial of service attacks, or accelerate existing operations with hardware, such as fast (100 cycles) domain transitions using VMFUNC [22] and hardware interrupt routing via remapping [28].

4.2 Building higher-level abstractions

Tyche's isolation API only provides the notion of trust domains tied to a set of resources. This differs from previous security monitor implementations that provide fixed programming abstractions, namely enclaves [13, 23, 40] or confidential VMs [57]. With Tyche, higher-level abstractions, including but not limited to sandboxes, enclaves, and confidential VMs, are implemented on top of the monitor's isolation API by libraries running within the trust domains.

Our prototype libtyche explores compartmentalization and confidential computing abstractions. The library loads an ELF binary as a domain using a manifest that describes which segments should run in which privilege ring, whether they are shared or confidential, and if their content is part of the attestation or not. The library further supports generating a binary's hash offline to be compared with the attestation provided by Tyche. Using libtyche, we prototyped user and kernel compartments, as well as Tyche-enclaves.

Tyche-enclaves present notable improvements over SGX ones [30]. First, they limit accidental leakage of information by requiring untrusted memory regions to be explicitly shared. Second, they allow virtual address reuse thus enabling an arbitrary layout and number of enclaves in the same process. Third, they support nesting and sharing among enclaves. Our enclaves can map libtyche in their domains to spawn nested enclaves, and share exclusively owned pages with them to create secured communication channels.

libtyche is still at an early development stage but we have plans to explore various system extensions, including sandboxing unsafe code downloads in the kernel [16], safely multiplexing (with and without SR-IOV) PCI devices, *e.g.,* GPUs, among TEEs, extending Linux Kernel-based Virtual Machine (KVM) with a Tyche backend for confidential VMs, providing RDMA support for Tyche-based TEEs running on separate machines, extend attestation to multi-domain deployments with the insurance that all communication paths are secured and attested, and building physical attack resistance with multi-key memory encryption technologies [2, 26].

5 CONCLUSION

The *isolation monitor* creates trust by breaking commodity systems' monopoly on isolation and abolishing trust hierarchies based on privileges. It enables the creation, nesting, and combination of confidential computing and compartmentalization abstractions. Our early prototype suggests the isolation monitor can be built with standard architectural support for virtualization or physical memory protection.

ACKNOWLEDGMENTS

The authors thank James R. Larus, our shepherd Donald E. Porter, and the anonymous reviewers for their feedback. This project is supported in part by the Microsoft-EPFL Joint Research Center and a VMware faculty award.

REFERENCES

[1] AMD. Secure virtual machine architecture reference manual, 2005.

[2] AMD. Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/, 2018.

[3] AMD. Sev-snp: Strengthening vm isolation with integrity protection and more. *White Paper, January* (2020).

[4] ARM. Building a secure system using trustzone technology. *White Paper, April* (2009).

[5] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., Goltzsche, D., Eyers, D. M., Kapitza, R., Pietzuch, P. R., and Fetzer, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)* (2016), pp. 689–703.

[6] Bahmani, R., Brasser, F., Dessouky, G., Jauernig, P., Klimmek, M., Sadeghi, A.-R., and Stapf, E. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proceedings of the 30th USENIX Security Symposium* (2021), pp. 1073–1090.

[7] Baumann, A. Hardware is the new Software. In *Proceedings of The 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)* (2017), pp. 132–137.

[8] Baumann, A., Peinado, M., and Hunt, G. C. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst. 33*, 3 (2015), 8:1–8:26.

[9] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. J. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 267–284.

[10] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)* (2008), pp. 309–322.

[11] che Tsai, C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D., and Porter, D. E. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 9:1–9:14.

[12] che Tsai, C., Porter, D. E., and Vij, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017), pp. 645–658.

[13] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J. S., and Ports, D. R. K. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)* (2008), pp. 2–13.

[14] Corbató, F. J., and Vyssotsky, V. A. Introduction and overview of the multics system. In *AFIPS Fall Joint Computing Conference (1)* (1965), pp. 185–196.

[15] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., and Adve, V. S. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)* (2015), pp. 191–206.

[16] Engler, D. R., Kaashoek, M. F., and Jr., J. W. O. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 251–266.

[17] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 287–305.

[18] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI)* (1996), pp. 137–151.

[19] Ghosn, A., Kogias, M., Payer, M., Larus, J. R., and Bugnion, E. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)* (2021), pp. 255–267.

[20] Ghosn, A., Larus, J. R., and Bugnion, E. Secured Routines: Language-based Construction of Trusted Execution Environments. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)* (2019), pp. 571–586.

[21] Gudka, K., Watson, R. N. M., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Marinos, I., Neumann, P. G., and Richardson, A. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2015), pp. 1016–1031.

[22] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K., and Marty, M. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)* (2019), pp. 489–504.

[23] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and Witchel, E. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)* (2013), pp. 265–278.

[24] Hsu, T. C.-H., Hoffman, K. J., Eugster, P., and Payer, M. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 393–405.

[25] Intel. Trusted execution technology. https://www.intel.com/content/www/us/en/developer/articles/tool/intel-trusted-execution-technology.html, 2014.

[26] Intel. Multi-key total memory encryption. https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/002/intel-multi-key-total-memory-encryption/, 2017.

[27] Intel. Intel memory protection keys (intel mpk). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2020.

[28] Intel. Intel®64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2022.

[29] Intel. Architecture specification: Intel trust domain extensions (intel tdx) module. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1eas.pdf, 2023.

[30] Intel. Intel software guard extensions (intel sgx). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2023.

[31] Israeli, A., and Feitelson, D. G. The Linux kernel as a case study in software evolution. *J. Syst. Softw. 83*, 3 (2010), 485–501.

[32] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. NoHype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)* (2010), pp. 350–361.

[33] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D. A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 207–220.

[34] Lampson, B. W., and Sturgis, H. E. Reflections on an Operating System Design. *Commun. ACM 19*, 5 (1976), 251–265.

[35] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., and Song, D. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the 2020 EuroSys Conference* (2020), pp. 38:1–38:16.

[36] Lehman, M. M., and Parr, F. N. Program Evolution and Its Impact on Software Engineering. In *Proceedings of the 2nd International Conference on Software Engineering (ISCE)* (1976), pp. 350–357.

[37] Liedtke, J. On micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 237–250.

[38] Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.-L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D. M., Kapitza, R., Fetzer, C., and Pietzuch, P. R. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017), pp. 285–298.

[39] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)* (2016), pp. 49–64.

[40] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy* (2010), pp. 143–158.

[41] Montesquieu. *The Spirit of Laws (De l'esprit des lois).* 1748.

[42] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)* (2019), pp. 225–242.

[43] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S. V., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2012), pp. 736–747.

[44] Nikola Đuza. JavaScript Growing Pains: From 0 to 13,000 Dependencies. https://blog.appsignal.com/2020/05/14/javascript-growing-pains-from-0-to-13000-dependencies.html, 2020.

[45] Popek, G. J., and Goldberg, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[46] Project Oak. Oak (SEV). https://github.com/project-oak/oak, 2019.

[47] RISC-V Foundation. RISC-V SBI specification. https://github.com/riscv-non-isa/riscv-sbi-doc, 2023.

[48] Rust Foundation. The rustonomicon - meet safe and unsafe. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html, 2023.

[49] Swift, M. M., Martin, S., Levy, H. M., and Eggers, S. J. Nooks: an architecture for reliable device drivers. In *ACM SIGOPS European Workshop* (2002), pp. 102–107.

[50] Thompson, K. Reflections on Trusting Trust. *Commun. ACM 27*, 8 (1984), 761–763.

[51] Trusted Computing Group. Trusted Platform Module (TPM) – ISO/IEC 11889. https://www.iso.org/standard/66510.html, 2015.

[52] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kägi, A., Leung, F. H., and Smith, L. Intel Virtualization Technology. *Computer 38*, 5 (2005), 48–56.

[53] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium* (2019), pp. 1221–1238.

[54] Van Strydonck, T., Noorman, J., Jackson, J., Dias, L., Vanderstraeten, R., Oswald, D., Piessens, F., and Devriese, D. Cheritree: Flexible enclaves on capability machines. In *EuroS&P-8th IEEE European Symposium on Security and Privacy* (2023), IEEE.

[55] Vilanova, L., Ben-Yehuda, M., Navarro, N., Etsion, Y., and Valero, M. CODOMs: Protecting software with Code-centric memory Domains. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 469–480.

[56] Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R. M., and Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 457–468.

[57] Zhang, F., Chen, J., Chen, H., and Zang, B. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011), pp. 203–216.

[58] Zhou, Z., Shan, Y., Cui, W., Ge, X., Peinado, M., and Baumann, A. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *Proceedings of the 17th Symposium on Operating System Design and Implementation (OSDI)* (2023).