Efficient language level isolation

Semester project report

Solène Husseini (Advisor) Mathias Payer
(Advisor) Atri Bhattacharyya

2023-06-09

Efficient language level isolation

Background

The goal of this project is to implement efficient language level isolation. In practice, this means the ability to restrict the permissions of given pieces of code to a subset that the programmer considers safe. This compartmentalization approach to security is already well known and implemented through process isolation, in the Firefox and Chromium browsers for instance. However, the tradeoff is that of higher overhead because of inter-process communication and increased application complexity.

Thankfully, a recent paper introduces Enclosures (Ghosn et al. 2021), a language level mechanism for specifying isolation boundaries in user code. It aspires to be used transparently and allow existing code to be easily isolated, which would address the issue of complexity. Its main implementation relies on virtualization however, which incurs a performance impact that may be unreasonable for highly optimized latency sensitive applications like browsers or web servers.

This second concern could be addressed by SecureCells (Bhattacharyya et al. 2023), a proposal that changes the way permissions are ascribed to memory regions and allows separating access rights to parts of memory within one process, thus avoiding to split the application in processes and pay the cost in IPC.

Thus, to implement language level isolation we choose to reuse both of these papers and explore implementing a SecureCells backend for Enclosures. We first introduce the papers, describe how Enclosures is implemented, and then discuss a possible design and implementation. We conclude by presenting multiple issues that this project encountered, and direct next efforts to alleviate them.

Enclosures

Enclosures introduces a new language level construct and policy for compartmentalizing libraries within a program. This new construct, called an “enclosure” and introduced by the keyword “sandbox”, allows the user to define a closure in which access to memory allocated by other libraries is restrained. Thus, each library allocates memory into its own arena, and all memory restrictions are based upon it. In particular, an enclosure assigns RWX permissions to arenas of libraries directly mentioned inside of it and their transitive dependencies, and no permissions to all other arenas. The programmer may then specify permissions for any arena at their discretion, to allow accessing parameters or closed-over variables. It is also possible to restrict which syscalls can be called within the enclosure.

SecureCells

Secure Cells changes the usual permission model by assigning them to contiguous virtual address ranges (in Linux, VMAs), instead of pages. Each of these VMAs is called a Secure Cell (SC). Additionally, each thread is equipped with (among others) a SID register, which identifies the current Security Division (SD) that it is running as. The permissions of each SC to each SD is described in a special 2D protection table (PTable), each entry containing a bit for Read, Write and eXecute access. Moreover, a set of 8 new instructions manages the permission changes and switches between SDs. By assigning different SDs to compartments, an application can efficiently implement a relatively fine grained security policy.

Implementation of Enclosures in Go

Enclosures are implemented in a modified Go compiler, thus we contextualize here their implementation: how the Go runtime works, how control flows between goroutines and enclosures, and how different backends can be hooked into it.

The Go execution model

All Go code is managed by the runtime. It is a special module that is injected in every Go binary by the compiler. It manages allocation and garbage collection, creation and scheduling of goroutines (green threads), and interfaces with the OS for syscalls. It is a critical component to take into account for any compartmentalization project, since the binary starts executing in the runtime, which then sets up the allocator and the main goroutine. Any code that will execute and all allocated memory will go through the runtime first, thus it appears like an obvious central point in which to manage compartments.

Runtime structure

Execution context The Go runtime defines three structs `g`, `m` and `p`, that hold context related to each goroutine, OS-thread and cpu core, respectively. Since goroutines are "green threads" that are scheduled and migrate independently of the physical OS thread they are running on, the scheduler needs these structs to keep track of the environment.

In particular, each goroutine created by the user stores its context in a `g`. It contains a pointer to its stack, and information used by the scheduler and garbage collector, as well as a pointer to its current `m`. The `m` holds thread-local state, such as a pointer to a signal-handling goroutine, a pointer to its TLS, and info related to scheduling, particularly around blocking syscalls, since they require particular care. It also holds a pointer to the current `p`, which represents a physical cpu core, which contains lock-free data, which is meant to be accessed without locks (since it never moves from the current core), such as the `g` runqueue `runq` and the `mcache` (explained below).

The Enclosures implementation also adds a few fields in `g` and `m`, in particular `sbid`, which holds the id of the sandbox currently running (or 0 if not sandboxed). These fields dictate the security policy to apply to the current goroutine, and are the argument passed to the hooks described below. This sandbox id has the form `"packageid:enclosureid"`, where `packageid` is the Go module in which the enclosure is declared, and `enclosureid` is a unique increasing number that is assigned to each enclosure in a given module to differentiate them.

Transitions Understanding the control flow between different goroutines and the scheduler is crucial to implement a backend correctly.

Go programs start by creating a goroutine (and thus its associated `g`) on which to run the main function. In order to schedule it and other future goroutines, the runtime also creates one scheduler goroutine called `g0` for every `m` (OS-thread). This `g0` allows each core to schedule its own goroutines without having to communicate and synchronize with the other threads on every context switch, thus each `g0` only runs on its associated `m`.

Go follows a cooperative model of concurrent execution: each goroutine is responsible for eventually yielding back into the scheduler.

The first reason to yield would be when communicating with other goroutines, such as locking a mutex, or recieving or sending a message on a channel, since it may block. Instead of blocking the whole `m`, the goroutine yields back to the scheduler, and will get rescheduled when ready.

Another reason is to execute syscalls: instead of blocking all other goroutines that would run on this `m`, the scheduler lets other `m`s steal some of its own to keep running as many goroutines as possible.

A third reason is when a goroutine needs to reallocate its stack. Indeed, the compiler emits checks in the function prelude and at compiler-generated safepoints. They check if the goroutine's stack pointer is smaller than `g.stackguard0`, and if it is, yield into the scheduler to reallocate its stack. This can also be used by the garbage collector to stop all running goroutines when performing collections by setting the field to a large value manually.

However, not every action is run on the `g0`. Many runtime functions called by goroutines, like memory allocation functions, are called on the current goroutine's stack without switching to the `g0`. This has the advantage of improving performance, but complicates our backend's implementation if we want to separate the runtime functions and datastructures from untrusted goroutines, as it would now require a compartment switch to allocate memory for instance.

Some scheduling tasks also bypass the `g0`, for instance a goroutine may have a list of others that depend on it completing an action (i.e. recieving a message from a channel), in which case the goroutine would yield to the one in this list directly without going through `g0`. This also informs the design of a backend, as now not only can context switch from the runtime to arbitrary code and vice versa, but also from arbitrary code to arbitrary code, where both potentially have different privileges.

Arenas The Go compiler uses a memory hierarchy to make memory allocation more efficient. First, it uses a 2-level virtual page table to represent all pages that are possible to allocate, shown as "arenas" in figure X. For x86_64, the L1 only contains one entry and L2 contains 4 millions, each entry (arena) spanning 64MB. The L2 table is mmaped and contains pointers to `mspan`s, that are lazily filled in when the runtime requests memory. Once an arena (thus represented by an mspan) is full, another arena will be mmaped and its `mspan` will be stored in the corresponding entry in the virtual page table.

In order to speedup allocation, the main allocation structure, called `mheap`, contains an array that maps each allocation size to an `mcentral`. An `mcentral` keeps a linked list of `mspan`s which only contain objects of a specific predetermined size, to enable garbage collection and allocation optimizations. To avoid looking up a global and thus potentially contended datastructure every time, each `p` contains an `mcache`, which holds an array of per-size linked lists containing free `mspans`. Once they run out, a few others are fetched from the corresponding `mcentral`, thus amortizing the cost of synchronization across cores.

Enclosures hooks Litterbox exposes multiple hooks that a backend may implement:

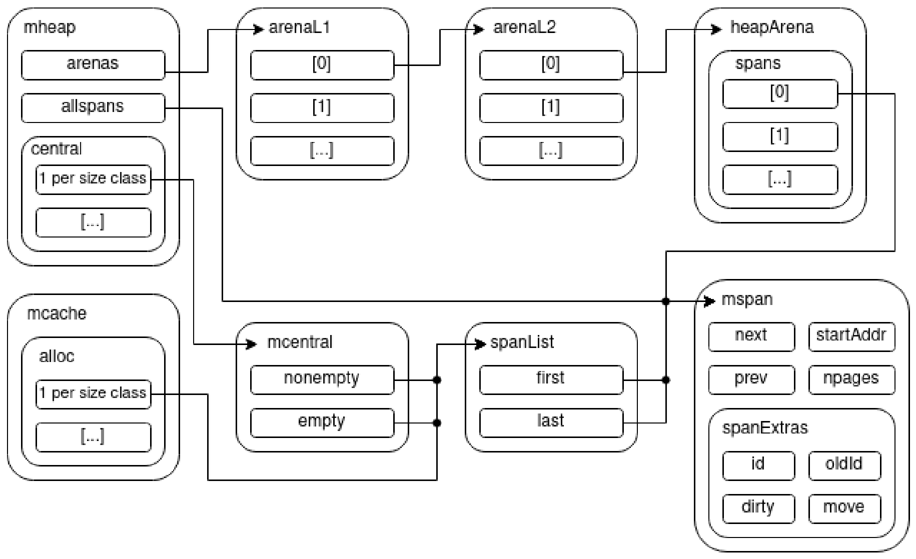

Figure 1: Memory hierarchy diagram

Hook	Where it is called	Purpose
`Init()`	Before registering all other hooks	Setup the environment (i.e: MPK: call pkey_alloc)
`Prolog(SandId)`	Before starting an enclosure	Setup permissions to the one of the enclosure
`Epilog(SandId)`	After exiting an enclosure	Revert the to the permissions of the caller
`Transfer(oldId, newId, start, size)`	When allocating freed memory	Move a span (~= page) from an arena to another
`Register(id, start, size)`	When allocating new memory	Set the permissions for this new fresh span
`Execute(SandId)`	When changing privilege level	Switch privilege level to a different one
`Mstart()`	When creating a new M (OS thread)	Setup per-M data (in thread local storage for instance)
`RuntimeGrowth(isHeap, id, start, size)`	When allocating memory from the OS	Register new pages to the backend (i.e. after mmap)

where SandId is a sandbox id, and oldId, newId and id are package ids. `Mstart`, although implemented, seems to be never called in the code. I do not know if this is intentional or not.

Note that although these nice hooks exists, many places in the runtime are still modified to dynamically check under which backend they are running, making the implementation much more difficult to understand properly since the functionality of a single backend isn't actually contained inside these hooks (isMPK, isVTX).

Example of a translation from sandbox to normal Go:

```
foo := sandbox[](bar int) int {
    return dothings()
}
res := foo()

Becomes:

foo := func(bar int) int {
    builtin.Prolog()
    defer builtin.Epilog()
    return dothings()
}
res := foo()
```

Control flow over time

To illustrate the places where these hooks are called and which permissions are what, let's examine a simple example.

```
func main() {
    // SIM is a backend that doesn't do anything.
    // I added prints to show what's going on.
    gosb.Initalize(backend.SIM_BACKEND)

    call1 := sandbox["main:RWX", ""]() {
        fmt.Println("inside start")
        // Yield to scheduler.
        runtime.Gosched()
        fmt.Println("inside end")
    }

    fmt.Println("outer start")
    call1()
    fmt.Println("outer end")
}
```

```
>init

outer start
```

```
>prolog : newid= 3:0
stats> isg: false gid:  :mid:  :g0id:
stats> isg: false gid: 3:0 :mid: 3:0 :g0id: 3:0

inside start

>execute : newid= 3:0
stats> isg: true gid: 3:0 :mid: 3:0 :g0id: 3:0
stats> isg: true gid: 3:0 :mid: 3:0 :g0id: 3:0

inside end

>epilog : newid= 3:0
stats> isg: false gid: 3:0 :mid: 3:0 :g0id: 3:0
stats> isg: false gid:  :mid:  :g0id:

outer end
```

The output of the program shows which hooks are called and with which parameters. A line with `>` indicates which hook is called as well as their argument. The `stats>` lines are printed before and after the context switch, and display the following info: - isg: is the hook called on the main scheduling goroutine (`g0`) of the `m`? - gid: sandbox id of the current `g`. Empty string means root (it is unclear why “” was chosen instead of a default recognizable value like “0:0”). - mid: sandbox id of the current `m`. - g0id: sandbox id of the current `g0`.

When entering the enclosure, the `prolog` hook is called with the destination sandbox id as argument, and we observe the change from `""` (root) to `"3:0"` (package 3 enclosure 0). Then, when we call the `runtime.Gosched()` function to forcefully go back into the scheduler, inexplicably no hooks are called. Then, when our goroutine is rescheduled, the `execute` hook is called to go back into the enclosure. Finally, the epilog hook is called when exiting the enclosure and returning to `main`.

This example exhibits a strange behavior: no hooks are called when calling into the runtime. Yet, we are certain that this `execute` call is coming from the runtime to our enclosure and not the other way around, since the argument passed to it is the enclosure's sandbox id and not `""`, and the call is done on the scheduler goroutine `g0`, not on the enclosure's. We can also notice that the ids are not updated when inside the scheduler goroutine since the hook wasn't called. This necessarily means that either the implementation lets any enclosure access the scheduler which contains critical data that malicious code could easily exploit, or there is a special mechanism in the scheduler that context-switches without using hooks.

Backends

Not only does Enclosures specify a compartmentalization policy, it also introduces 2 backends that can plug into generic hooks to implement and enforce it.

Intel MPK The first is using Intel MPK (Memory Protection Keys), a hardware extension that allows tagging each page with a 4-bit key and adding a 32-bit PKRU per-thread register that specifies the allowed accesses of this thread with 2 bits per key, Access Disable and Write Disable. During every access, the Disable bits of the registers corresponding to the key of the page being accessed are checked, and if they conflict with the access (access or write), they raise a SIGSEGV. Specifically, in Enclosures, each library's arena is tagged with its own key, and each entry and exit into an enclosure writes the PKRU register to change the access permissions. Evidently, this does not protect against a malicious library or against remote code execution (through a vulnerability for example), since the containerized code can write the PKRU register and disable all MPK memory protections. Thus, while being fast, MPK acts more as a sanitizer than a real containerization mechanism.

VT-X The second backend uses VT-X to run the whole program through a hypervisor. The memory protection is accomplished by modifying the permissions on the page table on each entry and exit of an enclosure, which incurs some overhead. The virtualization can also restrict syscalls, a feature which the other backends do not provide. Overall, VT-X is the most featureful and effective backend, but it incurs non-negligible overhead.

Subtle differences The MPK backend only works on select (mostly server) processors, and thus cannot run on most machines, making its deployment less easy, while VT-X is supported on most x86_64 hardware. A more critical difference is that VT-X disallows calling an enclosure from within another one, while MPK does. This changes quite significantly the possible designs for the SecureCells backend, since following how VT-X is implemented restricts the compartment changes to a single nesting depth, from full privilege to sandbox and back, but not from sandbox to another sandbox. Since this difference doesn't appear to be documented and was discovered late into the project, I assumed the MPK model to be the reference, since this is the code I looked at the most extensively by virtue of being much simpler than VT-X.

Storage of package data embedded in the binary

The Enclosures compiler stores data about package ids and which code and data correspond to which enclosure in a json file embedded in one of the `.sandboxes` and `.bloated` sections of the binary. This data is then dynamically read at application startup to setup internal datastructures in the runtime, before calling the `Init` hook.

Design of the backend implementation

Mapping Enclosures policy to Security Divisions and Cells

Let's examine an example to illustrate a simple but naïve mapping from Enclosures to SecureCells. This is the same example program as in the Enclosures paper.

```
package main

import (
    "img",
    "libFx",
    "os",
    "secrets"
)

var key := "myPrivateKey"

func main () {
    original := secrets.LoadImage("myimg.png")
    rcl := sandbox["secrets:R","none"]func(o *img.Png) *img.Png {
        return libFx.Invert (o)
    }
    inverted := rcl(original)
    res := img.Watermark(inverted, key)
    os.Write("signed.png", res.Bytes())
}
```

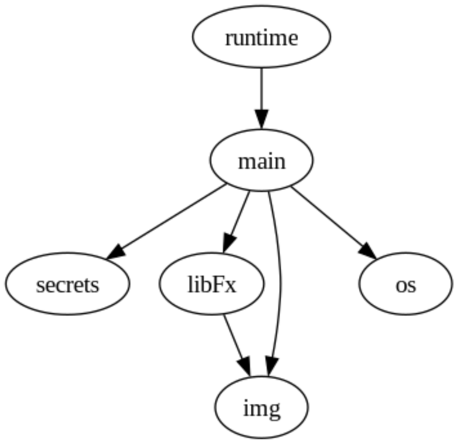

Figure 2: Dependency graph

In the modified Go compiler `gosb`, the package and enclosure ids could be assigned as such:

module	package id
runtime	0
main	1
secrets	2
libFx	3
img	4
os	5

And the enclosure assigned enclosure id 0, since it is the only one in package main. Thus its sandbox id would be `"1:0"`. Since each enclosure maps to a single unique sandbox id and since the number of enclosures is known at compile time, a straightforward mapping of enclosure to security division could look like this:

SD \ SC	runtime	main	secrets	libFx	img	os
1:0	–	–	r-	rw	rw	–
out of sandbox	rw	rw	rw	rw	rw	rw

The sandbox id gets assigned a security division that only has access to the transitive dependencies of the enclosure, here `libFx` and `img`, plus read access to `secrets` as specified directly by the programmer.

Issues with the naïve mapping

Shared use with the runtime The simple mapping presented above encounters unclear situations when put into the context of the Go runtime.

First, the garbage collector will periodically scan the stack to find root pointers, thus all gc threads need to run in a SD that has at least read access to all SCs.

Second, every thread periodically needs to check data structures from the `runtime` module. For instance, as explained above, a special field in `g` called `stackguard0` holds a pointer to the stack which is checked on function entry (and at special compiler-generated safepoints). If the stack pointer is beyond this stack guard, the thread calls into the runtime to reallocate the stack; it is also used by the runtime to gracefully pause a goroutine at will. This means that a thread should have access to its own `g`. However, a `g` contains a lot of data that is critical for the runtime and gc, thus letting a sandbox access it would compromise its compartmentalization. One way to circumvent this would be to allocate a separate SC for the fields that need to be accessed from both the sandbox and the runtime, but this is not simple to implement: since an arbitrary number of goroutines can run concurrently, additional memory allocation logic must be added to manage these special shared cells. Furthermore, calls to runtime functions are sometimes done directly on the goroutine's stack as an optimization, thus these calls would need to be transformed to context switches.

Passing and returning values The current Enclosures implementation desugars to a function, as shown above, thus it allows passing and returning values. Since in the Go abi the caller allocates stack slots for arguments (even if they fit in registers) and return values, the stack would need to be shared between function calls. Perhaps surprisingly, this would also be the case for arguments passed or returned by value from a language point of view (with a type that isn't a pointer type `*`), since they are still passed on the stack.

The main issue with sharing stacks is that any sandbox could read or write to data that happened to be on the caller's stack frame, breaking confidentiality and integrity. It also makes it trivial to ROP, by overwriting a return address on the stack and redirect code execution. This is especially potent when the caller of the enclosure has rwx privileges on the malicious library, as it can then directly jump back into its own code as the privileged enclosure.

However, the alternative is to allocate a new stack in a separate cell when entering an enclosure, which incurs the same overheads as described above.

Compromising on the security of the stack The Enclosures paper addresses the concerns about stack sharing by ensuring that every call to an enclosure would split the stack and thus allocate a new stack, separate and containerized from the caller's. However, I could not figure out if and where that was implemented in the modified Go compiler, nor how argument passing was handled in that case. I also checked the disassembly but nothing suggested that the stack was reallocated. These concerns are also invalidated by our need for a more recent Go version (for RISC-V support), which doesn't use split stacks anymore, and directly reallocates stacks to be bigger when needed.

In light of the above issues, we consider the stack to be trusted and do not allocate different stacks on compartment switch. This does reduce compartmentalization, especially since SecureCells has been designed to work with stack switching using trampolines, but Enclosures's current backends like VT-X and MPK do not protect the stack either, so we can still get a comparable implementation with SecureCells.

Alternatives There are multiple ways to protect the stack. We discuss them here although none were implemented as they are out of scope of this project.

The first is to forward all the arguments passed to the enclosure to the `Prolog` hook, and let the backends execute the body of the function using a custom abi that would pass all the required arguments in registers. Then, the `Epilog` hook would take care of transferring back the return values. Since the hooks are run as the `runtime` SD, they have rw access to both stacks. This also simplifies the allocation of the new stacks, as they now only need to be accessible in the target, and thus it avoids creating cells for each pair of caller/callee: they can be allocated in the same way as any memory in the target SD, and this allocation could also be directly handled by the `Prolog` and `Epilog` hooks.

The second is to make enclosures goroutines instead of functions. They already have their own stack and mechanism for passing arguments, and clear entry points for hooks when they are scheduled and descheduled. Furthermore, transferring data between enclosures (and thus SCs) can be handled transparently by channels, copying data from one SC to another without special care. Although out of scope of the SecureCells backend, they also have a mechanism to handle syscalls (though not prevent them), which could simplify further efforts to enforce syscall restriction. The start time of a goroutine is of course higher than a simple function call, but they would fit well in the use case outlined by the SecureCells paper where a few long-lived goroutines would process packets one after the other, using channels to transmit the data.

Using Arenas as Secure Cells

Enclosures separates allocation requests per-package, thus `mspan`s have an extra field `spanExtras.id` that tracks their package id. However, SecureCells works at a VMA granularity, thus it is not scalable to add a new SC per span, as it would defeat the purpose of SecureCells anyways. Therefore, a solution would be to track package ids on an arena (see above) granularity instead. Each arena is 64MB large, so the number of SCs would remain reasonable. The arena size is also configurable easily in the compiler, so it can be reduced if needed.

Changing to an arena brings a host of new challenges to overcome however. In the first place, datastructures like `mcentral` and `mcache`, which are performance critical, track `mspan`s regardless of which arenas they are stored in. Two approaches could be used: keep the current list of `mspan`s but ignore the ones with the wrong package id, which would significantly slow down memory allocation, or keep a list per package id, which is much more memory intensive but should minimally hurt performance since it only requires an additional array index to get a page.

Any strategy described above still has the shortcoming of requiring an SD switch on memory allocation, since the `p`, `mcache` and `mspan` datastructures are themselves allocated by the runtime and should not be accessible from within the enclosure.

Mapping Enclosures hooks to SecureCells instructions

Assuming the tradeoffs described above, we can map most Enclosures hooks straightforwardly to SecureCells instructions:

Hook	SecureCells action
`Init`	Initialize the PTable with the package information embedded in the binary.
`Prolog`	Call SDSwitch to enter the target SD.
`Epilog`	Call SDSwitch to go back to the source SD.
`Transfer`	This corresponds to SCReval or SCTransfer depending on the implementation. Discussed below because of complications.
`Register`	Similar to `Transfer`.
`Execute`	Call SDSwitch to enter the target SD (same as prolog).
`Mstart`	Not implemented so not used.
`RuntimeGrowth`	Modify the permissions in the PTable for the newly allocated VMA.

The `Transfer` and `Register` hook are special because they uncover another aspect of incompatibility between Seccells and Enclosures. They are called when an allocation needs to be reused after being garbage collected and allocated again. Indeed, the sandbox in which it was freed and the one in which it was allocated may be different in Enclosures, thus this hook was meant to let the backend change the permission of the allocation. However, Seccells works with permissions at a VMA level and thus this change of allocation is impossible to perform efficiently. I believe this issue can only be addressed by deeply modifying the allocator to separate allocation between each SC and thus avoiding need to change the permission of allocated pages.

Even if the permissions were tracked per-page, another issue arises from SecureCell's cooperative permission transfer. All instructions that transfer permissions take effect only if the target SD accepts them with `SCRecv`. This means that in the `Transfer` hook, not only do we need to transfer the permissions from one SD to another, but also both need to send and accept the change. There does not seem to be a straightforward way to fix this without compromising on security: since the source SD doesn't know which allocations it can let go or not (only the garbage collector knows), an attempt to design a simple function that calls `SCTfer` on an arbitrary pointer could allow any malicious compartment to get leak confidential data or cause denial of service.

The alternative would be to use a sequence of `SCInval` when garbage collecting followed by `SCReval` during allocation. However, `SCInval` requires that no other SD has access to the target SC. Unfortunately, the garbage collector does not and cannot run in the same SD as the ones executing user code, thus the This leads back to the same issue as in the previous paragraph.

The big takeaway of this attempt is that hierarchical systems where one compartment is intended to have more privileges than others, and manage memory and scheduling in their place, is difficult to reconcile with SecureCell's permission transfer model, which brings up the question of whether this is a problem unique to Go, an issue that applies more generally to languages driven by a runtime / garbage collector, or an artifact of this project's approach.

Implementation of the backend

Now that we have established the requirements and tradeoffs of using SecureCells as a backend for Enclosures, we discuss the obstacles encountered, in particular the issue of porting Go to RISC-V, and the compromises we have settled for, such as creating a library to emulate the SecureCells instructions on x86_64.

Issues with the Go compiler

Ideally, using SecureCells as a backend for Enclosures should be as simple as implemented the provided hooks; but many obstacles kept appearing.

The first issue was that `gosb`, the Go compiler with Enclosures, was based off of a random commit around the Go 1.13 release. Unfortunately, this release had no support for targeting the RISC-V architecture, as the first one that did was Go 1.14. However, we could not simply rebase or advance the compiler to Go 1.14 for a few reasons. First, since the commit of the fork was between releases, there was no straightforward way of assessing the exact features supported by the compiler; second, cherry-picking commits until Go 1.14 would mean fixing conflicts with around 950 commits, which did not seem feasible, especially because Enclosures modifies critical parts of the runtime which have also been concurrently overhauled for Go 1.14.

The second issue is that although SecureCells provides modified versions of `qemu` and `linux`, they are understandably unstable and do not work well with Go executables, since they use many syscalls and make assumptions about the internal behavior of said syscalls, such as relying on being able to mmap some specific virtual addresses. We tested Go executables produced by the latest Go version 1.20.4 to make sure the RISC-V support had matured enough. Combined with other stability issues, we couldn't get Go executables to work on SecureCells' `qemu`.

LibSecureCells

In light of these issues, I decided to emulate the SecureCells environment in software using a library that could be a drop-in replacement to the SecureCells instructions, to try and see how far we could mitigate the issue of being unable to run Go on SecureCells' `qemu`.

Emulation The goal of libSecureCells is to provide an interface which can easily be used in place of the hardware instructions, but without providing the kind of actual memory protection that SecureCells provides, since that would be prohibitively expensive. It is written in 700 lines of C and meant to be compiled for and ran on x86_64; its development is not complete because integrating it into Enclosures proved to be fruitless, but it may help lay a foundation for future efforts at emulating SecureCells.

The library has a corresponding function to each SecureCells instruction, as well as a global struct standing in for the normally hardware based PTable. Each of the function checks and stores the permissions in the PTable, returning an error if an invalid operation is performed, such as trying to switch to or give permissions to a non existent SD. Since libSecureCells is a fully userspace program, additional functions are provided, such as `RegisterVMA` and `UnregisterVMA` which ought to be called after / before every `mmap` / `munmap` to register the new VMA in the PTable.

The table is global but not protected by a mutex, thus may not be used from multiple threads. This is not a fundamental limitation, but simply out of scope of the project.

Sanitizer mode A very quick attempt has also been made to provide a sanitizer mode, which calls `mprotect` on every SC on every `SDSwitch` to effectively imitate the memory protection behavior of SecureCells. It is very slow, and does not protect every SC (code sections and dynamic libraries in particular are hard to manage properly without compiler and linker support), but can be used to catch accesses to any mmaps that contains general data and not code or the stack. Ideally this mode to be correct but incomplete, that is it doesn't catch every invalid access but it also doesn't segfault on any correct access. In order to simplify the registration of VMAs at startup, the function `LoadVMAsFromProc` parses `/proc/self/maps` and tries to copy the correct permissions in the PTable. All SCs extracted by this function will not be `mprotect`ed later to ensure that libSecureCells does not inadvertently unmap its own code section while iterating over every SCs to change their permissions in `SDSwitch`.

Switching stacks An attempt has also been made to provide an easy to use interface to switch stacks during an `SDSwitch`. A global array `rsps` allows SDs to store and recover their `rsp` when performing an `SDSwitch`. A hand written assembly function zeroes out registers and jumps to target address while saving the old and loading the new `rsp` from `rsps`. We also provide a function called `MapStack` which simplifies the process of creating stacks for new SDs.

Integration into Enclosures I did not attempt to integrate libSecureCells into Enclosures. Indeed, as discussed before, we could not settle on a satisfying design fitting the current implementation of the compiler, because of all the issues we encountered along the way. However, we do provided an example program that showcases a very basic use of the library.

Issues

In order to achieve our initial goal, we needed a few crucial parts to work to make progress. However, many things didn't work as expected, and the lack of some supporting pieces also contributed to preventing this project from succeeding.

Fundamental issues

The main roadblock of this project was to get both papers' implementations working together. Enclosure was constrained to an old Go version that couldn't target RISC-V, while SecureCells could only run RISC-V binaries. Compounding the issue, SecureCells was also unable to run executables from a modern unmodified Go compiler. Without a way to run a program using both, there was no way to test or even use the programs, rendering the whole project useless.

However, even if it did work, another more fundamental issue arose: the Go runtime and the Enclosure implementation do not map well to SecureCell's security model.

- SecureCells enforces permissions per-VMA, thus we cannot reuse the existing memory allocator since it manages memory at `mspan` (~= page) granularity.
- Goroutines load fields from runtime datastructures on every function call and execute runtime code for memory allocation on their own stack for performance, but these would require a context switch and thus a very high performance hit to isolate properly with SecureCells.
- Memory allocation is managed centrally by the garbage collector, which doesn't fit with SecureCell's collaborative permission transfer model.
- Modern Go versions forgo the split-stack approach and realloc the whole stack instead, forcing enclosures to allocate a new stack when called, which compounds the performance impact.

Secondary issues

In addition to these issues, a few more implementation problems became apparent throughout the project. These are not issues that would prevent the whole project from working, but did present hurdles to overcome.

- Although Enclosures provides hooks to implement other backends, some hooks like `Mstart` are never called in the code, while others are not called in some circumstances like `Execute` not being called when switching into the scheduler (where only runtime code is ran).
- The runtimes checks in multiple places some variables called `isVTX` and `isMPK`, breaking the hooks interface since they modify the behavior of the runtime directly.
- MPK and VTX both seem to have an implementation issue where VTX will crash when calling an enclosure from within another one, while MPK gives full permissions to the current thread when returning from an enclo-

sure, letting malicious code use a nested enclosure call to escalate privileges. Since I initially looked at the MPK backend, I assumed that nested enclosures were supported, but I am not sure if that is actually the case given that it does not work with VTX.

What could work instead

In light of the above issues, it seems inefficient to put more effort in trying to port Enclosures to SecureCells. This doesn't mean that our original goal, efficient language level isolation, cannot be achieved, but rather that the particular means we tried to use to implement it are flawed.

An alternative design for SecureCells would be to compromise on Enclosure's ability to be seamlessly integrated with existing code, by creating a language or an interface in an already existing language specifically designed to enable language level isolation. One such isolation-aware design could circumvent the issues of allocating memory by integrating tightly with the per-VMA permission model, and structure its execution around self-contained long-living blocks to reduce the overheads of creating SCs and stacks.

Paradoxically, stackful symmetric coroutines seem to be a perfect fit: they have their own stack, can yield to other coroutines directly, mirroring `SDSwitch`'s interface, and are designed to be resumable and thus long-lived. Go's goroutine design could have been a decent fit if it wasn't for all the concerns about allocation, scheduling and access to sensitive runtime datastructure.

Conclusion

This project did not manage to enable efficient language level isolation using Enclosures and SecureCells, but it did provide the opportunity to get insights into their requirements and concrete challenges. The problems encountered along the way can inform the design of future language level constructs that aim to provide isolation, while libSecureCells can help inform the feasibility of these constructs without having to rely on a hardware and kernel implementation of SecureCells.

References

Bhattacharyya, Atri, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andrés Sánchez Marín, Babak Falsafi, and Mathias Payer. 2023. "SecureCells: A Secure Compartmentalized Architecture." In *44th IEEE Symposium on Security and Privacy.*

Ghosn, Adrien, 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*, 255–67.