Enclosure: Language-Based Restriction of Untrusted Libraries

Adrien Ghosn
EPFL
Switzerland

Marios Kogias*
Microsoft Research
United Kingdom

Mathias Payer
EPFL
Switzerland

James R. Larus
EPFL
Switzerland

Edouard Bugnion
EPFL
Switzerland

ABSTRACT

Programming languages and systems have failed to address the security implications of the increasingly frequent use of public libraries to construct modern software. Most languages provide tools and online repositories to publish, import, and use libraries; however, this double-edged sword can incorporate a large quantity of unknown, unchecked, and unverified code into an application. The risk is real, as demonstrated by malevolent actors who have repeatedly inserted malware into popular open-source libraries.

This paper proposes a solution: *enclosures*, a new programming language construct for library isolation that provides a developer with fine-grain control over the resources that a library can access, even for libraries with complex inter-library dependencies. The programming abstraction is language-independent and could be added to most languages. These languages would then be able to take advantage of hardware isolation mechanisms that are effective across language boundaries.

The *enclosure* policies are enforced at run time by LitterBox, a language-independent framework that uses hardware mechanisms to provide uniform and robust isolation guarantees, even for libraries written in unsafe languages. LitterBox currently supports both Intel VT-x (with general-purpose extended page tables) and the emerging Intel Memory Protection Keys (MPK).

We describe an *enclosure* implementation for the Go and Python languages. Our evaluation demonstrates that the Go implementation can protect sensitive data in real-world applications constructed using complex untrusted libraries with deep dependencies. It requires minimal code refactoring and incurs acceptable performance overhead. The Python implementation demonstrates LitterBox's ability to support dynamic languages.

CCS CONCEPTS

• **Software and its engineering** → **Runtime environments**; • **Security and privacy** → **Software security engineering**.

KEYWORDS

Security, intra-address space isolation, programming languages, software packages

ACM Reference Format:
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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446728

1 INTRODUCTION

Programming has changed; programming languages have not. Modern software development has embraced abstraction and reusable software components. Today, applications are built using open-source libraries (aka packages) that offer diverse, tested functionality that increases programmer productivity. In the extreme, an application can become a collection of libraries orchestrated by application-specific code. To facilitate code sharing, modern languages provide tools and online repositories to publish, find, download, access, and update public libraries, *e.g.,* Python modules [3], Go packages [27], Ruby gems [4], and Rust crates [5].[1]

Although languages embrace libraries, few, if any, provide mechanisms to address the insecurity and fragility inherent in their use: (1) packages come without a formal specification of what they do, and do not do; (2) their developer is typically unknown, thus untrusted; (3) they can import other unknown and untrusted packages, and lack traceable dependence management; and (4) most important, programs run in a single trust domain that does not isolate code or data from different packages.

In general, a developer's trust in a public package often appears to be based on its popularity or a shallow code review. Careful inspection is both impractical, since importing a single package may incorporate hundreds or thousands of transitively dependent packages [49, 50], or infeasible, as a package's code may change frequently. As a result, an application can become a patchwork of code from untrusted and unverified sources.

Malevolent individuals have been quick to exploit the opportunity to insert malicious code in popular packages [14, 15, 19, 72], to modify an IDE to insert the code [59], or to substitute modified clones [16–18, 37, 43]. These attacks are easy to implement and provide unimpeded access into hundreds, if not thousands, of applications for malicious code that steals private information or opens backdoors. For example, malicious Python packages stole SSH and

*Work done while the author was at EPFL.

[1]Frameworks, such as node.js, also support library repositories. This paper considers them from a language-specific perspective, e.g., as JavaScript packages.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446728

GPG keys from the local file system [15, 18]. More generally, even legitimate third-party libraries may implement undocumented functionality that operates outside of its advertised scope. For example, the Facebook iOS SDK to identify users also shared device information with Facebook without user consent [70].

Although the systems, programming languages, and security communities have long studied software isolation [11, 13, 21, 31, 32, 38, 40, 40–42, 44, 47, 53, 54, 57, 61, 64, 65], previous approaches do not address the full range of requirements for package isolation and security: (1) low-level systems abstractions may not match programming language requirements [11, 21, 40, 44, 53, 54, 65] or may require extensive application refactoring [13, 41, 42]; (2) pure language approaches, *e.g.,* Rust or JavaScript isolates, are limited to a single language and programs written only in the language; (3) mixed approaches, *e.g.,* Erim [61], Hodor [31], and Glamdring [38], ignore package structure, are unaware of dependencies among packages, or lack expressive access rights.

While packages are at the root of security and fragility problems, their unique characteristics also facilitate solutions. Specifically, packages consist of code and data written to run as part of any program, which means they must have clearly defined entry points, not be dependent on a program's environment, and explicitly declare and import their dependencies. They can run in isolation given access to their input data and the packages they depend upon. The clear boundaries of a package can help partition a program's memory address space into isolated regions that prevent code in the package from improperly accessing the rest of the program's environment.

We propose a new programming language construct that gives a developer fine-grain control over the package resources that a computation can access, even for packages with complex dependencies. It introduces a dynamically-scoped set of restrictions on which parts of the address space can be accessed and which system calls can be invoked. The abstraction is language-independent and could be added to most languages. Its implementation relies on hardware isolation mechanisms that provide trustworthy, fine-grain access control within a virtual address space [1, 10, 33, 60, 67].

This construct is called an *enclosure*. It implements an isolation policy for a closure by binding it to a memory view and a set of permitted system calls, which restricts access to program resources by the code invoked in the closure, regardless of which package contains it. The current system starts with a view that limits access only to the resources in the packages that the closure invokes. A developer can restrict or extend this view by selectively enabling read, write, or execute access rights for a specific package. The developer can also selectively allow system calls. The policy is dynamically scoped and applies to all code executed by the *enclosure*, which in turn can invoke other *enclosures* that further restrict the accessible resources.

LitterBox enforces *enclosure* policies at run time. It is a language-independent framework that uses hardware mechanisms to provide uniform and robust isolation guarantees, even for packages written in unsafe languages. LitterBox exposes a high-level API that is reusable across programming languages. The isolation is built on one of several different hardware technologies, and LitterBox hides the hardware complexity.

```
1  package main
2
3  import(
4    "img"
5    "libFx"
6    "os"
7    "secrets"
8  )
9
10 var key := "myPrivateKey"
11
12 func main() {
13   original := secrets.LoadImage("myimg.png")
14
15   rcl := with["secrets:R","none"]func(o *img.Png) *img.Png {
16     return libFx.Invert(o)
17   }
18
19   inverted := rcl(original)
20   res := img.Watermark(inverted, key)
21   os.Write("signed.png", res.Bytes())
}
```

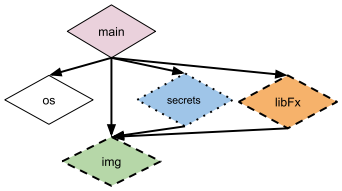

Figure 1: The `rcl` *enclosure* prevents the call to the public package `libFX` from modifying or leaking sensitive information. The top-right corner shows the application's package-dependence graph, with `rcl`'s natural dependencies in dashed borders, and its extended read-only view to `secrets` in dotted borders. Color-coding of variables highlights which package arena holds the corresponding value.

This paper makes the following contributions:

- *Enclosure* is a dynamically scoped programming language construct that imposes user-defined access policies on code invoked within it. These policies restrict, at package granularity, what parts of a program (data and code) this code can access and which system calls it can invoke. Untrusted packages, even those with deep dependence graphs, can be isolated from sensitive information.
- LitterBox is a language-independent framework to enforce *enclosure* policies with robust hardware isolation mechanisms. LitterBox currently supports both Intel VT-x (with its general-purpose extended page tables) and the emerging Intel Memory Protection Keys (MPK).
- We describe an efficient implementation of *enclosures* and LitterBox for the Go language. It has low overheads for real-world applications and can drastically reduce a program's trusted codebase.
- We describe a prototype implementation of *enclosures* and LitterBox for the dynamic programming language Python.
- We present experiments demonstrating that the overhead can be as low as 1.02x for real applications.

LitterBox and both language frontends are open-sourced [9].

2 *ENCLOSURE* CONSTRUCT

An *enclosure* is a programming language construct that enables a developer to restrict code's access to program resources to prevent untrusted code from accessing, modifying, or leaking sensitive data. It limits the code to access only functions and data from specified program packages (the memory view) and to execute only explicitly allowed system calls. These restrictions are dynamically scoped, so they apply to the closure's body and the code invoked by it.

2.1 Definitions

A *package* can export four items for use by other packages: (1) functions (code), (2) variables (mutable data), (3) constants (immutable data), and (4) arena (heap). Variables are either (1) static variables (*e.g.,* pre-allocated globals) or (2) dynamic variables (dynamically allocated objects). A package's functions allocate dynamic variables within the package's arena.

A *program* is a collection of packages, organized as a directed package-dependence graph. This graph is statically determinable from the packages' import statements. A package `Foo` has a *direct* dependence on package `Bar` if `Foo` imports `Bar`. Package `Foo` has a *transitive* dependence on `Bar` if there is a directed path from `Foo` to `Bar` of length greater than 1 in the graph. A package's *natural* dependencies is the set of packages contained in its direct and transitive dependencies. A package `Bar` is *foreign* to `Foo` if it is not part of `Foo`'s natural dependencies.

A *closure* is a function combined with an environment that holds the bindings for its free variables. A closure belongs to the package that defines it, and it shares some of the package's natural dependencies.

An *enclosure* binds a dynamically scoped *memory view* and set of allowed system calls to a closure. The memory view defines access rights to the program's packages by the code invoked in the closure. By default, *enclosures* prohibit all system calls and limit memory views to only the resources in packages in the closure's *natural* dependencies. User-defined policies can selectively authorize system calls and restrict or extend the memory view.

2.2 *Enclosure* Expression

Enclosures are declared with the following syntax:

Stmt	::=	**with** [Policies] ClosureDef
ClosureDef	::=	**func** (args) resultType { body }
Policies	::=	MemModifiers, SysFilter
MemModifiers	::=	(pkg : U | R | RW | RWX)*
SysFilter	::=	none | all | (net | io | file | mem | ...)*

The *enclosure* expression returns a closure that is permanently associated with a memory view and system call filter. The closure can be bound to a variable and reused throughout the program's lifetime. The memory view and system call filter will be enforced during every execution of the closure.

`MemModifiers` and `SysFilter` specify the *enclosure*'s memory view and authorized system calls, respectively. `MemModifiers` extend or restrict the closure's memory view by specifying access rights, similar to those in the Unix file system, to a package: `R` grants read-only access to a package's data and constants, `RW` grants read access to its constants and read-write access to variables, `RWX` gives full access to its resources: *i.e.,* read for constants, read-write for variables, and the ability to invoke functions. `U` unmaps a package, so it is completely inaccessible in the *enclosure*.

When an *enclosure* manipulates data or functions from a foreign package, *e.g.,* passing one of its functions as a callback, the developer must explicitly specify the policies governing the closure's access. Explicit access specifications prevent accidental sharing of the foreign package's data.

`SysFilter` allows programmers to specify which system calls a closure can invoke. System calls are grouped into categories around logical services, *e.g.,* `file` for filesystem operations, `net` for network access, or `mem` for calls such as `mmap` and `mprotect`. A category included in `SysFilter` is allowed in the *enclosure*.

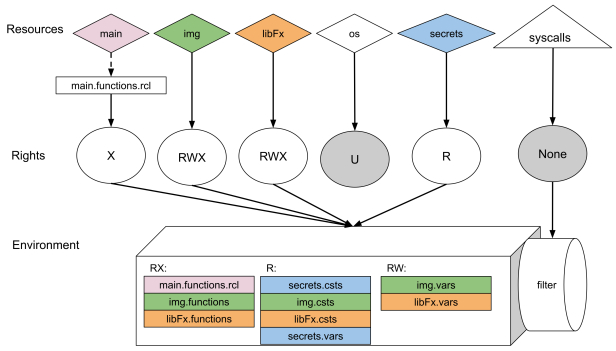

Figure 2: Programs resources made available while executing the `rcl` *enclosure* defined in Figure 1.

A call to an *enclosure* triggers a *transition* into a dynamically scoped environment restricted by its memory view and system call filter. These transitions are called *switches*. The closure runs inside this restrictive environment until it returns, thereby triggering a switch back to the caller's environment. *Enclosures* nest dynamically, but a switch can only enter an equal or more restrictive environment, preventing an escalation of privileges. It can return to a less restrictive environment. An *enclosure* faults if it violates the policies defined by its memory view and system call filter. A fault stops the execution of the closure and aborts the program.

Figure 1 presents an example of an *enclosure* in a small Go program and its corresponding directed package-dependence graph. Line 15 defines the `rcl` *enclosure* that calls the `Invert` function from the public package `libFx`. The *enclosure*'s natural dependencies are `img` and `libFx`. The `R` memory modifier extends `rcl`'s memory view to include the foreign package `secrets`, with read-only access. The `none` system call filter explicitly prohibits all system calls. At line 19, the *enclosure* computes and returns the inverse of the `original` image. As `original` belongs to `secrets`' arena, `rcl` is unable to modify it. Furthermore, `rcl`'s memory view does not include `main` or `os`, and so it would fault if it tried to access the `key` private key.

Figure 2 shows which resources belong to which package. The *enclosure* memory view and system call filter define which access to each package's resources are permitted. The `rcl` closure runs in an execution environment in which these restrictions are enforced.

2.3 Threat Model

We make no assumptions about the logic of the code running inside a restricted environment. Code from packages running in the environment can be implemented in unsafe languages, access raw memory, and execute system calls. *Enclosures* ensure that this code faults if it tries to access a package outside of its memory view or perform a prohibited system call. The developer is responsible for declaring *enclosures* to properly encapsulate untrusted code in their applications.

Enclosures assume that packages have a well-defined layout, *i.e.,* that their functions, variables (including heap), and constants can

be identified and inspected to verify that they follow the format allowed by the LitterBox backend (see §5). In particular, packages cannot share memory pages. Enforcing this assumption is the responsibility of the compiler and is verified by LitterBox at run time.

We assume that the underlying operating system and hardware are correct. Side channels such as rowhammer [34] or microarchitectural flaws [35] that modify or leak memory content are out of scope.

3 ENCLOSURE POLICIES

Enclosures can impose both fine-grain isolation policies on a single function invocation as well as program-wide policies on all uses of a package. They are similar in many aspects to program sandboxes: explicit transition into an isolated environment, a possibility of nesting restrictions, and explicit control of sharing exceptions. Because *enclosures* nest, they can be declared at any level of an application (*e.g.,* main program, a framework, or a package), which allows fine-grained tuning of isolation to the specific requirements of the code invoking the *enclosure*.

3.1 Default Policy

An *enclosure* associates a memory view, a collection of packages and respective access rights, and a system call filter with a closure and the code it invokes. The dynamic scope of this construct imposes its restrictions on the closure's natural dependencies, the code in the packages invoked by the closure. This dynamic behavior not only allows a given package to be subject to different restrictions when two *enclosures* use it, but it also allows enclosures' authors to restrict blackbox code whose source is unknown, unavailable, or too complex to manually inspect.

By default, *enclosures* prevent system calls and limit the memory view only to allow access to resources in a closure's natural dependencies. This default policy was chosen for its simplicity and usability. Other designs are possible, for example: disabling access to all packages and requiring a programmer to supply an allowed package list or allowing access to all packages and expecting a denied list. However, both alternatives require in-depth knowledge of a program's package-dependence graph and extensive, brittle annotation. By contrast, *enclosure*'s policy allows isolation of a complex subsystem without an understanding of its potentially complex and evolving dependence graph of transitively invoked packages. It treats packages as a blackbox, yet provides them with a sufficient environment to run normally.

Enclosure's default policy disables all system calls. This decision forces programmers to state their assumption of which system services a package and its dependents might reasonably execute. Once again, this choice is not intrinsic. The proposed `SysFilter` syntax could be changed to allow finer-grained filtering, for example, by filtering on system call arguments.

3.2 Program-wide Policies

Enclosures are a *local* mechanism that can enforce higher-level, program-wide policies. These policies are restrictions that apply across the full execution of a program. For example, package `Foo` should never have access to package `Bar`. An *enclosure* whose memory view unmaps `Bar` will enforce this restriction. To impose a program-wide policy, all calls into `Foo` must be enclosed. Currently, a programmer must manually insert an *enclosure* statement at each call site or provide wrappers for `Foo`'s functions that encapsulates them in *enclosures*. A compiler could automate this process by wrapping all calls into `Foo` in *enclosures* that do not allow access to `Bar`.

Program-wide policies implemented with *enclosures* allow enforcement of high-level security requirements, such as guaranteeing sensitive information's confidentiality and integrity. Confidentiality of a package's data is enforced by enclosing calls to other untrusted packages that should not access this information. Alternatively, these packages can be prevented from leaking information by disabling all system calls. A package's integrity can be ensured by mapping it read-only in the enclosed code. In Figure 1, `secrets`'s confidentiality is guaranteed by disabling all system calls for `rcl` and integrity is enforced by making `secrets` read-only. § 6 contains other examples, including one showing how *enclosures*' non-disruptive integration with programming languages can implement secured-callbacks.

3.3 Limitations

Enclosures have a few inherent limitations.

Because they operate at package granularity, *enclosures* cannot selectively share a subset of a package's code or data. This could present challenges when a particular package holds a sensitive state and is shared by mutually distrustful packages. A possible solution is to refactor the program's code to extract the package's state and split it into separate packages that can be independently shared with each distrustful parties.

A second limitation relates to information flow control. As explained above, *enclosures* can enforce the confidentiality of selected data by either not sharing it with untrusted code or disabling the *enclosure*'s system calls, thus preventing leakage. However, when enclosed code requires access to sensitive data and system calls, *enclosures* cannot guarantee that no information will be leaked. This is a challenging problem because any system call can be used as a side-channel to exfiltrate sensitive data shared through the *enclosure*. Section 6.5 provides a specific example of this situation and a mitigation.

Third, as mentioned in § 2.3, side-channels attacks and microarchitectural flaws are not addressed.

4 LITTERBOX DESIGN

Enclosures consist of two separate parts: (1) *frontend* language-specific support, provided by a language's compiler and runtime, and (2) the *backend* that uses hardware to enforce a closure's memory view and filter system calls.

LitterBox is a language-independent backend for *enclosures*. It supports diverse frontends with a simple API that offers transparent control over multiple hardware isolation technologies. Figure 3 presents a general overview of how a language frontend interacts with the LitterBox backend to implement *enclosures*.

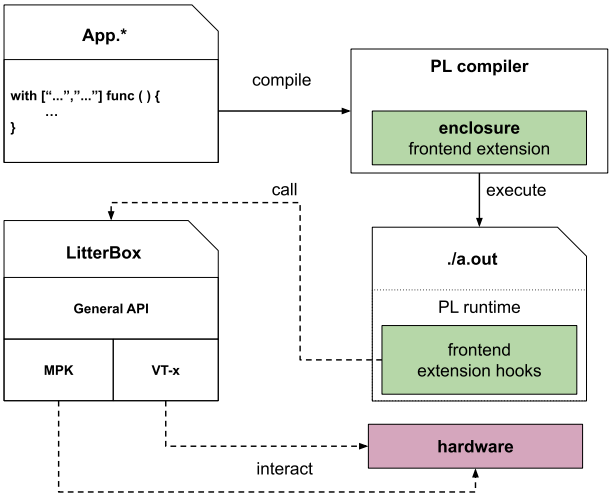

Figure 3: Overview: language support for *enclosures* with *frontend* extension inside the PL's compiler, and runtime hooks to call the language-independent LitterBox *backend*.

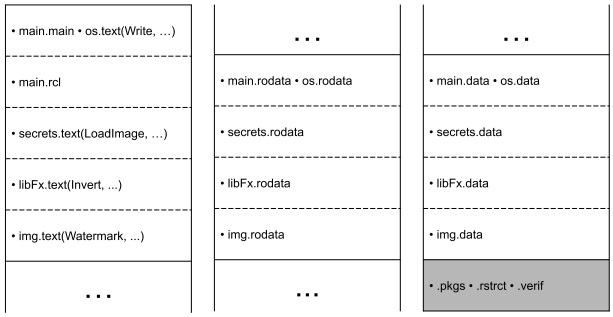

Figure 4: Figure 1's final executable content produced by Go's frontend support for *enclosures*. ELF sections from left to right: `.text (RX)`, `.rodata(R)`, and `.data (RW)`. Dashed lines represent intra-ELF section page-aligned symbol addresses, and greyed out entry the frontend's generated ELF sections for LitterBox.

4.1 LitterBox Abstractions

LitterBox defines simple system-level abstractions to represent a program's resources as sections, packages, and *enclosures*.

A *section* is a contiguous, page-aligned virtual memory region in the program's address space. Its start address, size, and default access rights (*i.e.,* read (R), write (W), execute (X)) characterize it. Sections can be dynamically allocated at run time, *e.g.,* with `mmap`.

A *package* is a collection of non-overlapping sections. It has a unique name and typically contains one or more *text* (RX), *rodata* (R), and *data* (RW) sections. A package's arena, §2.1, is part of its data sections and is not shared with other packages.

An *enclosure* consists of a unique identifier, the virtual address of its closure, its memory view as a set of package names and associated access rights, and its system call filter. The closure resides in its own text section owned by the package that declares it.

As packages partition the program's address space, LitterBox uses package dependencies to compute an *enclosure*'s complete memory view. This operation occurs at startup time for compiled statically-linked languages and package-load time for dynamic languages.

4.2 LitterBox API

LitterBox exposes a small API to the frontend language implementation. LitterBox supports *enclosure*-defined operations with four functions: (1) `Init`, (2) `Prolog`, (3) `Epilog`, and (4) `FilterSyscall`. It provides two additional functions for language runtimes: (5) `Transfer` for dynamic memory management and (6) `Execute` for user-level scheduling.

A runtime's initialization code in statically-linked languages or the package import and *enclosure*-parsing code for dynamic languages calls `Init`. It takes a description of the program's packages and *enclosures* and computes the memory views. LitterBox initializes the underlying hardware to create, for each *enclosure* memory view, the corresponding restrictive execution environment.

`Prolog` and `Epilog` provide the switch mechanism that allows a program to enter and leave an *enclosure*'s execution environment.

`FilterSyscall` is called when an *enclosure* attempts to perform a system call. It either permits the call to execute or rejects it and raises a fault.

Memory allocators require a mechanism to shrink or extend a package's arena. The `Transfer` function dynamically repartitions heap memory by transferring a section from one package to another. It permits memory allocators to reuse freed memory sections for a subsequent allocation, even across packages.

Some modern languages, such as Go, provide user-level threading to support concurrent execution. The language's runtime implements a scheduler that yields execution from one user thread to another. `Execute` enables run-time scheduling of user-level threads by providing a switch mechanism between two unrelated protection environments. The language's scheduler calls `Execute` to transition from one user thread execution environment to another. Thus, the scheduler can preempt, block, or resume an *enclosure*'s execution in the correct execution environment.

5 IMPLEMENTATION

Language support for *enclosures* requires changes to a programming language's syntax, compiler, and runtime. The full Go extension for *enclosures* is a 1,000 LOC patch to the Go compiler and runtime. The patch is modular and self-contained, so it can be easily maintained. The Python extension prototype is a fork of CPython with 600 LOC changed to introduce a multi-segmented heap for dynamic memory management. LitterBox is 6,500 LOC written in Go.

5.1 Go Frontend

Parsing: We extend Go's syntax to accept the *with* keyword using the syntax in Section 2.2. *Enclosure* policies are parsed as literals, *i.e.,* string constants. This allows the compiler to validate their satisfiability at compile time. Packages can define `init` functions to

be executed at package load time, and additional syntactic sugar is needed to tag package import statements with *enclosure* policies. This encloses the execution of the package's `init` function inside an *enclosure*. The parser also registers per-package *enclosures* and assigns unique identifiers.

Compiling: The compiler relies on the type checker to identify and register an *enclosure*'s direct dependencies and insert the `Prolog` and `Epilog` calls. It also augments calls to the dynamic allocator (`mallocgc`) with the caller's package identifier. The compiler outputs one *code object* per package that contains the expected `.text` (functions), `.data` (global variables), and `.rodata` (constants) sections, as well as a `.rstrct` section containing the package's *enclosures* configurations and direct dependencies.

Linking: The linker has global knowledge of the program's package-dependence graph and assembles packages' code objects into a single executable. For each code object, it extracts the `.rstrct` sections, computes every *enclosure*'s memory view, and marks packages that appear in at least one *enclosure*. The linker's symbol address-assignment algorithm segregates marked packages resources in separate sections so that no two marked packages overlap. *Enclosure* closure functions are isolated into their own memory sections.

The linker outputs three distinguished ELF sections as part of the executable. The `.pkgs` and `.rstrct` sections hold descriptions of packages and *enclosures* to be passed to LitterBox's `Init` during runtime initialization. A `.verif` ELF section stores the call-sites to LitterBox hooks, which LitterBox uses to filter API calls at run time. Figure 4 illustrates the executable corresponding to Figure 1's program.

Runtime: Go's dynamic memory allocator divides the heap into class-size sections, called spans, cached per hardware thread and used to satisfy allocations based on the requested size. The *enclosure*-extension adds a level of indirection by dynamically assigning spans to packages' arenas. After adding a span to a given arena, the runtime calls LitterBox's `Transfer`.

Go scheduler *enclosure*-extension maintains a mapping between a routine and the corresponding execution environment and relies on split-stacks to isolate frames preceding the *enclosure*'s call. To avoid escalation of privilege attacks, execution environments are transitively inherited by goroutine creation so that user-level threads created inside an *enclosure*'s environment continue to execute in the same environment. The scheduler uses the `Execute` hook to switch between goroutines associated with different environments. Similarly, garbage collection needs full access to the program's resources but executes on top of runtime goroutines associated with a trusted execution environment.

5.2 Python Frontend

LitterBox can support dynamic programming languages. Our Python prototype is based on a fork of CPython 3.9.1. Rather than repeating implementation details common to the Go frontend (parser extension, or instrumenting *enclosures* bodies), this section focuses on challenges attributable to: 1) Python's dynamic behavior, and 2) CPython's implementation.

Dynamic behavior: Python is a dynamic language that accumulates and processes knowledge about a program during its execution. Modules (packages) are lazily imported when a file is parsed and functions are compiled only when needed. As a result, and unlike Go, LitterBox must accept multiple calls to `Init`, each of which provide only partial information about a program. CPython's import mechanism registers modules and their direct dependencies with LitterBox. Similarly, the compiler registers *enclosures* and their direct dependencies as they are compiled.

In this dynamic setting, and unlike Go, LitterBox, not the compiler, must compute the transitive dependencies of modules and *enclosures* full memory views. Furthermore, the execution of an *enclosure* can trigger new imports, so LitterBox's default policy makes these new packages available to the executing *enclosure*, unless explicitly restricted by user policies.

Python provides little control over dynamic memory allocation. The language does not offer an equivalent of Go's `new` or C's `malloc` functions to identify dynamic allocation. Without explicit allocation, it is difficult for a programmer to encapsulate data in a specific module. To allow a programmer to express this intent, we implemented `localcopy`, a function similar to Python's `copy.deepcopy`, which creates an object copy in the caller's module.

CPython internals: CPython is the reference implementation for the Python programming language. It is a highly optimized, complex system that presents some challenges in providing strong isolation guarantees.

CPython's default object memory allocator is a singleton whose state resides in global static variables. Its design is very similar to Go's, in that it manages `mmap`ed arenas divided into class-sizes. We made some small changes to encapsulate the allocator's state in a structure, which allowed multiple allocator instances to co-exist with non-overlapping arenas. This in turn enabled us to assign a memory allocator per module and segregate objects allocated by different modules on distinct memory pages. Our memory allocator further distinguishes functions (code) and objects (data) in separate arenas within one module. This allows LitterBox to hide a module's functions when the module is mapped without execution rights, while still allowing access to its data.

For performance reasons, CPython co-locates data and metadata, specifically the reference counting counters in the headers of objects. While efficient, this implementation decision makes it difficult for an isolation mechanism to enforces read-only semantics on an object, as it would preclude updating reference counts. Similarly, the CPython generational garbage collector (GC) embeds a linked list's `next` pointer inside object, which might be inaccessible within an *enclosure*. To circumvent these problems, our CPython extension performs a controlled switch to a trusted environment, with full access to program resources, to modify reference counts in read-only objects or enqueue on the GC linked lists. While sufficient for a prototype, this approach is expensive as the full cost of two switches is incurred on every access to read-only objects. In the future, our Python extension will separate objects' data and metadata.

5.3 LitterBox Implementation

LitterBox provides support for two hardware-enforced isolation mechanisms: Intel VT-x (LB^{VTX}) and Intel MPK (LB^{MPK}). While these two technologies differ greatly, LitterBox provides a common implementation and only differentiates between the selected

hardware for three operations: (1) creating and enforcing an execution environment (`Init`, `FilterSyscall`), (2) extending a package's arena (`Transfer`), and (3) performing a switch between execution environments (`Prolog`, `Epilog`, `Execute`).

Common Aspects: LitterBox validates the configuration passed to `Init` by ensuring that sections are aligned and non-overlapping and that the memory views and authorized system calls can be satisfied. At that point, LitterBox performs an important optimization by clustering the packages across all memory views that have the same access rights. This clustering creates larger, logical *meta-packages* that can be efficiently managed. LitterBox derives the set of memory sections and associated access rights that, together with system call filters, define the *enclosure* execution environment *description*.

LitterBox code and data consist of two packages, namely `user` and `super`. The `user` package is available in all execution environments and provides authorized access to `Prolog`, `Epilog`, `Execute`, and `Transfer` hooks. The `super` package contains the *enclosures* definitions, the verification list of allowed call-sites to the API, and the descriptions. It also handles the logic that validates calls to the LitterBox API, modifies execution environments, and performs switches.

Both hardware implementations use execution environment descriptions to initialize their underlying hardware and create different execution environments.

LB$^{\text{VTX}}$: Intel Virtualization Technology extension (VT-x) [60] extends the x86 ISA to simplify hypervisor implementation. It relies on an extended-page-table (EPT) in hardware to map host virtual (HVA) and guest physical addresses (GPA). It defines VMX root for a hypervisor with unmodified CPU behavior and non-root mode for guest operating systems with restricted CPU behavior. The non-root mode has access to the virtual machine's CR3 register and can manage its guest virtual (GVA) to GPA mappings.

LB^{VTX} relies on Linux's Kernel-based Virtual Machine (KVM) module [2] for Intel VT-x to create a virtual machine (VM) in which the application executes. An execution environment in the context of LB^{VTX} is a page table mapping that enforces the *enclosure* description, in other words, its memory resources are associated with the correct access rights in user-space. LB^{VTX} creates a separate page table for each *enclosure*. It also allocates one trusted page table with user-access to all packages except LitterBox's `super` to run non-enclosed code. Finally, `super` is mapped in the guest kernel address space (non-root kernel mode) and implements the guest operating system. For simplicity, LB^{VTX} strives to preserve $GPA == GVA == HVA$ whenever possible. It only breaks the invariant $GPA == GVA$ when necessary to circumvent VT-x's 40 bits physical address space. Once all execution environments are initialized, LB^{VTX} enters the VM and resumes the application's execution in non-root user mode, with the trusted page table mappings.

To perform a switch, LitterBox functions perform a specialized system call to our guest operating system. The system call handler has access to the `super` package, and checks that the call-site to LitterBox's API corresponds to the program's specifications supplied to the `Init` function, which is found in the `.verif` ELF section. If the transition is authorized, the guest operating system switches the VM's CR3 register (the page table root) to the target execution environment and returns (`iret`). Using a single VM per application and implementing switches as system calls, rather than instantiating a VM per *enclosure*, reduces both the complexity of LitterBox management of KVM state and the overhead of switches because a syscall is less costly than a VM EXIT.

`Transfer` is also implemented as a system call that updates the relevant execution environments' page tables. Similar to switches, the call-site and the validity of the arguments are checked before applying the desired modifications.

The handler filters system calls according to the current execution environment's filter. If authorized, system calls are passed through to the host [11] via a hypercall (`VM EXIT`). The system call is performed in root user mode, which then returns to the VM with the results (`VM RESUME`).

A fault triggers a `VM EXIT`, prints a trace of the root-cause, and stops the program's execution.

LB$^{\text{MPK}}$: Intel Memory Protection Keys (MPK) [33] extend the x86 ISA to enforce memory page protections without domain switches. Page table entries are tagged using 4 previously ignored bits to encode 16 different tags, called keys. A new user-writable and readable register, `PKRU`, uses two bits per key (32 bits total) to encode access and write capabilities for pages tagged with the corresponding keys. Hardware enforces PKRU permissions on data access. The Linux kernel provides system calls to manage keys, *i.e.,* allocate and free, and the `pkey_mprotect` system call to tag a range of addresses with a key.

LB^{MPK} relies on Intel MPK to isolate *enclosures*. It allocates one key for each meta-package. In practice, clustering packages results in fewer than 16 meta-packages whose views fit into the 16 keys. Libmpk [51]'s key virtualization could be used to overcome Intel MPK's limitation if the need arises. Similar to Erim [61], LB^{MPK} scans the program to ensure that only the LitterBox package modifies the PKRU register. As in LB^{VTX}, all calls to the API are checked against the verification information stored in `super`.

An execution environment for LB^{MPK} is simply the PKRU register's value that encodes access rights for all meta-packages.

A switch validates the transition using `super`'s verification and writes the PKRU register.

A transfer is slightly more complicated as it must invoke a `pkey_mprotect` system call to update the relevant page table entries' key.

System calls are filtered by translating the `FilterSyscall` function into a BPF filter loaded via seccomp [6, 39], which indexes the current environment (from the PKRU value) to a mask of permitted system calls. We use a Linux kernel patch [45] to expose the `PKRU` register to seccomp. As Intel MPK is an emerging technology, we consider it a reasonable assumption that future versions of the kernel will incorporate a similar mechanism. An alternative would be to implement techniques similar to Erim [61] or rely on a BPF map updated upon switches.

A fault in LB^{MPK} stops the program's execution.

6 EVALUATION

This evaluation is performed on an Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz running Ubuntu 20.04 LTS with Linux kernel version 5.4.0-42-generic, and a patch [45] to access `PKRU` value in

	Baseline	LB^{MPK}	LB^{VTX}
call	45	86	924
transfer	0	1002	158
syscall	387	523	4126

Table 1: Microbenchmarks results in nanoseconds.

seccomp. We report numbers for the Go frontend implementation based on LitterBox. The evaluation is divided into two parts: (1) microbenchmarks to measure the cost of LitterBox's fundamental operations with both hardware enforcement mechanisms, and (2) macrobenchmarks to study *enclosure*'s usage in realistic applications, divided into a qualitative and quantitative studies.

As a baseline, we report unmodified Go performance, noted as Baseline, where *enclosures* are replaced by vanilla closures. LitterBox's Intel MPK hardware enforcement is reported as LB^{MPK}, and Intel VT-x as LB^{VTX}. All benchmarks run single threaded in order to accurately quantify the overheads of domain crossings (*i.e.,* switches).

6.1 Microbenchmarks

We rely on microbenchmarks to answer the following questions: (1) What is the cost of performing a call to an *enclosure*? (2) What is the basic cost of memory management calls to the `transfer` LitterBox's hook? (3) What overheads does LitterBox impose on system calls?

To answer each of these question, this evaluation uses three microbenchmarks to measure LitterBox's overheads:

- `call`: measures the time required to call and return from an empty *enclosure*.
- `transfer`: calls LitterBox's `Transfer` on a 4-page memory section.
- `syscall`: an *enclosure* performs a `getuid` system call in a loop.

We run each microbenchmark a million times and report the median latency value, in nanoseconds, in Table 1. These latencies are shared by the Go and Python frontends, as they both use LitterBox's backend.

call: The cost for the Baseline is 45*ns* and 86*ns* and 924*ns* for LB^{MPK} and LB^{VTX} respectively. This translates to an overhead of ~40*ns* (86-45) per *enclosure* call for LB^{MPK} and less than 1 µs (924-45) for LB^{VTX}. LB^{MPK} is thus able to perform a single switch in approximately 20*ns* by writing the `PKRU` register. LB^{VTX} switch depends on a system call to change the `CR3` register, so effectively we measure the cost of two system calls.

transfer: The relative performance of each backend changes when it comes to memory management. LB^{VTX} is able to efficiently transfer a memory span by toggling the presence bits in the corresponding page tables. LB^{MPK}, however, requires a `pkey_mprotect` system call, which is ~6 times slower than LB^{VTX}.

syscall: LB^{MPK} incurs negligible overheads as system call filtering requires a few operations to accept or reject a system call based on the PKRU value. LB^{VTX} relies on hypercalls to service system calls and pays the full cost of a `VM EXIT` of ~4µs. This approach is similar to other container technologies such as gVisor [69].

These benchmarks suggest that both implementations impose reasonable overheads when it comes to an *enclosure* call, as these can potentially be amortized by the closure's service time. While LB^{VTX} more efficiently handles memory sections being transferred between packages, LB^{MPK} wins when it comes to filtering and executing system calls. Thus, depending on application characteristics, users can make an informed decision on which version of LitterBox to use.

6.2 Macrobenchmarks

This section uses popular Github Go packages to benchmark the performance of small applications derived from each package's "hello world" sample, to determine the worst-case performance overheads of LitterBox. In these applications, *enclosures* are used, in very different ways, to safely leverage the *unmodified* public package. Table 6.2 reports the achieved raw performance with Go Baseline, LB^{MPK}, and LB^{VTX}, and respective slowdowns for each benchmark.

Reducing the application's TCB: *Enclosures* drastically reduce the trusted codebase (TCB), *i.e.,* code executing with full access to the program's address space and syscall API. As shown in Table 6.2, each application consists of less than a hundred lines of code (LOC) that import thousands of LOC from public dependencies. In every macrobenchmark, a single *enclosure* declaration, using the default policy, completely encloses public library code and its transitive dependencies, thus preventing any public package from accessing and leaking sensitive information held by the application.

Processing Sensitive Images with Bild: Bild [55] is a popular Go Github public package for parallel image processing. While presenting attractive functionalities, such as an `Invert` function for images, bild silently drags-in over 160K lines of code of unverified origin. This is an daunting quantity of potentially harmful code to examine while writing a simple 32 LOC application that loads and inverts an image. We declare an *enclosure* to enclose the call to bild's `Invert` function. We further disallow all system calls and extend the *enclosure* memory view with read-only access to the `main` package that holds the sensitive image to invert. This simple benchmark is the textbook example of how sensitive information can be safely exposed to an untrusted package, while preventing modifications of program state or leakage (*e.g.,* via system calls).

The benchmark is purely computational and memory-intensive as it allocates and computes an inverted image. LB^{VTX} shows a mere 5% slowdown. As predicted by microbenchmarks in §6.1, LB^{VTX} overhead to call an *enclosure* is absorbed by the closure's service time and its efficient mechanism for transfers. LB^{MPK} achieves a respectable 12% slowdown, attributable to the cost of frequent transfers to populate the arena with memory spans of various sizes to satisfy bild's dynamic memory allocations.

Securing an HTTP server: Go provides an HTTP server implementation in the `net/http` package. A typical concern in web-facing applications with TLS support is to protect private keys and certificates from potential attacks delivered via user requests. Such attacks can, for example, attempt to trigger a buffer-overflow in the request-handler to leak sensitive data. This benchmark defines the request handler as an *enclosure* with no access to the packages used by `net/http` and no system calls. To measure raw overheads, the

	Baseline	LB^{MPK}		LB^{VTX}		Benchmark information				
	raw	raw	slowdown	raw	slowdown	App TCB #LOC	Enclosed #LOC	#Stars	#Contributors	#Public deps
bild	13.25ms	14.88ms	**1.12x**	13.91ms	**1.05x**	32	166K	2.9K	15	1
HTTP	16991reqs/s	16738reqs/s	**1.02x**	9560.14reqs/s	**1.77x**	31	-	-	-	-
FastHTTP	22867reqs/s	22025reqs/s	**1.04x**	11375reqs/s	**2.01x**	76	374K	13.1K	100	3

Table 2: Macrobenchmarks results.

handler's logic only selects a 13KB in-memory static HTML page to service the request. This is a typical use of *enclosures* to prevent potentially harmful code from accessing sensitive resources.

Once again, the observations made in §6.1 are confirmed. As the benchmark is primarily dominated by socket operations, LB^{VTX}'s high overhead in servicing system calls introduces a 1.77× slowdown. This time, as the *enclosure* does not perform dynamic memory allocations, LB^{MPK} is able to perform almost as well as the baseline.

Using a Public HTTP Framework: FastHTTP [62] is an industry-grade Github public Go package that implements a performance-oriented HTTP server. FastHTTP offers high throughput, as long as we can trust over a 100 programmers and more than 350K LOC. To prevent FastHTTP from accessing an application's sensitive resources, we create and run the server in an *enclosure*, only allowed to perform `net`-related system calls (*i.e.,* socket operations). The *enclosure* forwards requests to a trusted handler goroutine via go channels. This benchmark shows how trusted callbacks can easily be implemented. To measure overheads precisely, the trusted handler simply returns 13KB static HTML pages as before. In a more realistic deployment, the handler would access a private database or other sensitive information, which would be completely unavailable to the *enclosure* running the FastHTTP server.

Similar to the simple HTTP experiment, LB^{MPK} achieves a throughput comparable to the baseline. We observe a small slowdown that seems to be due to the server's consumption of dynamic memory. This cost is however greatly diminished by FastHTTP efficient usage of memory, *e.g.,* `HTTPRequest` object reuse across requests. This allows LB^{MPK} to avoid numerous costly transfers. LB^{VTX} has a 2× slowdown due to system calls. Note that the LB^{VTX} slowdown in FastHTTP is larger that in HTTP. This is not due to an increase in the frequency of system calls as FastHTTP and HTTP have a similar system call trace. However, FastHTTP service time to accept connections and parse requests is significantly smaller, while the system call overhead remains the same.

6.3 Usability

We consider a wiki-like web-app [7, 56] that stores its pages in a Postgres database, as depicted in Figure 5. This web-app is written in Go, relies on the deprecated `pq` [22] public library as a Postgres driver and on the `mux` [58] package to route HTTP requests consisting of `GET` (read pages) and `POST` (create a page). Together, `pq` and `mux` incorporate 44 public Github packages as dependencies.

To prevent any public package from subverting our application, we rely on two *enclosures* that communicate with trusted code via Go channels.

First, the HTTP server Ⓑ consisting of `mux` and its transitive dependencies is enclosed without access to the database, the file-system, or the rest of the application holding sensitive information,

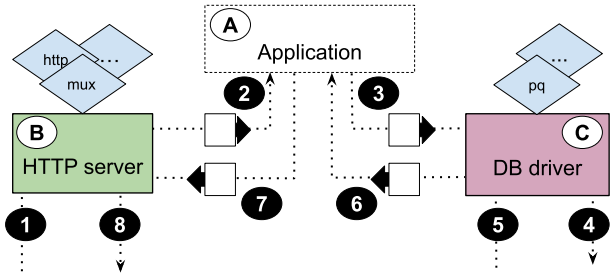

Figure 5: *Enclosures* isolating the HTTP server and the database driver in a wiki-like web application.

e.g., page templates and the database password. It is however authorized to create and read ①/write ⑧ to its own network sockets. Similar to the FastHTTP experiment, HTTP handlers forward parsed requests to trusted code on a private Go channel ②.

Second, `pq` and its natural dependencies are isolated in an *enclosure* Ⓒ, acting as a proxy server only allowed to communicate with Postgres via a pre-defined network socket. This enclosure has no access to the HTTP server's logic, the file-system, and network operations on other sockets. This database proxy server accepts SQL requests on a Go channel ③, communicates them to Postgres, ④ and ⑤, and returns the result to trusted code ⑥.

The trusted code base, *i.e.,* non-enclosed Ⓐ, consists of the application's glue code, responsible for reading requests forwarded by the enclosed handlers ②, contacting the enclosed database proxy server ③, validating the SQL query result ⑥, and generating and forwarding ⑦ the HTML response.

The throughput slowdown is similar to the one in the FastHTTP experiment.

6.4 Python *Enclosures*

The CPython extension is an unoptimized prototype. While § 6.1 presents reasonable overheads for LitterBox basic operations, we would like to study how the challenges described in § 5.2 affect the performance of Python programs. This section quantifies the impact of these limitations on the Python *enclosures* performance and provide insight into improving *enclosure* on this platform.

Consider a Python program with a single *enclosure* that encapsulates the use of the `matplotlib` module. User sensitive data from a `secret` module is shared read-only with a closure that generates a plot from the data and writes the result to disk. The experiment runs using LB^{VTX} to understand the relative impact of each overhead, including system calls.

We first try a conservative approach where each reference count operation and garbage collection triggers a switch to a trusted environment before returning to the *enclosure*, as described in § 5.2. This

experiment shows a ~18x increase in execution time for *enclosures*, as compared to standard Python. We measure nearly 1M switches due to reference counting and garbage collection. The delayed initialization of the *enclosure* environment, including computation of package dependencies, *enclosures* memory views, and configuration of the underlying hardware mechanism (KVM), represents 4.3 percent of the measured slowdown. System call overheads (requiring a VM exit) account for less than 1 percent of the slowdown. This can be explained since the system calls (futex and write) have a service time larger than the system call overhead measured in § 6.1.

We run a second experiment to simulate changes that allow updating a read-only object's reference count without a switch. To do so, the `secret` module is mapped with read-write access and switches for reference count operations are disabled. The measured slowdown is now ~1.4x and is dominated by the delayed initialization cost. Note that this cost has to be paid once, at the first invocation of an *enclosure* and can be amortized if the *enclosure* is called multiple times.

From this experiment, we believe that decoupling CPython data and metadata would enable more efficient support of *enclosures* and should be the main focus of future work.

6.5 Security

This section shows that *enclosures* can address the threats from the malicious packages cited in §1 [15–18]. To this end, we re-created Python and Go packages that perform the same attacks as the original malicious ones. These attacks mostly access local secrets, either within the program's memory or on the local file system (*e.g.,* private SSH keys), and attempt to exfiltrate them via the network or open backdoors on the local system.

Enclosures easily detect and protect against most attacks with a basic configuration, *i.e.,* the default memory view and limited system calls, while still allowing valid behaviors to run successfully. However, a few packages [15, 17] presented a challenge. These packages provide a valid functionality that requires access to a secret and system calls that could be used to exfiltrate sensitive data. For example, the `ssh-decorater` package [15] allows SSHing to a given IP address and executing python commands on the remote server. The public library was, however, infected with malicious code exfiltrating user credentials to another server, via a `POST` request. To prevent this attack, we modify the application code to pass a preallocated socket and private key to the enclosed `ssh-decorater` public package, therefore enabling us to disable socket creation and file-system access. Another solution extends the sysfilter categories to only allow connect system calls to a list of pre-defined IP addresses, allowing us to grant socket creation and file-system access to `ssh-decorater`, while preventing it from contacting a malicious server. Note, however, that the valid remote host can still be used as a relay to send the credentials to the malicious server.

A similar issue arose with malicious clones of the Python Django framework. To protect against these, we took an approach similar to the one used in FastHTTP with secured callbacks.

7 DISCUSSION

Granularity: Packages' composition and size make possible effective isolation inside an application's address space. Their coarser granularity is far easier to manage than individual objects, and better fits the granularity of page-based hardware isolation mechanisms. Moreover, packages can often be clustered into efficient meta-packages, as explained in §5.3. Clustering reduces the number of keys needed to tag an entire address space and, in many cases, fits into the 16 possible Intel MPK keys.

Explicit scoping: *Enclosures* utilize dynamic scoping to control the application of restrictions on program resources. *Enclosures* isolate untrusted packages by explicitly enclosing invocations of their functions. An alternative approach to protect against public packages could automatically enforce a transition to a restricted execution environment on every single untrusted package function invocation. This approach is, however, limiting as compared to *enclosures* for 3 reasons: 1) it requires modification of untrusted package code, 2) it imposes a switch per call into a package, prevents programmers from controlling switches, and might result in large overheads, and 3) *enclosures* can emulate this approach, as mentioned in § 3.2.

Hardware enforcement: As the need for in-application isolation grows, hardware should evolve to provide efficient, reliable, and easy-to-use enforcement mechanisms. Intel MPK offers a first-generation solution that exploits unused bits in the `PTE` to store protection keys on existing hardware. However, Intel's decision to permit key modification by unprivileged code is debatable, especially given system calls' low cost. An ideal solution would combine MPK's low overheads and ease-of-use with VT-x's robust protection model, scalability to multiple address spaces (each with 16 keys), and the ability to filter system calls in a protected library operating system [11]. This last aspect is similar to the gVisor/Sentry mechanism for containers [69].

Capabilities: Capability support, as proposed by CHERI [67] or CODOMs [63], is an attractive approach that offers simplicity, expressiveness, and strong guarantees at the cost of more substantial hardware changes. LitterBox, with its decoupling of the API and hardware implementations, could support capabilities in the future.

8 RELATED WORK

Isolation is the combination of a policy (what is isolated) and mechanism (how is the isolation enforced). The system may restrict interactions for various reasons, such as limiting error propagation or constraining untrusted components. LitterBox focuses on untrusted packages that share the same address space as a trusted program. Intra-address space isolation has been studied along different dimensions, we list here key differentiating factors.

Operating system mechanisms: The most common software isolation mechanism is operating system processes. While the application and its packages could be partitioned into separate processes, the cost of IPC and the complexity of argument marshaling along with the high implementation complexity limit this approach to a few examples such as web browsers [41].

Previous work proposed incorporating some process isolation and control mechanisms within an address space [13, 32, 40]. Wedge's sthreads restrict memory accesses and system call capabilities associated with a thread [13]. LWC's light-weight contexts extend this approach by providing control over the memory view, system call capability, and execution state inside an object [40]. SMV's

secure memory views provide a more straightforward approach, a uniquely identified memory domain that can be accessed when it is attached to the current thread [32]. Common to all these approaches is that they require code refactoring to use the new mechanisms.

By contrast, *enclosures* offer a more natural separation that closely integrates with the language. Unlike LWC, a developer does not need to be aware of all of a package's transitive dependencies or their layout in memory and can, with a single line of code, completely encapsulate them. Moreover, LitterBox could employ these systems to enforce memory isolation and syscall filtering. LWC presents an interesting OS abstraction and could provide an alternative LitterBox backend that does not require specialized hardware (*e.g.,* Intel VT-x).

Virtualization mechanisms: Virtualization enables a hypervisor to apply different access permissions to a collection of memory pages shared among several virtual machines. These VMs can encapsulate code running in the same address space, with different access rights. Dune [11] uses Intel VT-x [60] to virtualize the process abstraction and isolate software components. SIM [53] with Intel VT-x protects a trusted security monitor running in an untrusted guest. Nexen [54] and HyperSafe [65] focus on protecting and isolating hypervisors. LXD [47], Nooks [57], and Nested Kernel [21] isolate kernel submodules. TrustVisor [44] is a thin hypervisor that isolates portions of an application. SeCage [42] takes a data-driven approach, by automatically partitioning an application into security domains, based on the secrets they access, and isolates them from each other with Intel VT-x.

Unlike Secage, *enclosures* enforce security domain boundaries based on packages. This simplifies an application's partitioning, especially for environments where static and dynamic analysis is hard. *Enclosures* make it easier for developers to reason about isolated compartments and prevent accidental sharing of sensitive data, *e.g.,* via valid pointer references passed to the untrusted package. *Enclosures* further allow fine-grain schemes, such as exposing sensitive data to untrusted packages, while preventing it from being modified.

Virtualization generally incurs (high) performance overhead due to extended page tables and hypercalls. Intel MPK provides more specialized hardware support with lower overheads. For example, Hodor [31] shows that Intel MPK [33] isolates data-plane libraries with far smaller overheads than Intel VT-x.

Similarly, *enclosures* based on LitterBox support both Intel VT-x and Intel MPK as isolation backend mechanisms.

Programming language and runtime mechanisms: Programming languages provide information hiding and abstraction mechanisms, a weak form of isolation. These are software-engineering mechanisms meant to decouple components, rather than robust run-time isolation mechanisms. The frequent escapes from unsafe languages or run-time reflection demonstrates that this cat-and-mouse game is not a real security solution.

Software Fault Isolation (SFI) [64] adds memory isolation to unsafe languages by inserting dynamic checks on accesses. Control-Flow Integrity (CFI) [8] guarantees that only valid code areas can be executed. Together, CFI and SFI guarantee that only a subset of the application may access the protected memory region. However, these techniques require non-trivial static and dynamic analyses with non-negligible costs in terms of complexity and overheads. An alternative solution embodied by NaCl [68], XFI [73], WebAssembly [30], and CloudFlare's [71] use of V8 isolates [26] are akin to Proof-Carrying code [48] and enforce restrictions on which machine instructions can access segmented memory [52].

Enclosures built on top of LitterBox combine several of these approaches. Their close integration in a programming language allows statically delimited memory view boundaries and control of the generated code. Similar to PCC, the resulting binary abides by a certain format, verified by LitterBox along call-sites to its API, and leveraged to efficiently apply hardware-enforced isolation.

Language-based hardware-enforced mechanisms: Isolation can be provided by extending a language with security domains and enforcing isolation with specialized hardware. SOOAP [29] is a security analysis framework that relies on annotations to help refactor and compartmentalize existing applications. It could be used in conjunction with LibMPK [51], a library that virtualizes and manages MPK keys, to provide strong isolation guarantees. Erim [61], using Intel MPK, and Shreds [20], using ARM memory domains [10], expose isolated memory pools and associate code allowed to access them. Memsentry [36] provides an LLVM pass to implement data encapsulation, selectively enforced by MPK or other technologies. Similarly, Glamdring [38] relies on annotations to mark sensitive data and isolate code accessing it inside Intel SGX [1] enclaves. Glamdring and *enclosures*' specifications differ as they solve two (different) problems. *Enclosures* annotate the entry points of top-level packages, which are the fundamental abstraction they isolate, as secrets are for Glamdring. Gotee [25] isolates trusted code with its own memory pool inside SGX enclaves. JITGuard [23] relies on Intel SGX to protect jitted code.

In essence, most of these solutions focus on data encapsulation. *Enclosures* take a different approach: (1) resource partitioning closely follows the natural static package-dependence graph of the program; (2) *enclosures* promote packages to a higher-level construct embodying a basic unit of resources programmatically manipulated by developers to be isolated or shared according to strict policies. Instead of a strict "all-or-nothing" partitioning, *enclosures* provide a way to compose packages to form the memory view exposed to a closure.

Hardware extensions: Appropriate hardware support would make it easy to isolate packages. Multics's segments [12], implemented on the vintage GE 645, provide fine-grain access control appropriate for *enclosures*. More recent work has proposed extensions to modern processor to control memory accesses within a single address space. Mondrian memory protection [66] (MMP) implements word-granularity hardware-enforced memory isolation. Its permission control granularity would be ideal for isolating program objects. IMIX [24] and MicroStache [46] extend the Intel x86 ISA with instructions to access safe memory regions. CODOMs [63] tags code pages with keys delimiting the memory resources and privileged instructions they are allowed to use. Capability Hardware Enhanced RISC Instructions [67] (CHERI) is a hardware extension that allows fine-grained compartmentalization and enforces spatial, referential, and temporal memory safety. CHERI operates at the object-level and requires a deeper understanding and instrumentation of third-party packages than *enclosures*.

As LitterBox exposes a stable high-level API and hides hardware details, any of these technologies could be added as a hardware-enforced isolation mechanism without requiring any changes to the application or the programming language itself. CODOMs and CHERI expressiveness makes them particularly appealing candidates. CHERI could be used as a non-page based LitterBox backend, which would reduce memory fragmentation or allow to discriminate access to CPython's data and metadata while keeping them co-located (§ 6.4).

Enclosures' most similar project is Verona [28], a recently open-sourced Microsoft project that introduced a safe, infrastructure programming language. Like *enclosures*, it provides linear regions that compartmentalize legacy components by encapsulating their code, data, and dynamic allocations. The language executes unsafe components in sandboxes and looks to using CHERI in the future. Unlike *enclosure*, and to the best of our knowledge, Verona does not provide developers with a fine-grain method to composing access policies to the program's resources on a per code invocation-basis, and requires the application's main logic to be rewritten in a different language.

9 CONCLUSION

Enclosures provide a mechanism to execute untrusted packages inside a restricted environment, easily tunable by programmers, that limits access to a program's memory and its system resources. Using packages, and their transitive dependencies, as the basic unit of shareable resources results in easy-to-understand and manipulate isolation boundaries within an application.

Enclosures are language-independent and make no assumptions about the safety of the code. Instead, LitterBox provides support for *enclosure* policies based on hardware-based isolation mechanisms.

Our evaluation proves that *enclosures* can be efficiently added to Go, a language with a complex runtime, and provide robust isolation guarantees using both Intel VT-x and Intel MPK. Our Python implementation confirms the generality of the approach and support for dynamic languages.

ACKNOWLEDGMENTS

We thank the ASPLOS anonymous reviewers and our shepherd Mengjia Yan for the detailed comments. Moreover, we thank our master students, Charly Castes and Elsa Weber, who prototyped the Intel MPK support and the Python frontend for *enclosures*. This work was funded in part by a VMware Research Grant and the Microsoft Swiss JRC TTL-MSR project.

REFERENCES

[1] [n.d.]. Intel SGX - Software Guard Extensions Programming References. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
[2] 2020. Linux Kernel-based Virtual Machine. https://www.linux-kvm.org.
[3] 2020. Python Packge Index. https://pypi.org/.
[4] 2020. Rubygems stats. https://rubygems.org/stats.
[5] 2020. Rust The Cargo Book. https://doc.rust-lang.org/cargo/commands/.
[6] 2020. Seccomp BPF. kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.
[7] 2020. Writing Web Applications Golang. https://golang.org/doc/articles/wiki/.
[8] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity.. In *ACM Conference on Computer and Communications Security*. 340–353.
[9] Adrien Ghosn. 2020. Enclosures: language-based restriction of untrusted libraries. https://github.com/aghosn/enclosures.
[10] ARM. 2020. ARM1136JF-S and ARM1136J-S Technical Reference Manual. https://developer.arm.com/documentation/ddi0211/latest/.
[11] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features.. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*. 335–348.
[12] A. Bensoussan, C. T. Clingen, and Robert C. Daley. 1972. The Multics Virtual Memory: Concepts and Design. *Commun. ACM* 15, 5 (1972), 308–318.
[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments.. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*. 309–322.
[14] Catalin Cimpanu. 2018. Somebody tried to hide a backdoor in a popular javascript npm package. https://www.bleepingcomputer.com/news/security/somebody-tried-to-hide-a-backdoor-in-a-popular-javascript-npm-package/.
[15] Catalin Cimpanu. 2019. Backdoored Python Library Caught Stealing SSH Credentials. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/.
[16] Catalin Cimpanu. 2019. Malicious Python libraries targeting Linux servers removed from PyPi. https://www.zdnet.com/article/malicious-python-libraries-targeting-linux-servers-removed-from-pypi/.
[17] Catalin Cimpanu. 2019. Twelve malicious Python libraries found and removed from PyPi. https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/.
[18] Catalin Cimpanu. 2019. Two malicious Python libraries caught stealing SSH and GPG keys. https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/.
[19] Catalin Cimpanu. 2020. Malicious npm packages caught installing remote access trojans. https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/.
[20] 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*. 56–71.
[21] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. 2015. 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)*. 191–206.
[22] Daniel Farina. 2020. pq - A pure Go postgres driver for Go's database/sql package. https://github.com/lib/pq.
[23] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX.. In *ACM Conference on Computer and Communications Security*. 2405–2419.
[24] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-Process Memory Isolation EXtension.. In *Proceedings of the 27th USENIX Security Symposium*. 83–97.
[25] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 571–586.
[26] Google. 2020. Chromium V8 isolates. https://chromium.googlesource.com/chromium/src/+/master/third_party/blink/renderer/bindings/core/v8/V8BindingDesign.md#Isolate.
[27] Google. 2020. Golang add dependencies to the module and install them. https://golang.org/cmd/go/#hdr-Add_dependencies_to_current_module_and_install_them.
[28] Microsfot Confidential Computing group. 2020. Project Verona: a programming language for the modern cloud. https://www.microsoft.com/en-us/research/project/project-verona/.
[29] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP.. In *ACM Conference on Computer and Communications Security*. 1016–1031.

[30] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly.. In *Proceedings of the ACM SIGPLAN 2017 Conference on Programming Language Design and Implementation (PLDI)*. 185–200.

[31] 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.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 489–504.

[32] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications.. In *ACM Conference on Computer and Communications Security*. 393–405.

[33] Intel. 2020. Intel®64 and IA-32 Architectures Software Developer's Manual.

[34] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*. 361–372.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution.. In *IEEE Symposium on Security and Privacy*. 1–19.

[36] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware.. In *Proceedings of the 2017 EuroSys Conference*. 437–452.

[37] Lawrence Abrams. 2020. Malicious RubyGems packages used in cryptocurrency supply chain attack. https://www.bleepingcomputer.com/news/security/malicious-rubygems-packages-used-in-cryptocurrency-supply-chain-attack/.

[38] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX.. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. 285–298.

[39] Linux. 2020. SecComp Load Filter. https://man7.org/linux/man-pages/man3/seccomp_load.3.html.

[40] 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 Symposium on Operating System Design and Implementation (OSDI)*. 49–64.

[41] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. 2012. Chrome Extensions: Threat Analysis and Countermeasures.. In *Proceedings of the 2012 Annual Network and Distributed System Security Symposium (NDSS)*.

[42] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation.. In *ACM Conference on Computer and Communications Security*. 1607–1619.

[43] Lukas Martini. 2019. Fake version of dateutil and jellyfish. https://github.com/dateutil/dateutil/issues/984.

[44] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation.. In *IEEE Symposium on Security and Privacy*. 143–158.

[45] Michael Sammler. 2018. seccom: Add pkru into seccomp data. https://marc.info/?l=linux-api&m=154039581615478&w=2.

[46] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. 2018. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation.. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses(RAID)*. 359–379.

[47] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 269–284.

[48] George C. Necula. 1997. Proof-Carrying Code.. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 106–119.

[49] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You are what you include: large-scale evaluation of remote javascript inclusions.. In *ACM Conference on Computer and Communications Security*. 736–747.

[50] Nikola Đuza. 2020. 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.

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

[52] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[53] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure in-VM monitoring using hardware virtualization.. In *ACM Conference on Computer and Communications Security*. 477–487.

[54] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen.. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*.

[55] Anthony N. Simon. 2020. Bild: A collection of parallel image processing algorithms in pure Go. https://github.com/anthonynsimon/bild.

[56] Soham Kamani. 2020. Adding a database to a Go web application. https://www.sohamkamani.com/blog/2017/10/18/golang-adding-database-to-web-application/.

[57] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. 2002. Nooks: an architecture for reliable device drivers.. In *ACM SIGOPS European Workshop*. 102–107.

[58] Gorilla Web Toolkit. 2020. mux - Request router and dispatcher. https://github.com/gorilla/mux.

[59] Trend Micro. 2020. The XCSSET Malware: Inserts Malicious Code Into Xcode Projects, Performs UXSS Backdoor Planting in Safari, and Leverages Two Zero-day Exploits. https://documents.trendmicro.com/assets/pdf/XCSSET_Technical_Brief.pdf.

[60] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. 2005. Intel Virtualization Technology. *Computer* 38, 5 (2005), 48–56.

[61] 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).. In *Proceedings of the 28th USENIX Security Symposium*. 1221–1238.

[62] Aliaksandr Valialkin. 2020. FastHTTP: Fast HTTP implementation for Go. https://github.com/valyala/fasthttp.

[63] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with Code-centric memory Domains.. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*. 469–480.

[64] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation.. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. 203–216.

[65] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity.. In *IEEE Symposium on Security and Privacy*. 380–395.

[66] Emmett Witchel, Josh Cates, and Krste Asanovic. 2002. Mondrian memory protection.. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. 304–316.

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

[68] 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 *IEEE Symposium on Security and Privacy*. 79–93.

[69] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study.. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*.

[70] Eric S. Yuan. 2020. Zoom's Use of Facebook's SDK in iOS Client. https://blog.zoom.us/zoom-use-of-facebook-sdk-in-ios-client/.

[71] Zack Bloom. 2020. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/.

[72] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem.. In *Proceedings of the 28th USENIX Security Symposium*. 995–1010.

[73] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces.. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*. 75–88.