Endoprocess: Programmable and Extensible Subprocess Isolation

Fangfei Yang
fy15@rice.edu
Rice University
Houston, Texas, USA

Weijie Huang
wh31@rice.edu
Rice University
Houston, Texas, USA

Kelly Kaoudis
hi@kellykaoud.is
Trail of Bits
New York, New York
USA

Anjo Lucas Vahldiek-Oberwagner
anjovahldiek@gmail.com
Intel Labs
Hillsboro, Oregon
USA

Nathan Dautenhahn
ndd@rice.edu
Rice University
Houston, Texas, USA

ABSTRACT

Modern applications combine multiple components into single processes, leading to complex tradeoffs between isolation, performance, and programmability. We present the Endoprocess, a unique, microkernel-based approach for protection within process spaces. An endoprocess safely multiplexes process resources by exporting a low-level abstraction, the *subprocess*, that is transparently overlaid on existing process interfaces (like mmap, mprotect, etc), and provides extensibility and programmability through custom application-layer modules. We report experimental results of an initial prototype and highlight several application domains. Overall, the endoprocess presents a path for protection within processes while remaining compatible with existing OS abstractions and multiplexing them in a secure and extensible way.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**.

KEYWORDS

Language based security, Access control systems, Subprocess isolation, Compartmentalization, OS virtualization

ACM Reference Format:
Fangfei Yang, Weijie Huang, Kelly Kaoudis, Anjo Lucas Vahldiek-Oberwagner, and Nathan Dautenhahn. 2023. Endoprocess: Programmable and Extensible Subprocess Isolation. In *New Security Paradigms Workshop (NSPW '23), September 18–21, 2023, Segovia, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3633500.3633507

1 INTRODUCTION

The Process is no longer a meaningful way to isolate and manage resources within modern applications. Adopted by all early operating system designs [12]—micro, mono, and Multics alike—today's process environment is a security nightmare. Most processes, even those running applications written in safe languages, contain unsafe components that lead to bugs allowing attackers to leak data, modify runtime state, or commandeer execution [1, 6, 29, 42, 54].

NSPW '23, September 18–21, 2023, Segovia, Spain

ACM ISBN 979-8-4007-1620-1/23/09.
https://doi.org/10.1145/3633500.3633507

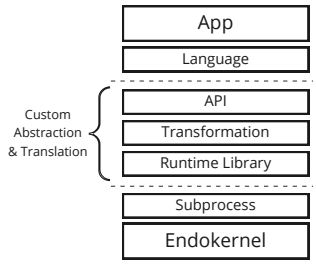

Figure 1: **The Endoprocess Approach. An endoprocess is a process retrofitted with a nested monitor, the endokernel, that exports the subprocess. The subprocess can be used directly or safely extended outside the endokernel to support customized in-process protection.**

As well, supply chain attacks have become a serious problem as programmers prioritize quickly adding features over security [8, 24, 64], including in *safe languages* like Rust [21]. Beyond over-privileged and buggy runtimes, emerging software paradigms combine mutually distrusting components into single process spaces without isolation, such as browsers, serverless [16, 50], heterogeneous language runtimes [19, 40], and user-space operating systems [28, 34, 56].

Since the modern process is a multiprogramming [12] runtime, there is need for *subprocess* access control protections. While process-level access control systems like SELinux [52] could provide the right abstraction, the mechanism cannot isolate within a process because it cannot distinguish between subprocess entities. Extending the OS to include a subprocess is untenable. Modifying the kernel to support subprocess isolation would tightly couple the application and OS by pushing application-level concerns and abstractions into kernelspace [15], could make the OS less secure [55], and would introduce costly context-switching overheads.

Alternatively, a security monitor can be nested within the process, leading to simpler and more efficient intra-process access controls [17, 20, 23, 45, 49, 57, 59, 63]. Prior nested subprocess monitors tailor their abstraction to a particular isolated application and/or OS, limiting programmability and flexibility, while also forcing application operators to choose between security or compatibility because they neglect to securely emulate necessary OS interfaces [10, 26].

We present the Endoprocess (Figure 1), a novel process organization for securely nesting protection within a process space. The endoprocess monitor, the *endokernel*, exports a low-level *subprocess* abstraction that overlays existing process interfaces (like mmap, mprotect, etc) to safely multiplex process resources for intra-process

entities. The endokernel provides extensibility and programmability through a *translation layer* that maps custom application-layer abstractions to the low-level subprocess interface without compromising the endokernel. The endokernel is nested within a process and designed so that it can safely isolate subprocesses while supporting non-bypassable, backwards-compatible emulation of OS interfaces, not conflicting with process-level protections, and remaining portable to different isolation mechanisms.

Our prototype, Little Mac, uses a lightweight mechanism (Intel® MPK) for isolating the endokernel and its subprocesses, and introduces a new syscall monitor that is thread safe, nested and ensures mediation. In Little Mac we introduce customized virtual privilege rings that aid in least-privilege refactoring, where complex resource sharing makes it practically impossible otherwise. By assigning a subprocess to one of three privilege levels and labeling code and data that belongs-to the subprocess, the Endoprocess automatically transforms and isolates.

2 MOTIVATION

In this section, we present a motivating use case, its threat model, and describe why current access control systems are insufficient for isolating within processes.

2.1 Least-Privilege Compartmentalization

Consider a web server that multiplexes client requests within a single process. For example, an NGINX worker process receives HTTP requests from distinct clients. The worker process independently parses and routes each message (with regard to authentication and authorization logic as applicable) to find and return the requested resource to the correct client. Resources at the application level either reside in process-accessible memory, or are accessible locally or remotely through file or socket descriptors *e.g.,* client data or server configuration on the file system; or resources like databases located on backing servers.

A remote attacker could send a crafted request that exploits a buffer overflow in the parser and launches a code-reuse attack. Such a malicious client gains access to all runtime data that the worker process is authorized to access, even data belonging to other clients, the web server runtime, or other system users. This data can then be exfiltrated through abuse of the server's network privileges. This means that the attacker has gained the local system privileges of the user that the process runs on-behalf-of.

One way to stop this privilege escalation is through enforcing least-privilege security policies that decompose the runtime into parts that can only access the state required to do their job. This limits the resources and memory the attacker can access, and reduces the possibility of lateral movement to other components of the system. For example, when processing a given HTTP request, an HTTP header parser may only need read access to an in-memory data structure containing the request headers. A clear policy restricts parser privileges to only reading the header data structure. This way, if compromised, the HTTP header parser cannot interact with any other memory in the process, nor inject code, nor read files, nor send messages on the network.

2.2 Threat Model

We assume any component within a process may have exploitable vulnerabilities or be directly malicious. We assume that some operations (file operations, memory mappings, networking, etc.) can by design invoke privilege escalations [10] that bypass components' memory protections. An attacker can escalate their privilege through exploiting some component within a process. From this privilege escalation the attacker may gain read or write access to the entire process' memory space; or they may gain ability to launch data-only attacks without full control; or they may gain the ability to execute code on-behalf-of the system user that the current process runs as; or they may gain access to any system resources that the current user has authorization to access. The goal of an endoprocess is to prevent these intra-process privilege escalations.

The operating system, hardware platform, and monitor are part of the Trusted Computing Base, and assumed bug free and to have no backdoors. Side-channel attacks are considered out of scope. Finally, we assume developers can identify application components, annotate each component with the correct policy, and define appropriate boundaries, privilege restrictions, and data sharing policies.

2.3 Process Isolation Limitations

While being operated on, application and user state resides in runtime memory and is otherwise stored persistently on disk. Operating systems isolate *runtime* data using virtual memory (address space) and *persistent* data using file and network based access control systems. Typically, an operating system provides discretionary access controls (DAC) that enable data owners to specify policies on which users, groups, and applications are permitted to access persistent data. Supplementing DAC, modern process-level protections evolved Mandatory Access Control (MAC) systems like SELinux, TrustedBSD, and AppArmor. Administrators configure system-wide MAC policies that operate at a process-level granularity. Since these these mechanisms operate at the process level, they cannot isolate sensitive resources running inside a process from a compromised component in the same process.

One approach to providing access controls for application compartments is to privilege separate [46] the application into multiple processes and use address-spaces to isolate the runtime state and existing access control mechanisms to isolate persistent data. Coarse-grained least-privilege [48] protections following such an approach have previously been applied to web browsers and cryptographic libraries [3, 22]. Unfortunately, attackers operate at the granularity of individual program objects, which requires much more efficient isolation than even state of the art process-level mechanisms provide [17, 23, 45, 49, 57, 59]. Each component within the process can access the full process address space, so we must also isolate exploitable components *within* the process at subprocess granularity.

2.4 In-Process Monitor Limitations

Several research and industry solutions have made great strides towards efficient, in-process isolation [17, 20, 23, 44, 45, 49, 57, 59, 63]. The core idea is to combine fast isolation hardware with a lightweight security monitor that isolates memory between *subprocess* components. This means that a particular component of a web

server, like an HTTP header parser, can be placed in a memory *sandbox*. Any request for memory outside of the sandbox will be denied based on a fine-grained access control policy.

Unfortunately, modern operating systems expose runtime state through interfaces that allow the monitor to be bypassed [2, 10, 20, 26, 44, 59, 63]. For example, POSIX allows access to runtime memory through the `/proc/self/mem` pseudo-filesystem interface, which can corrupt or leak monitor state as depicted in the following pseudocode:

```
open("/proc/self/mem","r").seek(s).read(0x10)
```

One approach is to deny interface access, as in intra-app sandboxing [17, 43, 60, 62], but such work indiscriminately denies *all* interface access. Denying all interface access means that the monitor cannot be inserted, nor isolate itself, nor allow application-level authorized use of any interface, limiting the applications of such systems to sandboxing only.

Distinct from the clear need to isolate the monitor, subprocesses require the ability to have multiplexed access to some, but not all, system interfaces. For example, in an application using OpenSSL for cryptographic functionality, only the OpenSSL library needs access to the key files stored in the user's `.ssh` directory. The rest of the application, and indeed the rest of the process running the application, does not need this access. Prior systems cannot enforce such a restriction because the main part of the application still requires use of the filesystem.

Finally, the abstractions that security monitors in prior work provide for specifying access control policies are bespoke and require significant security and domain-specific application expertise to use effectively. Specifying and maintaining such policies is extremely costly, if possible at all for a typical developer or administrator.

3 ENDOPROCESS ORGANIZATION

An endoprocess is a process that includes a nested security monitor, the *endokernel*, that exposes a universal *subprocess* abstraction. Developers can directly implement a diverse array of access control policies by placing components into subprocesses, which are mediated by the endokernel. It is also possible to extend the Endoprocess with *policy languages*, *isolation abstractions*, and *support libraries* to simplify creating and maintaining access control policies. Policies are mapped to subprocess protections through a *translator*. We depict the endoprocess structure in Figure 2.

3.1 Design Principles

We argue that a practical and generally useful subprocess access control framework must: (1) enable operators to write and apply common-sense, component-level access control policies (2) without unnecessarily restricting the flexibility or integrity of the application or the in-process monitor, that are (3) uniformly enforced at the system layer. These requirements embody a *microkernel*-inspired approach to subprocess access control, and guide our design choices.

Least-Effort Programmability and Maintainability. Firstly, a major challenge in making it attractive to retrofit an application to run in an endoprocess is minimizing the level of effort required to leverage the new abstractions. If the total effort required is too much, then writing an isolation policy and any subsequent maintenance will not be practical. Since the access control mechanism may not be available everywhere a modified application will run, or developers may attempt to run unmodified applications on the architecture, the most maintainable approaches will transparently execute either unmodified applications on the architecture, or modified applications without the architecture. However, such a method can add maintenance overhead in order to support cases where the protection mechanism may not be available. That is, if the approach requires modifying code to call a runtime library that implements the abstraction, then the translator must remove the dialect code if no protection mechanism is available. We find that the application developer is best suited to specify their own application access control policies [11]. We require a lightweight annotation dialect that application developers can write with minimal effort, that can be systematically lowered using standard compilation toolchains.

Avoid Unnecessary Coupling For Flexibility. Secondly, existing approaches strongly and unnecessarily couple the application and system layers of the runtime environment [9, 17, 35], hardcoding application-specific functionality into the monitor and making sweeping changes to the OS and application that limit their flexibility. This eliminates potential for competing implementations of an abstraction, and for easy extensibility. Reusing the common core of such a system is not possible, as each new runtime requires significant, error-prone porting work. Configuring these prior systems can require adding untrusted code to the monitor, broadening system-layer attack surface. Applications typically include components with unique needs for isolation, performance, and programmability. The protection system must be flexible and expressive enough to systematically support these needs.

Retain Functionality Without Compromising Security. Finally, prior works that utilize a security monitor not only tightly couple it to a specific application and runtime, they map application level abstractions to the system layer in ways that neglect critical security or functionality needs. For example, many prior subprocess-level isolation mechanisms implement some form of memory isolation, but these systems often neglect preventing memory access through system interfaces [34, 49, 57], or memory can leak through OS interfaces like `/self/proc/mem` [10]. There is also a general lack of support for commonplace functionality most modern applications leverage like multi-threading, process control, and signals. Overall, we observe a lack of systematic analysis of required functionality and associated security needs in prior work. A general purpose in-process security monitor and protection framework must support the functionality expected by applications.

3.2 Endokernel

The endokernel is a self-protecting, nested security monitor [11, 51] designed to enforce two privilege levels and mediate interaction between subprocesses. We chose to nest the endokernel within the process in order to eliminate costly context switches to supervisor mode and to virtualize the process runtime so that the endokernel is not bypassable. As such, the main objectives of the endokernel are firstly to isolate itself from the application layers above, and secondly to isolate application layer components from each other,

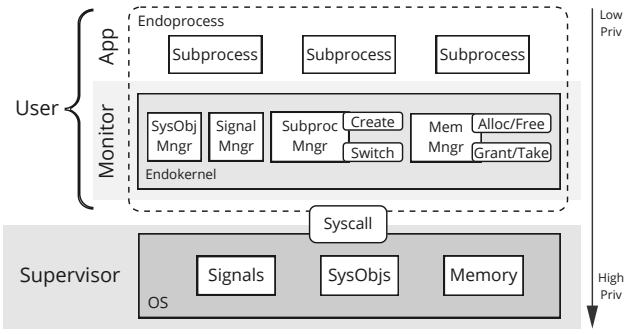

Figure 2: **Endoprocess Architecture**

within the process, under the subprocess abstraction. With the endokernel in place, we can enforce properties such as isolated secrets or code-pointer integrity (as demonstrated by [57]), or alternately can expose the subprocess abstraction to enable developers to isolate sensitive parts of a given application.

The endokernel protects its own runtime state through **memory virtualization**. This means that it must be loaded into the process (sharing the process' single address space), and must use a memory protection mechanism to ensure that the endokernel can be safely entered and that the endokernel memory cannot be tampered with. Examples of memory protection mechanisms that have been used for in-process memory isolation include: randomization, software fault isolation, memory protection keys, and hardware virtualization nested paging [13, 26, 32, 36, 41].

Once memory isolation and protected entrypoints are in place, memory access through system level interfaces must be mediated. To virtualize system interfaces, the endokernel ensures that the application layers above the endokernel cannot directly interact with the operating system through **syscall virtualization**. In modern CPUs, the operating system is typically accessed through system calls provided by hardware. System call monitoring can be enforced with any number of mechanisms such as language based hooking or hardware monitoring, however, the primary requirement is that each call goes through the endokernel layer of virtualization. To remain consistent with our design goal of decoupling the subprocess abstraction from the OS, the mechanism must require only minimal OS modifications to support.

Once the syscall monitor is in place, we enforce policies over operating system interfaces using **system object virtualization**. This requires in depth analysis of each interface to derive policies ensuring that the endokernel cannot be bypassed. Typical OS abstractions that may otherwise bypass endokernel self-protections and need to be accounted for include: memory access through file system interfaces, signals, threads, process creation and destruction, address space manipulation such as mapping, remapping, or protection changes, as well as file system and network interfaces. Such an analysis and virtualization policy are possible for any operating system assuming the semantics of the interface and OS are known. For example, in our prior work, we examined and virtualized POSIX in Linux [26].

3.3 Anatomy of an Endoprocess

Once a process is split into the monitor and application layers, the endokernel can then support primitives for isolating components within the application layer. While a large variety of abstractions could be supported, we aim for the most general and flexible. The *subprocess* is a universal, system-level abstraction that aims to provide process equivalent functionality with subprocess-level access controls. A subprocess is a collection of program code and data along with assigned privileges, and is inspired by the protected subsystem [33] and encapsulated-object [61] protection models.

A subprocess is created by labeling *lexical scopes* that align with static program scopes, defining the boundaries of an isolated subsystem. After defining the boundaries, each program object is assigned to a single lexical scope. Assigned code and data *belongs-to* the subprocess, and all data belonging to the subprocess is placed in segments of the virtual address space called *subspaces*. A subspace is the smallest unit of memory protection to which access rights can be granted. A subprocess's lexically scoped privileges include several subspaces by default (analogous to the associated memory segments in a process): code, stack, heap. The lexical scope also contains privileges for system abstractions such as files, sockets, interrupts, *etc.* A subprocess's lexically-scoped privileges are activated when entering the lexical scope through statically declared entry points, while the caller's privileges are deactivated during a *context-switch*.

A subprocess can share data using either message passing (*e.g.*, microkernel-like copying) or by setting up a shared subspace that contains the data in question. Since pointers map back to subspaces that are owned by a particular subprocess, data sharing and management is greatly simplified because developers only need to allocate data into an appropriate subprocess and directly share a pointer through an interface—retaining C/C++ ABI semantics. This makes sharing pointers across subprocesses operate as sealed-capabilities, where permissions for sharing are explicitly expressed by creating shared subspaces by an owning subprocess. This ensures a sealed-capability cannot be passed to another subprocess as it would require explicit policy for any called subprocess.

Lexical scopes can be extended with an orthogonal privilege scope for time slicing access across program execution. A *dynamic scope* is a memory view and set of system object privileges that can be dynamically created and destroyed, and is one of the most novel parts of the Endoprocess approach. A developer modifies their code to create a dynamic scope through an endokernel interface. When called dynamically, the endokernel creates a dynamic scope object which includes privileges for subspaces and system object privileges, and returns a reference to the caller. The dynamic scope belongs-to the calling subprocess and is mapped into the lexical scope's privileges and inherits the system object privileges of the current lexical scope. A given dynamic scope holds privileges to data associated with a particular application-layer entity, so that only one application-layer entity's data and privileges can be used at a time, effectively time-slicing execution.

A dynamic scope can be used by a subprocess to temporarily lower or elevate privileges in one of two ways. First, a subprocess changes privileges within its own lexical scope by binding a dynamic scope to a function call. On the context-switch, the bound

dynamic scope is activated and all other dynamic scopes in the subprocess are deactivated. The called context has access to the dynamic scope through a dynamic scope object handler. As described for data sharing between lexical scopes, data in the shared subspace is referenced through standard pointers, enabling transparent sharing. The dynamic scope object is deactivated on return from the call.

Second, a dynamic scope can be bound on the context-switch to another subprocess, in which case it operates similarly to a shared subspace, but adds the dynamic scope object to the call while also ensuring that the protection system removes access after the call completes. The dynamic scope is inherited from the caller and can be fully or partially passed along to a separate subprocess, making time slicing across subprocesses easy to apply without significant code changes. The last major benefit of the dynamic scope is that when it is destroyed, all associated privileges and subspaces are also automatically removed, making any pointers to its data no longer usable.

The ambient authority at any given point of a subprocess's execution is the union of the active lexical and dynamic scopes. For preexisting, persistent resources (e.g. files, or handles in Windows), a developer establishes an access control policy specifying which resources belongs-to which lexical scope and dynamic scope, and then applies privilege sharing through lexical or dynamic scoping interfaces. For newly instantiated objects (*e.g.,* sockets, files, raw memory mappings, signals, processes via `exec`, *etc.*) a belongs-to assignment is granted to the creating lexical scope and dynamic scope, and shared using the lexical or dynamic scoping interfaces.

A subprocess can register both synchronous and asynchronous interrupts including signals or events such as aborting from `rseq`. The endokernel delivers the interrupt to the registering subprocess. The endokernel allows interrupts to be multiplexed between subprocesses. Each subprocess can be configured to allow other subprocesses to handle its interrupts or choose to block them. For example, if a global timer interrupt occurs in a subprocess that should not handle it, the endokernel will block its handling until there is a switch to a subprocess that can handle this interrupt. As well, if a segmentation fault occurs within a subprocess that is not allowed to handle it, the user can configure another lexical scope that can be trusted to handle the interrupt, or can simply allow the default behavior.

3.4 Programmable Languages and Translation

The subprocess provides a universal way to isolate and share resources within a process. While recent efforts show promise in automating boundary selection that maps directly to the subprocess model [47, 58, 61], these works do not adequately derive sharing policies that enhance security while minimizing effort and performance. The *translation* layer not only simplifies subprocess boundary selection, but also sharing policy specification, integration, and management with minimal programmer effort.

The Endoprocess approach includes a lightweight interface for creating custom domain specific access control frameworks. These are implemented as abstractions that fit the code naturally and provide automated policy structure for least-effort application and clarity to reason about security properties. For example, common access control concepts naturally map to the subprocess, such as virtual privilege rings or specialized relations between subprocesses such as sandboxing or client-server time slicing. Source level specifications are translated into subprocess enforcement. If a protection mechanism is not available, a policy is translated into a standard application runtime. Runtime *libsep* libraries implement classes of access control frameworks on top of the subprocess abstraction, eliminating the need for a developer to implement their own.

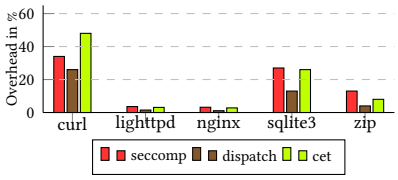

Figure 3: **Normalized overhead of diff. Linux applications. Std. dev. below 2.4%.**

4 LITTLE MAC

Our prototype Little Mac is one potential implementation of the endokernel abstraction. Little Mac is an Intel® MPK-based in-process security monitor that enforces isolation between subprocesses and ensures that all memory accesses have complete mediation, even through system interfaces. In Little Mac we implement a novel system call monitoring framework, and demonstrate the use of virtual privilege rings for privilege separation.

Implementation. We use MPK domains to implement access permissions for subprocess heaps and additionally to protect Little Mac's own sensitive data. Little Mac also maintains ownership and accessibility records for each heap.

As the foundation of our system call monitoring framework, we implemented a Little Mac-generated trampoline based on the principle that only code that passes a specific checkpoint can call a system call. Using this trampoline, we check restrictions on switching between different subprocesses and virtualize system calls. We ensure that only Little Mac can create syscall instructions at specific locations by using MPK [1] in combination and disabling other syscall instructions from running outside Little Mac via seccomp or `dispatch`. This ensures that the subprocess policy on system interfaces, interrupts (mainly signals), and Little Mac are not corrupted by the operating system. In particular, we ensure that signals do not break subprocess requirements for control flow, and that threads do not break our trampoline.

Lastly, Little Mac and other runtime components share information through a read-only PKEY in the user `fs` register. We show the abstraction overhead of using Little Mac for intra-process isolation of several distinct applications running on Linux in Figure 3.

Virtual Privilege Rings for Privilege Separation. One of the major challenges in privilege separation is the complexity of specifying boundaries and sharing policies. For example, while isolating the OpenSSL library from the rest of the program is powerful, the library requires significant interactions with the main program code

[1] Intel® CET[27] can also be used to implement our trampoline, though it should be noted that use of full CFI introduces greater performance overhead.

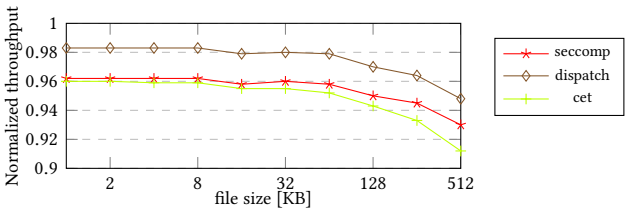

Figure 4: **Normalized throughput of privilege separated NGINX using TLS v1.2 with `ECDHE-RSA-AES128-GCM-SHA256`, `2048`, `128`. Std. dev. below 1.5%.**

and data. In addition, direct privilege separation is difficult because it necessitates changes to complex components like allocators. To show an example of how the application of least privilege can be simplified, we developed a Virtual Privilege Rings abstraction that provides three privilege levels with default security policies intended to reduce the effort required to manually assign cross-context policies.

Developers can annotate their data and code into subprocesses (called boxes here for ease of reference) and assign a specific virtual privilege rings privilege level. Then, Little Mac can automatically enforce the desired isolation. A sandbox has no privilege to any system interfaces or the ability to call other subprocesses (boxes). The mainbox remains untouched but has access to all sandboxes, allowing for incremental separation. A safebox isolates code and data from all other boxes but has access to all mainboxes and sandboxes. Other contexts can be created to accommodate more complex policies.

NGINX is a web server framework that multiplexes handling many requests from distinct clients in a given worker process, spawned from a main server process. Within NGINX, we sandboxed the HTTP parser and safeboxed OpenSSL in NGINX, restricting the mainbox from accessing anything in the `.ssh` directory. In Figure 4, we show the performance of NGINX separated using this virtual privilege rings-based approach with a maximum overhead of less than 10%.

5 MIGRATION OF EXISTING APPLICATIONS

A developer can choose to use one or several dynamic libraries as isolation boundaries, and our runtime will use the default policy to protect the data of the entire dynamic library, restricting external access to it only. This strategy may be crude, but it is already effective in defending against attackers who can only externally read and write memory. In our NGINX use case, OpenSSL is isolated in this way. We simply add code to the OpenSSL library that calls the runtime and changes the allocator used by OpenSSL.

Taking it a step further, developers can manually annotate code and data to be protected or to isolate functionality that is either error-prone or a likely attacker target, such as a parser or an authentication library. By adding annotations to a particular function's declaration and related data, developers can mark it as allowed to be called externally. Additionally, we are also developing automated tools that facilitate generation of the required markers using individual code files as isolation boundaries. These annotations are reflected in specific data and memory layouts through the compiler, and the runtime uses the documented policies accordingly.

At the language level, our annotations overlay isolation operations on top of program functionality, preserving existing semantics. Furthermore, our annotations facilitate the reuse of isolation information instead of having to rewrite code. If no protection mechanism is in place at runtime, the presence of our annotations has no effect, akin to the presence of CFI markers in assembly files. Application developers integrating previously-annotated third-party components can still choose appropriate levels of granularity and security for their own use cases.

6 USE CASES

In this section, we highlight how Little Mac can secure several notable examples of the trend toward deploying applications that firstly consist of a specific set of tightly-coupled heterogeneous components that should be accorded differing privileges and levels of trust, and secondly handle the data of mutually distrusting entities. For each of these different application runtime environments, we demonstrate the flexibility and portability of our mechanism.

6.1 Application Runtimes

Compartmentalization. Through compartmentalization, safety-critical code can be isolated from the rest of the application. Little Mac simplifies the adaptation of such a strategy. With the translation layer in place, developers can easily define the boundaries of each compartment using subprocesses, and set the privileges of compartments using virtual rings. In our application of Little Mac to NGINX earlier in this paper, the HTTP parser and OpenSSL library each get their own subprocess where the HTTP parser is sandboxed, and OpenSSL is protected from the rest of the application. We annotated the parser functions so that they were sandboxed, and marked the entire OpenSSL library as a safebox.

Multi-user applications. By binding dynamic scopes to user authentication information and carefully managing the lifetime of each bound dynamic scope, Little Mac can ensure that each user's private information is not accessible to other users or other parts of the application. For a database application, Little Mac can create a subprocess when an incoming connection from a new user is accepted, bind the user's authentication information to a new dynamic scope, and ultimately unbind the dynamic scope when the connection is closed. Any remote, authenticated operation will now depend on having the Little Mac-enforced privilege to access the appropriate dynamic scope.

Containerized runtimes. Containers simplify repeatable service builds and deployments; however, it is difficult to isolate even simple components from one another within a given container. Multiple containers (microservices) generally are deployed today if developers require service-level isolation such as between a UI-serving web front end, an authenticated backend, and a database. However, such splitting introduces unnecessary communications overhead. We can use the Little Mac-protected endoprocess space locally akin to how one might otherwise employ a service mesh to coordinate, monitor, and secure multiple networked containers running distinct application components. Such a runtime would require development

of additional translation-layer modules. However, running each containerized application component as a subprocess would allow communication between these components to occur through shared memory, while still isolating each component's private memory and other resources.

Serverless runtimes. Separation into AWS Lambdas can also be employed today when developers would like to run simple network-accessible components without taking on infrastructure management themselves. However, the overhead of networking between these lightweight components remains. Cloudflare uses V8 Isolates as the foundation of Workers [5], a somewhat more secure alternative to Lambdas at the edge. It should be noted that V8 Isolates do run Javascript, and each Isolate can run unrelated Javascript applications simultaneously. Even an isolated V8 runtime environment could still be subject to Javascript concurrency and memory bugs that could result in an attacker gaining inappropriate access to other users' data or even potentially breaking out of the Isolate. If augmented with Little Mac, each Lambda or Worker could be assigned a subprocess that determines its lexical and dynamic scope, based on the application it belongs to and the user's request. This assignment would provide each serverless runtime with a non-bypassable foundation in the subprocess as enforced by the monitor. Since Little Mac multiplexes (virtualizes and mediates) all resource access on-behalf-of a given subprocess, an integration between the endokernel and the original serverless runtime that accommodates JIT code would be required to achieve this augmentation, but the serverless runtime would greatly benefit from enhanced security measures without incurring the performance overhead associated with splitting into multiple Lambdas or Workers for isolation.

Sandboxing using a nested system call monitor. Application kernels like gVisor [18] use a system-call monitor running in a separate process that multiplexes system resources and enforces developer policy regarding system calls originating from a given sandboxed, containerized application. However, this approach incurs a significant performance overhead from IPC and can only enforce limited policies via common mechanisms such as seccomp, when compared to Little Mac. Replacing gVisor's monitor with the endokernel, we could eliminate the overhead of having a separate monitor process, since endokernel both isolates itself and nests within the process.

Unikernels. The Unikernel [38] packs an application into a single address space, eliminating need for a traditional kernel. However, intra-unikernel risks presented by potentially vulnerable or buggy components within the unikernel to other unikernel components naturally still remain [53]. The endokernel offers security guarantees at a minimal cost, as it is specifically designed for isolating components and does not have any additional default abstractions. By treating the core of the unikernel as a subprocess, it could be protected while minimizing the performance impact typically associated with switching through system call instructions. This approach effectively reduces the security risks associated with running multiple potentially distrusting components within a single address space. Moreover, developers would be enabled to enhance both security and reliability by leveraging annotations within their application code to further isolate components or data associated with external users using Little Mac.

6.2 Mechanism Portability

Hardware. Different hardware mechanisms can be used to implement the Endoprocess. In addition to MPK, a popular capability hardware mechanism, CHERI, encodes privileges (capabilities) in new pointer types that can be used by the compiler to implement compartmentalization. Also, since user programs can only obtain specific entry point addresses, they cannot break the Endokernel's abstraction, and all switches depend on the Endokernel providing the correct target function address. Based on this, the implementation of system calls is similar to that of MPK. Therefore, we can still implement the Endokernel on such hardware.

OS Interface. The endokernel isolates memory and prevents ways that existing OS interfaces can violate that isolation. Extending the endokernel to non-POSIX system interfaces (such as Windows) is possible and would demonstrate the portability of the approach.

7 RELATED WORK

Process-based Sandbox Process-based sandboxing [7, 30, 46] is a common approach to isolate untrusted code. It is now widely used in browsers, such as Chromium and Firefox, because processes provide strong isolation at the address space level. Ptrace and seccomp are used to monitor the system call usage of sandboxed processes and sometimes SELinux, Apparmor, and namespaces are also used to further restrict resource access.

However, beyond the inability to enforce subprocess level polices, these approaches lack flexibility. SELinux is based on pre-defined rule files and cannot be changed dynamically. Seccomp-BPF's filter program has limited ability—it cannot check if the open path is legal because it can only access the values of CPU registers. IPC is thus often used to communicate with a privileged process to enable complex policies. At the same time, mutual access between isolated modules must be translated into access to the IPC interface. This brings additional development costs and overhead to adapt existing applications to the sandbox, and limits its granularity.

OS-based Isolation Some works [4, 14, 25, 35, 39] depend on the operating system to provide the functionality needed for isolation within the same process space. The kernel creates separate page tables for modules that need to be isolated. And the kernel is responsible for switching page tables when switching between modules. This allows sharing between different modules. And since the kernel has information about user-space isolation, it can also mediate vulnerabilities caused by system interfaces [35].

However, switching between page tables via system calls still has a relatively large overhead. In addition to the syscall, page table switching also reduces the overall efficiency of the application's access to memory. And, for security reasons, operating systems can only provide relatively rigid interfaces to applications. Thus, similar to process isolation, they are constrained in policy programmability, and cannot accommodate the needs of different applications.

Virtualization-based Isolation Hodor and SeCage [23, 37] use virtualization instructions like Intel® VT-X [2] for switching memory views instead of system calls. This approach enhances the efficiency of context switching and has shown encouraging results. However, like OS-based isolation, these mechanisms still cannot

provide effective and flexible protection for system resources. Moreover, virtualization can only be effectively accelerated on the host machine, which limits its usage.

MPK based Isolation Intel® MPK [2, 44] enables switching memory permissions without kernel intervention, improving isolation efficiency and allowing for finer-grained isolation, yet posing potential security risks. Subsequent works, like ERIM [57] and Hodor [23], focused on MPK's efficiency and instruction protection. However, they ignore the necessity of fortifying monitors against operating system interface based bypass [10]. Enclosure [17] and PKRU-Safe [31] utilize MPK to provide memory sandbox for specific languages. They offer limited system call filters akin to seccomp using the language runtime to enforce policies. Unless employing a strict sandbox policy, they also suffer from issues similar to those faced by ERIM. Cerberus [59] introduced a PKU-based sandboxing framework to protect in-process monitors, but it lacks signal support and doesn't address operating system resource limitations. Jenny [49] further proposed a system-wide solution with programmable system call filters for modules, but signal support remains limited and security concerns persist.

MPK suffers from having only 16 hardware PKEYs. libmpk [44] mitigates this issue by multiplexing PKEYs. EPK [20] combines VMFUNC and MPK to allow more domains through the extended page tables. VDom [63] achieves this more efficiently by employing page tables in combination with PCID, thereby avoiding the VM overhead and also reducing the TLB shootdown. VDom, remains insecure to exploits at interfaces however could be used as an alternative mechanism for implementing the Endoprocess.

8 FUTURE WORK

We will refine the endoprocess model to support rich real-world application scenarios, such as providing isolation for each user session in multi-user applications. This is orthogonal to lexical scope isolation, creating a client context for private data and privileges, and enabling mediated interactions between user and module data.

We will explore automatic generation of boundaries and access control policies. Static and dynamic analysis will be used to infer global data sharing relationships and the use and transfer of data through interfaces. The aim is to allocate, copy, and manage access rights. This kind of analysis will be field-sensitive, providing information to disaggregate related structures to minimize sharing.

We will explore alternative protection mechanisms for isolating the endokernel and subprocess to demonstrate portability and explore performance trade-offs. For instance, compared to Intel® MPK, which changes memory accessibility directly through instructions, the ARM platform offers extensions for pointer protection. These features allow for finer-grained and more flexible memory sharing and permission management. However, they also present new challenges for the design and construction of nested monitors. This is because their security can't be simply attributed to the protection of register states and control flow. Instead, it depends on the confidentiality of these pointers during the program's execution, including interactions with the operating system.

9 CONCLUSION

In this paper, we propose the radical idea of moving away from the process abstraction as the fundamental unit of isolation. As applications already adopt this approach, and more following suit, we propose the use of the Endoprocess approach that nests a protection monitor inside the process. This approach offers complete isolation and exports the simple subprocess interface. The subprocess allows for fine-grained isolation without unnecessarily burdening the endokernel. Furthermore, it supports extensible and programmable abstractions through a translation layer. This translation layer can simplify usability and enable new applications to quickly build on the base layers while closely aligning with the specific requirements of each application. By leveraging a well-tested organizational philosophy (microkernel), the endokernel provides an essential substrate with powerful portability, flexibility, and performance features.

ACKNOWLEDGMENTS

We would like to acknowledge and thank our shepherd, David Barrera, the anonymous reviewers, and the NSPW attendees for providing instructive feedback that improved this paper. This research was supported in part by National Science Foundation Awards #2146537 and #2008867.

REFERENCES

[1] 2013. CVE-2013-4547. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4547

[2] 2020. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

[3] AGWA. 2020. AGWA/titus: Totally Isolated TLS Unwrapping Server. https://github.com/AGWA/titus. (Accessed on 07/07/2023).

[4] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, San Francisco, California, 309–322.

[5] Zack Bloom. 2018. Cloud computing without containers. https://blog.cloudflare.com/cloud-computing-without-containers/

[6] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca bx Shapiro, and Anna Shubina. 2014. Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier. *IEEE Security & Privacy* 12, 1 (Jan. 2014), 83–87. https://doi.org/10.1109/MSP.2014.1

[7] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/13th-usenix-security-symposium/privtrans-automatically-partitioning-programs-privilege

[8] Center for Internet Security. 2021. The SolarWinds Cyber-Attack: What You Need to Know. https://www.cisecurity.org/solarwinds/

[9] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (2016). IEEE Computer Society, 56–71. https://doi.org/10.1109/SP.2016.12

[10] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)* (2020). USENIX Association, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[11] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015) *(ASPLOS '15)*. ACM, 191–206. https://doi.org/10.1145/2694344.2694386

[12] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. 9, 3 (1966), 143–155. https://doi.org/10.1145/365230.365252

[13] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan.

2016. SpaceJMP: programming with multiple virtual address spaces. *ACM SIGPLAN Notices* 51, 4 (2016), 353–368.

[14] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 353–368. https://doi.org/10.1145/2872362.2872366

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[16] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference* (2020-12-07) *(Middleware '20)*. Association for Computing Machinery, 265–279. https://doi.org/10.1145/3423211.3425680

[17] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA, 2021-04-19) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728

[18] google. 2022. gVisor. https://gvisor.dev/.

[19] James Gosling. 2000. *The Java language specification*. Addison-Wesley Professional.

[20] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[21] Juan Andrés Guerrero-Saade. 2022. CrateDepression | Rust Supply-Chain Attack Infects Cloud CI Pipelines with Go Malware. https://www.sentinelone.com/labs/cratedepression-rust-supply-chain-attack-infects-cloud-ci-pipelines-with-go-malware/.

[22] H2O. 2023. h2o/neverbleed: privilege separation engine for OpenSSL / LibreSSL. https://github.com/h2o/neverbleed. (Accessed on 07/07/2023).

[23] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[24] Matt Howard. 2021. 2021 State of the Software Supply Chain: Open Source Security and Dependency Management Take Center Stage. https://blog.sonatype.com/2021-state-of-the-software-supply-chain.

[25] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria, 2016-10-24) *(CCS '16)*. Association for Computing Machinery, 393–405. https://doi.org/10.1145/2976749.2978327

[26] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. 2021. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. https://doi.org/10.48550/ARXIV.2108.03705

[27] Intel. 2021. *Chapter 18 Control-flow enforcement technology (CET)*. Vol. 1. Intel, 409–422. software.intel.com/content/www/cn/zh/develop/articles/intel-sdm.html

[28] Antti Kantee et al. 2012. Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels. (2012).

[29] Kelly Kaoudis and Sick Codes. 2021. Rotten Code, Aging Standards, & Pwning IPv4 Parsing. https://www.youtube.com/watch?v=_o1RPJAe4kU

[30] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*. USENIX Association, San Antonio, TX. https://www.usenix.org/conference/2003-usenix-annual-technical-conference/privman-library-partitioning-applications

[31] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[32] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014-10). USENIX Association, 147–163. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[33] Butler W. Lampson. 1974. Protection. 8, 1 (1974), 18–24. https://doi.org/10.1145/775265.775268

[34] 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

[35] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016) *(OSDI'16)*. USENIX Association, 49–64. http://dl.acm.org/citation.cfm?id=3026877.3026882

[36] 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* (New York, NY, USA, 2015) *(CCS '15)*. ACM, 1607–1619. https://doi.org/10.1145/2810103.2813690

[37] 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)*. 1607–1619.

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 461–472. https://doi.org/10.1145/2490301.2451167

[39] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. 2016. Trellis: Privilege Separation for Multi-user Applications Made Easy, Vol. 9854. 437–456. https://doi.org/10.1007/978-3-319-45719-2_20

[40] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[41] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. 2018. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *Research in Attacks, Intrusions, and Defenses* (Cham, 2018) *(Lecture Notes in Computer Science)*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer International Publishing, 359–379. https://doi.org/10.1007/978-3-030-00470-5_17

[42] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *2016 IEEE Cybersecurity Development (SecDev)*. 45–52. https://doi.org/10.1109/SecDev.2016.019

[43] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)* (2020-08). USENIX.

[44] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[45] Aditi Partap and Dan Boneh. 2022. Memory Tagging: A Memory Efficient Design. arXiv:2209.00307 [cs]

[46] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003) *(SSYM'03)*. USENIX Association, 16–16. http://dl.acm.org/citation.cfm?id=1251353.1251369

[47] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. 296–311.

[48] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. 63, 9 (1975), 1278–1308. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1451869

[49] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[50] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless computing: a survey of opportunities, challenges, and applications. *Comput. Surveys* 54, 11s (2022), 1–32.

[51] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. 2017. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA, 2017) *(NDSS '17)*. The Internet Society. http://www.internetsociety.org/using-replicated-execution-more-secure-and-reliable-web-browser

[52] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux Security Module. 1, 43 (2001), 139.

[53] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th*

ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (Lausanne, Switzerland, 2020-03-17) *(VEE '20)*. Association for Computing Machinery, 143–156. https://doi.org/10.1145/3381052.3381326

[54] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013) *(SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[55] Zahra Tarkhani and Anil Madhavapeddy. 2020. *Enclave-Aware Compartmentalization and Secure Sharing with Sirius*. arXiv:2009.01869 [cs] http://arxiv.org/abs/2009.01869

[56] unikraft. 2022. Unikraft/Unikraft. Unikraft.

[57] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-Process Isolation with Protection Keys (MPK). 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[58] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, Andre DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium* (San Diego, CA, USA, 2018) *(NDSS '18)*. The Internet Society.

[59] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[60] WebAssembly Community. 2020. *Security - WebAssembly*. https://webassembly.org/docs/security/

[61] Yudi Yang, Weijie Huang, Kelly Kaoudis, and Nathan Dautenhahn. 2023. Whole-Program Privilege and Compartmentalization Analysis with the Object-Encapsulation Model. In *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 1–12.

[62] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted X86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009) *(SP '09)*. IEEE Computer Society, 79–93. https://doi.org/10.1109/SP.2009.25

[63] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2023. VDom: Fast and Unlimited Virtual Domains on Multiple Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 905–919. https://doi.org/10.1145/3575693.3575735

[64] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What Are Weak Links in the Npm Supply Chain?. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 331–340. https://doi.org/10.1145/3510457.3513044