Gábor Pituk

CHERI: Sandboxing libraries in the run-time linker

Part II, Computer Science Tripos

Churchill College

2021–2022

Declaration of originality

I, Gábor Pituk of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I am content for my dissertation to be made available to the students and staff of the University.

Signed *Gábor Pituk*

Date *13 May 2022*

Acknowledgements

I would like to thank

- my supervisor, **Prof Robert N. M. Watson**, for his insights and guidance throughout the project,
- **Dapeng Gao** for our valuable discussions and his helpful advice,
- and Patrícia Stark, friends, and family for their support.

Proforma

Candidate number:	2366A
Title:	CHERI: Sandboxing libraries in the run-time linker
Examination and year:	Part II, Computer Science Tripos, 2021–2022
Word count:	11,973[†]
Lines of code:	2,628[‡]
Project originator:	Prof Robert N. M. Watson
Project supervisor:	Prof Robert N. M. Watson

Original aims of the project

The project aimed to explore the novel idea of sandboxing a shared library via the run-time linker setting it up in a separate protection domain through the use of CHERI capabilities. The core deliverable was a prototype of this approach supporting no-dependency libraries.

I intended to implement this prototype and evaluate its security guarantees, its effects on vulnerabilities, and the incurred performance overheads measured on an appropriate corpus of real-world libraries.

Summary of work completed

Despite being a novel approach, the project was successful. I implemented the basic prototype and extended it in several ways.

The evaluation performed shows that privileges are effectively reduced, code changes to existing software are minimal, and the performance overhead on applications is very low, making this a promising approach to practical compartmentalisation of existing software.

This is the first piece of work to demonstrate sandboxing on Arm's Morello experimental CHERI hardware.

Special difficulties

None.

[†]URL: `https://app.uio.no/ifi/texcount` (accessed 25-April-2022)
[‡]My contributions to the codebases, computed over `git` diffs.
URL: `https://github.com/AlDanial/cloc` (accessed 1-May-2022)

Contents

Chapter 1: Introduction

This dissertation explores the idea of sandboxing libraries in the CHERI run-time linker. CHERI, described in more detail in § 2.1.1, extends processor Instruction-Set Architectures (ISAs) with hardware capabilities that can be used to control privileges of running code, making it possible to create a sandbox within the process address space. Current sandboxing mechanisms incur significant performance overheads, therefore there is a desire for new approaches to compartmentalisation that reduce overheads. A novel approach using CHERI is presented, where the basis of sandboxing is *shared libraries* as opposed to processes, providing strong security guarantees with *very minor overheads* and minimal source code disruption to existing software, given that the libraries have few dependencies.

I demonstrated this by implementing and evaluating this approach on a software stack consisting of historically unsafe data processing libraries. The performance results show negligible end-user overheads, making this a promising approach to practical software compartmentalisation. This work is the first to demonstrate software compartmentalisation on Arm's Morello hardware running a CHERI architecture.

1.1 Motivation

Sandboxing refers to executing software in an environment with reduced privileges, to mitigate the effect of malicious code execution. We might need to assume code to be malicious because we do not trust its source, e.g., when executing JavaScript received from an untrusted web server, or because an attacker might exploit software vulnerabilities stemming from *bugs* or back doors in the code, and attempt to make the system execute malicious code. By limiting the privileges of the code being executed we can mitigate the consequences of a successful attacker achieving arbitrary code execution capability. This concept is called the *principle of least privilege* [21], which is employed in most modern user-facing systems, e.g., operating systems.

1.2 Previous work

1.2.1 Sandboxing mechanisms

There are numerous sandboxing mechanisms allowing users to run untrusted code. Most of these, such as SELinux [17] or macOS's App Sandbox [2, 24] are coarse-grained sandboxing mechanisms based on processes, focussing on access control, and incur significant overheads. Others, such as Java's class loading mechanism and security policy – described in more detail in § 2.1.5 – provide intra-address-space mechanisms, but cannot be applied to C/C++ code, which makes up most of the traditionally extremely vulnerable programs and libraries. CHERI promises scalable fine-grained compartmentalisation of such software.

1.2.2 Previous work within CHERI

The need for compartmentalisation in CHERI

CHERI's memory safety guarantees mitigate many memory-safety vulnerabilities, or transform more serious vulnerabilities – e.g., arbitrary code execution through stack or heap buffer overflow – into crashes, as shown in [25]. An analysis performed by Microsoft Security Research Center [14] has shown that at least 67% of the vulnerabilities reported to them in 2019 are deterministically mitigated by CHERI; however, at least 18% are *not* mitigated. In particular, they demonstrated that some artificially injected vulnerabilities in the QtWebKit JavaScript Core can be exploited to achieve arbitrary command execution. Arbitrary code execution vulnerabilities still exist on CHERI – especially in software intended to allow code execution, such as Just-in-Time (JIT) compilers, where fine-grained memory control is difficult to achieve – or back doors might be present in software (supply-chain attack), justifying the need for software compartmentalisation.

The hope is that the intra-address-space separation of domains provided by CHERI capabilities can be used for finer-grained and lower-overhead compartmentalisation of software than current MMU-based solutions. CHERI enabled software by default provides some isolation guarantees because a memory access can only be performed if a capability authorising it is reachable from the register file. However, the set of reachable capabilities is often unnecessarily large, because domains are not fine-grained enough by default, or capabilities leak across domains, e.g., through temporary registers or the stack. Therefore, there have been efforts to try to provide more security guarantees in a scalable way.

`libcheri`

In 2005, `libcheri` [28] was an object-oriented approach to implementing mutually distrusting compartments interacting with each other. These compartments encapsulated code and data, and exposed an API callable from other objects, much like objects in object-oriented programming languages. One of the main limiting factors of this approach was the amount of source-code disruption it led to when trying to compartmentalise existing software.

The approach taken by this dissertation, putting individual shared libraries into sandboxes, has the potential to implement similar security guarantees, with minimal source code disruption.

1.3 Collaboration

PhD student Dapeng Gao's ongoing research overlaps with this project, and I reused two components from his work in my implementation: a thread-safe implementation of extra per-thread stacks, and a mechanism to replicate a trampoline, allocating pages on demand, totalling 209 lines of code (not counted in the 3,636 lines of code reported in the Proforma). These are explicitly referenced in the relevant parts of chapter 3 (§§ 3.3.1

and 3.3.4). The contents of the domain transition code barring the trusted stack implementation (§§ 3.3.2 and 3.3.3), and the entirety of the overall design of sandboxing (§ 3.1), the dependency policy (§ 3.2), and the evaluation performed (chapter 4) remain my contribution. In fact, this project is the first to have performed any evaluation of this novel approach.

Chapter 2: Preparation

This chapter provides relevant background on run-time linking (§ 2.1.2), CHERI (§ 2.1.1) and its hardware and software stack (§§ 2.1.3 and 2.1.4), and presents the requirements and the engineering approach (§§ 2.3 to 2.5).

2.1 Background

2.1.1 CHERI

Capability Hardware Enhanced RISC Instructions (CHERI) [26] is a research project at the University of Cambridge and SRI International extending current instruction sets with architectural capabilities, allowing fine-grained memory protection and software compartmentalisation. Capabilities are a new composite hardware data type representing pointers with encoded additional metadata, which can be stored in registers and memory. Instead of being able to dereference any machine word storing an address with conventional load and store instructions, in CHERI, these instructions accept capabilities, and they only succeed if the capability authorises that operation.

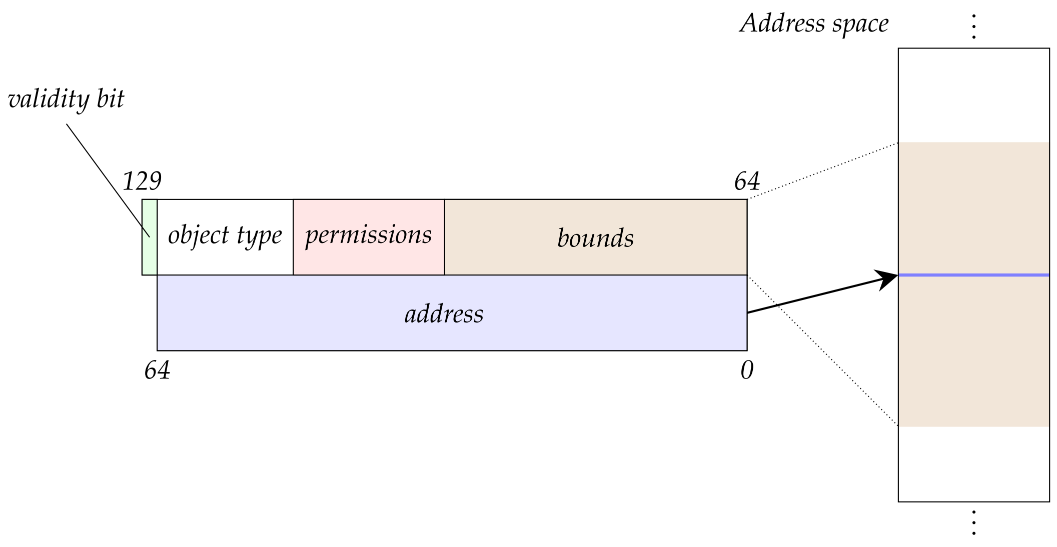


Figure 2.1: *Capabilities in CHERI*

Capabilities contain the following fields:

- *bounds*, i.e., base address and length, specifying the range of memory addresses for which it authorises operations,
- *address* that the capability points to; has to be within bounds for memory operations to succeed,
- *permissions*, specifying what operations the capability authorises, e.g., read, write, instruction fetch, etc.

- ▹ *validity tag*; memory operations only succeed on valid capabilities,
- ▹ and some additional flags described later.

In CHERI, data pointers, as well as code pointers, are represented as capabilities. The program counter (PC) is replaced with the program counter capability (PCC), which has to authorise instruction fetch operations.

Capabilities are unforgeable tokens of authority. The tag tracks *provenance validity*: any valid capabilities must be derived from other valid capabilities via valid transformations. These transformations are *monotonic*, i.e., the derived capability cannot have more privileges than the source capability, both in terms of bounds and permissions. Therefore, throughout execution – between transitions of the thread of control – the set of reachable capabilities cannot increase.

To implement transitions of the thread of control, CHERI introduces some operations providing *controlled non-monotonicity*. Capabilities can be *sealed*, meaning that they cannot be modified or dereferenced until unsealed by special operations such as "unseal and branch". Controlled non-monotonicity can implement domain transitions by jumping to a trusted entry point in the target domain that might have more privileges than the source domain. Other architecture-specific features are provided to implement controlled non-monotonicity.

CHERI instruction sets have been formally verified in [19] to have these security properties such as reachable capability monotonicity.

2.1.2 Run-time linking overview

This section introduces the *non-CHERI-specific* concepts of run-time linking; remarks about changes for CHERI can be found in § 2.1.3.

Software libraries provide a collection of functionality, often used by many different programs. In the world of C, libraries can be built individually, and programs can use pre-built libraries without the need to compile them. This leads to a customisable, efficient multi-stage build process:

- ▹ First, the compiler processes each compilation unit, producing a binary object file that might contain references to undefined symbols.
- ▹ Then, a program called the *static linker* combines these object files and the library files – potentially in multiple passes – into a final executable.

Static libraries are conceptually not much different from (a collection of) object files, the static linker puts the static library's code and data into the executable.

Dynamically linked libraries – also called *shared objects* – are however not statically linked into the final executable, but are mapped into the process's address space at load or run time by a program called the *run-time linker*. The static linker can put annotations in the final executable identifying the undefined symbols, and specifying which libraries need to be searched by the run-time linker. The mechanism is described in more detail in § 2.1.3.

There are many advantages of shared objects. They might reduce the size of binaries significantly. For example, a "hello world" program on my machine linked against the `libc` shared object is 14KB, compared to 3.6MB when linked against the static `libc` library. It also allows switching to a newer version or another implementation of the library without having to recompile the programs using it, we just have to replace the shared object in the run-time linker's search path. Read-only segments of the dynamically linked libraries can also be shared between many processes, assuming the OS primitive used supports it (e.g., `mmap()`), avoiding unnecessary duplication in memory. On the other hand, some overhead is incurred at load or run time because the symbols need to be resolved.

GOT

The Global Offset Table (GOT) is an indirection table storing absolute memory addresses of symbols. It allows a caller to reference code which can be loaded into any memory location (e.g., a shared library) without performing relocations for each reference site: instead of relocating every reference site, the run-time linker only needs to resolve each symbol and install its absolute address into the GOT, i.e., perform relocations per symbol referenced.

Executables and shared libraries have a GOT section mapped right after their code sections. References to other shared libraries look up the code pointer in the GOT entry of the function using PC-relative load with the known static offset and then dereference it.

By not modifying the code segments of objects via relocations, they can be shared among multiple processes through shared pages, only the GOT needs to be replicated for each process.

There is a trade-off in load-time overhead and access performance: indirect accesses through the GOT require an extra instruction and pollute the cache, but per-site relocations can be very costly at load time, e.g., replacing the holes with absolute addresses in each `libc` reference in a large program.

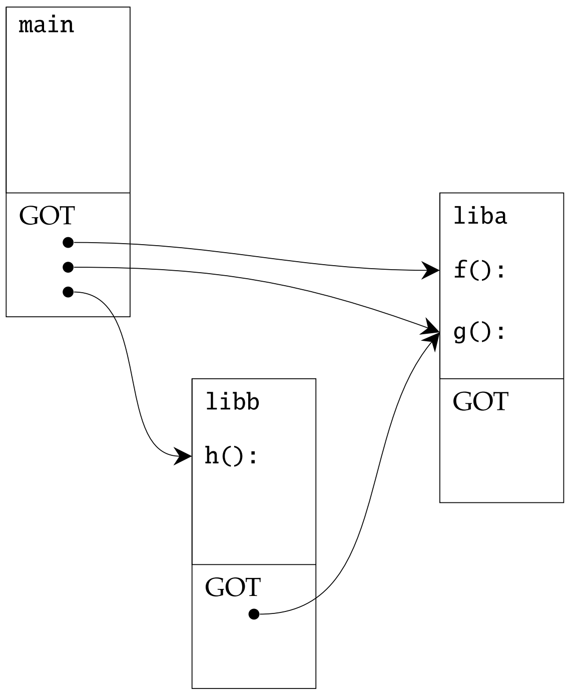


Figure 2.2: *GOTs referencing shared library functions*

Lazy binding & PLT

On program start-up the run-time linker opens all shared objects needed by the application, mapping them into process memory. If the `LD_BIND_NOW` environment variable is set, it eagerly binds all symbols, i.e., finds the definitions of all external symbols and fills in the corresponding GOT entries with the resolved pointers. *Lazy binding* is an optimisation to reduce the load time of objects, which is the default option these days. It defers binding external function symbols from load time to run time, triggering symbol resolution when the function is first called.

To do this, it initialises the GOT function entries to somehow call the run-time linker's symbol binding function which looks up the function definition, installs it into the caller's GOT, and jumps to the resolved function. For the symbol binding code to know which symbol needs to be bound, it needs to determine the caller's object and the index of the GOT entry that has been called. The caller's object can simply be found by looking at the return address, and the symbol's index can be determined either from the address of the GOT entry kept around in a register after loading it for the indirect jump (Arm's approach), or by initialising the GOT entry to a stub function which loads the index into a register or pushes it onto the stack before jumping to the symbol binding code (the x86 approach).

Most systems use an additional layer of indirection: the *Procedure Linkage Table* (PLT). It contains stubs for each external function, loading the GOT entry and jumping to it, which can be seen in listings 2.1 and 2.2. In the case of x86, it also contains an additional symbol binding stub to push the function's index onto the stack and jump to the symbol binding code. As opposed to loading and jumping to the GOT entry of a function `f` directly in the code, the stub `f@plt` is called just like if it was an internal function. This is because of the multi-stage compilation process: when producing object files the compiler would need to know which functions are going to be external or internal and emit different code accordingly, but this distinction is made only later in the linking stage. With the use of the PLT, the static linker only needs to change references to `f` into references to `f@plt` without having to modify instructions themselves (i.e., only change the relocations).

```
/* Load PC-relative page
 * address of the GOT entry */
adrp x16, page<.got.plt+index>
/* Add page offset */
add  x16, x16,
   pageOffset<.plt.got+index>
/* Load GOT entry */
ldr  x17, [x16]
/* Jump to GOT entry
 * keeping its address in x16 */
br   x17
```

Listing 2.1: *PLT stub on Arm*

```
/* Jump indirect to the GOT entry
 * using memory addressing */
jmpq  *<.got.plt+index>(%rip)
/* Lazy binding stub:
 * push index onto stack
 * as argument for binding code */
pushq index
/* Jump to binding code through
 * a stub in the PLT header */
jmpq  .plt
```

Listing 2.2: *PLT stub on x86*

2.1.3 CheriBSD

CheriBSD [7, 8] is an adaptation of the FreeBSD [23] operating system, supporting CHERI-enabled instruction sets, and a hybrid memory protection model based on both virtual memory and CHERI capabilities.

ELF

The *Executable and Linkable Format* [12] is a common file format used by UNIX systems to represent various types of binary files at various stages of the compilation process. It is the file format of choice in FreeBSD and CheriBSD. An ELF file can be one of three types: a relocatable object file, an executable, or a shared object.

A relocatable object file contains compiled code with holes for undefined symbols, a symbol table listing all the defined and undefined symbols, and a list of *relocations* telling the static linker how to fill in these gaps when combining them into an executable or shared library.

An executable can be directly executed on the processor; it contains segments such as code and data segments, the names and required versions of dynamically linked symbols, the shared libraries needed, etc. The code and data segments are mapped into process memory by the run-time linker (also known as the *loader*), and the rest is processed to perform dynamic linking.

A shared object also contains code and data segments to be mapped into process memory, the provided and undefined symbol names and versions, and any further shared libraries needed. Run-time relocations inform the run-time linker about any load-time work, e.g., filling in a GOT entry with an absolute address.

RTLD

RTLD is the run-time loader and linker in FreeBSD and CheriBSD. It is an intricate user-space program responsible for loading the program and shared library segments into process memory, and performing dynamic linking. On process creation, the kernel allocates the necessary resources for the process and then passes control to the RTLD start-up routine which maps the executable and any chains of needed shared libraries into process memory. It performs any run-time relocations such as initialising global pointers and the GOTs of the objects, either to RTLD's resolver if lazy binding is enabled, or the eagerly resolved symbol addresses otherwise. Symbol lookups are performed by traversing the DAG of the opened shared objects.

CheriBSD's CHERI-aware RTLD is largely the same, but pointers becoming capabilities requires that all pointers be derived from source capabilities returned by `mmap()` [27] with bounds and permissions properly set, including GOT entries.

`dl*` API

RTLD provides an interface for loading and closing shared libraries at run time, as opposed to load time. This can be used to implement programs supporting dynamically loadable plug-ins, language runtimes, etc.

```
void     *dlopen(const char *path, int mode);
void     *dlsym(void * __restrict handle, const char * __restrict symbol);
int       dlclose(void *handle);
char     *dlerror(void);
```

Listing 2.3: *Excerpt from the API provided by RTLD for run-time loading of shared libraries (in `dlfcn.h`)*

2.1.4 Arm Morello

The Morello project by Arm [5] is a research program extending the Armv8 64 bit ISA with CHERI capabilities resulting in the Morello ISA [3], designing a superscalar processor and SoC implementing it, and maintaining a software ecosystem for it.

The Morello ISA has a 129-bit capability view of all general-purpose and some special-purpose registers.

For controlled non-monotonicity, the Morello ISA complements sealed capabilities by the *Executive–Restricted mode mechanism*. User-space code at any point in time runs in either Executive mode or Restricted mode, determined by the Executive permission bit in the PCC. Some registers are *banked*, i.e., there are two versions of these registers and the mode determines which register is accessed. Code in Executive mode can additionally read and write the Restricted mode registers, but not the other way around. Transitions between different modes can occur on capability branch or return instructions if the target or return capability has a different Executive permission bit than the current PCC. Notably, the capability stack pointer (CSP) is one of these banked registers. To switch from Executive mode to Restricted mode, the specialised `BRR` and `RETR` instructions need to be used, preventing unwanted transitions.

This mechanism can be used to separate protection domains: a Restricted domain can invoke a trusted function in an Executive domain operating on a data structure (e.g., trusted stack) without exposing the data structure to the caller at all. This transition does not require any intervention from the kernel, which makes it scalable.

2.1.5 Linker namespaces

The idea to use run-time linkage to provide different domains of execution is not a novelty. There are relevant examples of run-time linkers providing multiple namespaces instead of a flat namespace where any entity is allowed to reference any other entity.

Android

The Android run-time linker [1] implements linker namespaces which provide fine-grained control over how shared objects can be linked together.

Android provides the Vendor Native Development Kit (VNDK) which is a consistent API and ABI to be used by hardware vendors to implement Hardware Abstraction Layer (HAL) modules for their devices. By keeping this interface fixed, whenever the framework modules are updated only the VNDK needs to be changed and the vendor modules do not need to be reimplemented.

Linker namespaces are provided by the run-time linker to *remove unwanted dependencies* and isolate namespaces so that symbols do not conflict, e.g., both a framework and a vendor version of `libcutils.so` is provided.

It provides mechanisms to create new namespaces, and define the visibility of shared objects across namespaces. When the linker is asked to load a shared object via `dlopen()`, it determines the current namespace by looking up which shared object the return address is coming from, then searches for the shared object in the paths corresponding to the namespace, falling back on other exported shared objects from other namespaces.

Note that the main goal of this modular design is to remove dependencies, not to provide security guarantees: for example, the caller might forge a return address from `libc` (cf. return-oriented programming), so that the system namespace will be determined, granting access to system symbols.

Java class loading

Java class loaders [20] are responsible for dynamically loading classes to the JVM before they are used, often lazily when the class is first referenced by the program. This is the JVM equivalent of run-time loading and linking.

The JVM maintains a tree-like hierarchy of class loaders: the bootstrap class loader is at the root of the tree, responsible for loading the class loaders themselves, the extension class loader loads the application classes, and the system class loader loads standard system classes. The application itself can register custom class loaders, e.g., for loading applets rendering webpages received from a web server.

The JVM restricts the visibility of classes based on which class loader the class has been loaded by. A class loaded by class loader A can only reference classes that have been, or can be, loaded by A or any of its ancestors. This defines a hierarchy of namespaces.

It means that name clashes are allowed, but an untrusted web applet cannot maliciously overwrite the `String` class for example. Furthermore, Java has the secure class loading mechanism [11] to enforce security policies based on class loader namespaces.

Trusting the Java runtime, this mechanism provides good security guarantees because Java code cannot forge pointers to violate this separation of protection domains. However, if we want to do something similar for C shared objects, we do need to worry about memory safety: this is where CHERI helps.

2.2 Sources of privilege of running code

Under CHERI, every memory operation or function call needs to be authorised by an architectural capability. Therefore, at any point in execution the privileges[1] of the running

[1] *Privilege* is used to mean any kind of right, not just for some special system resources.

code are determined by the set of architectural capabilities reachable from the (capability) register file. In particular, running code has access to

- ▹ any piece of memory authorised by general-purpose capability registers,
- ▹ any stack-allocated local variables, buffers, arguments, or return values through the (capability) stack pointer (CSP) or the frame pointer (FP),
- ▹ any piece of memory authorised by a capability within some other reachable memory area (*transitivity*);
- ▹ furthermore, through the program counter capability (PCC), depending on its permission bits, the running code can have access to:
  - ∗ the Executive version of banked registers to be accessed (§ 2.1.4), or
  - ∗ system calls to be performed,

  and if dynamically linked library code is executed, the symbols in the GOT, accessible through PCC-relative loads for example.

By limiting the set of reachable capabilities we limit what arbitrary code inside the sandboxed library can potentially do. This is a key observation when designing a CHERI capability-based sandboxing mechanism.

2.3 Requirements analysis

The most important design goals of the sandboxing mechanism are:

- ▹ **Security:** taking away as much privilege from the sandboxed library as possible. We have seen above that in CHERI C, sources of privilege are capabilities: either to access data or invoke functions. Therefore the aim is to minimise the capabilities reachable from the sandbox.
- ▹ **Compatibility:** supporting real-world libraries to be sandboxed with as little disruption as possible. While minor source code changes to sandboxed libraries and applications using them are acceptable, they should be minimised, and the linker should be binary compatible with any non-sandboxed applications.
- ▹ **Scalability:** providing a scalable mechanism for sandboxing libraries, both in terms of speed of domain switching and the number of sandboxes. CHERI achieves separation of domains within a single address space, promising software compartmentalisation without incurring the costs of process creation, etc., required by MMU-based approaches.
- ▹ **Generalisability:** exploring the viability of this shared-object-based model of compartmentalisation in a more general setting. There is an ongoing research effort to provide scalable and low-overhead compartmentalisation for both kernel and user-space software. This prototype ought to give insight into the feasibility of this approach in user space.

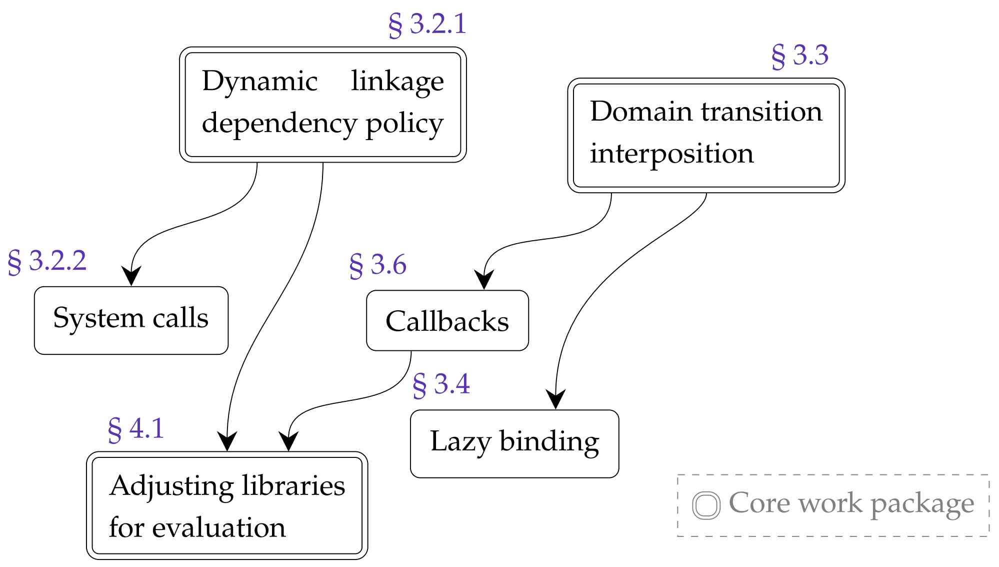

§ 3.2.1
Dynamic linkage dependency policy
§ 3.3
Domain transition interposition
§ 3.2.2
System calls
§ 3.6
Callbacks
§ 3.4
Lazy binding
§ 4.1
Adjusting libraries for evaluation
Core work package

Figure 2.3: *Dependencies between work packages; order of completion approximately corresponds to the vertical ordering of work packages*

Investigating numerous candidate libraries, such as `libpng`, `zlib`, `libxml2`, and Turbo JPEG revealed that sandboxing pure leaf libraries without any external function dependencies is insufficient, as all of these libraries depend on a range of `libc` or other symbols, or expose sandbox-unfriendly APIs such as user callbacks, etc. The project's work packages were revised accordingly to include an implementation of a dependency policy enforcement mechanism (§ 3.2), and extensions such as supporting callbacks (§ 3.6) were prioritised.

2.4 Starting point

The project is about extending the existing CheriBSD run-time linker with a sandboxing mechanism and therefore relies on its existing codebase. Furthermore, as mentioned in § 1.3, two pieces of code were reused from a project running in parallel with this project, which is a change from the original starting point (appendix B). The overlap in the two projects was unknown at the time of writing the project proposal, and re-implementing the same components would not have added value to this project.

I had no experience with CHERI or systems programming.

2.5 Software engineering approach

I followed an iterative development approach to minimise risk, implementing the security guarantees one by one. This allowed me to analyse the current version from an adversarial perspective and further refine the requirements. The deliverable is made up of work packages of different priorities, as shown in figure 2.3. Alternative plans were made in case the non-core work package implementing callbacks would not be successful, or the test hardware was not ready by the evaluation phase of the project.

I used Git to version-control code, evaluation notebooks, and the dissertation source, making regular backups to GitHub and iCloud, and also to rebase changes across different forks.

Alongside development, I maintained a unit test suite written using the ATF test library [13] and Kyua runtime [6] which is the choice by FreeBSD and CheriBSD, and the test programs of the evaluation corpus (§ 4.1) were used as an end-to-end test to demonstrate the correctness of the implementation.

Many of the tools used were irreplaceable parts of the CHERI toolchain, such as CHERI forks of LLVM, QEMU, GDB, or CheriBSD itself[2]. These pieces of software are all in the research phase; some bugs were discovered throughout the project and reported to the maintainers.

Some further tools were used for benchmarking, such as `libpmc`, an interface for hardware performance counters (easy-to-use, distributed as part of FreeBSD/CheriBSD[3], and has been adapted for Morello), the `libxo` utility to print data in JSON or XML format (useful for exporting the performance measurements, also distributed as part of FreeBSD/CheriBSD), Python notebooks with common packages for analysing and visualising the data, providing an efficient workflow.

All of these have permissive licences authorising my use to the best of my knowledge. My forks of the repositories are distributed under the original licences, and I am not redistributing any source code or binary not part of the original repository, which would require me to include their licences too.

[2]These are released under various permissive licences. URL: `https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-software.html` (accessed 1-May-2022)

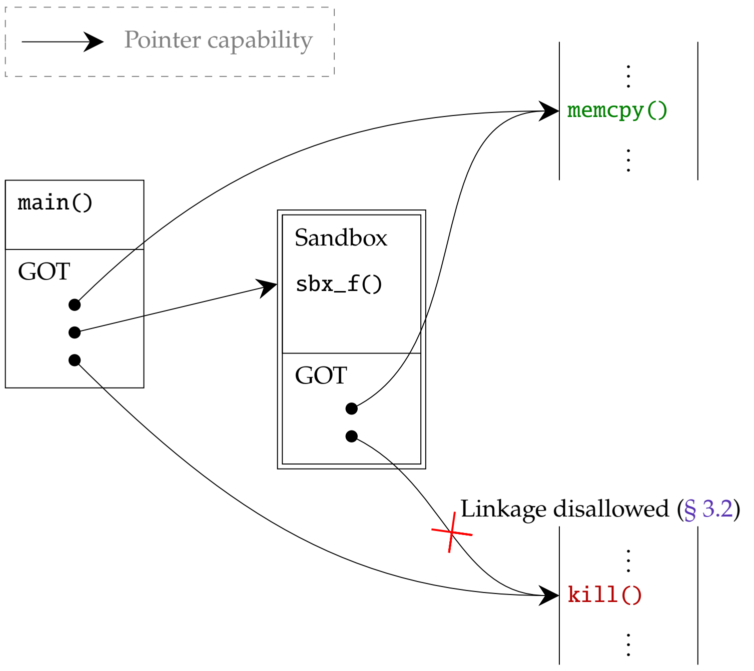
[3]Licence: URL: `https://github.com/freebsd/freebsd-src/blob/main/COPYRIGHT` (accessed 1-May-2022)

Chapter 3: Implementation

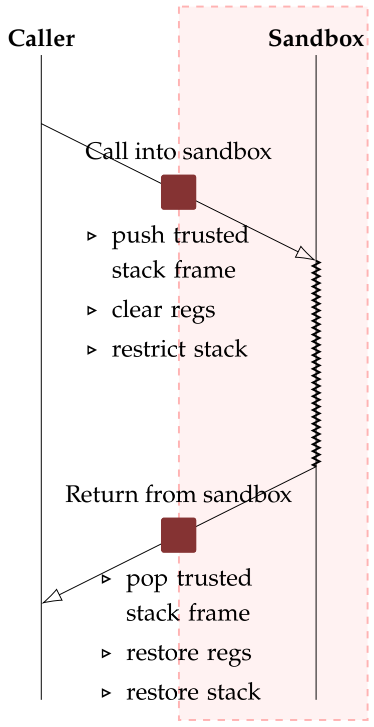
In this chapter, I describe the design and implementation of the modified run-time linker capable of sandboxing libraries, for CheriBSD running on Arm Morello architecture.

I extended the `dl*` API with a new function, `dlopen_sandbox()`, which loads a shared object in a sandboxed environment. This API is a convenient choice for my prototype because it did not require modifications to the ELF specification and the static linker, which would be necessary were sandboxing done implicitly at load time. However, it would be possible to introduce special ELF entries telling RTLD to set up objects in a sandbox, resulting in less source code disruption for existing software not using the explicit `dl*` API. Most of this implementation would immediately translate to that setting too.

3.1 Design overview



(a) *When RTLD loads the sandboxed library at run time it limits what external symbols it can access, determined by the access control policy*



(b) *Control flow when calling into the sandbox; a piece of domain transition code restricts privileges, and restores them on return* (§ 3.3)

Figure 3.1: *Design overview*

A sandbox is a restricted execution environment. We observed in § 2.2 that the privileges of the running code are determined by the reachable architectural capabilities from the register file. We limit what capabilities are reachable from within the sandbox in two ways. Firstly, we limit the flow of capabilities from other domains into the sandbox, secondly, we limit what functions the sandbox is allowed to call through dynamic linkage or via the system call mechanism.

The sandboxed library is allowed to call some trusted functions, but not other unsafe functions. The policy and mechanism for limiting what external library functions or system calls the sandbox is allowed to invoke are described in § 3.2.

Architectural capabilities might be passed into the shared objects in several different ways. They might be intentionally passed as arguments into the sandbox[1], or unwanted capabilities might leak into the sandbox through registers or the stack; we try to minimise the latter.

There are a few places where domain transitions occur between the sandbox and the trusted domain. Firstly, transitions from the trusted domain into the sandbox take place when a function in the sandboxed library is called from a trusted caller, or when a trusted function called by the sandbox returns; in these places, ideally, no unwanted capabilities should leak into the sandbox. Secondly, a transition from the sandbox to the trusted domain takes place when a sandboxed function returns into the trusted caller, or when a trusted function is called by the sandbox; in these places, the trusted domain's privileges are restored via controlled non-monotonicity.

To minimise the flow of unwanted capabilities into the sandbox, pieces of domain transition code (also referred to as *trampolines*) *interpose* cross-domain function calls and returns, ensuring that privileges are reduced and restored as appropriate when making a transition across sandbox boundary. The mechanism is described in § 3.3.

Furthermore, § 3.5 describes the changes needed to make sure the GOT entries do not grant excessive privileges to the sandbox.

3.2 Dependency policy

This section describes the policy and the mechanism to limit what external functions or system calls the sandbox is allowed to invoke.

3.2.1 External functions

It is an important design choice what external symbols sandboxed libraries are allowed to link against, i.e., what functions they might call or what globals they might access through their GOT, as these are sources of privilege. We consider policy and mechanism separately according to the literature [15, 11, 18].

[1]E.g., a capability pointer to a buffer to operate on might be passed as a function argument into the sandbox. Note that CHERI capabilities allow such buffers of memory to be passed into a different protection domain without the need for any copying.

Table 3.1: *Whitelisted symbols sandboxed libraries are allowed to link against*

Symbol	Library	Explanation
`__cxa_finalize`	`libc`	Weak symbol calling C++ destructors or functions registered with `atexit()`, used in the finaliser.
`_Jv_RegisterClasses`	`libc`	Weak symbol used by the GNU Java compiler called in the start-up code.
`__stack_chk_fail`	`libc`	Fail because stack overflow has been detected.
`__stack_chk_guard`	`libc`	Guard used by stack overflow detection.
`malloc`	`libc`	Allocate memory from the heap.
`calloc`	`libc`	Allocate multiple memory chunks from the heap.
`free`	`libc`	Free memory allocated from the heap, assuming it has a capability to it with write permission.
`memcmp`	`libc`	Compare two chunks of memory, assuming it has capabilities to them with read permission.
`memcpy`	`libc`	Copies a chunk of memory into another chunk of memory, assuming it has a capability to the whole of the source chunk with read permission, and a capability to the whole of the destination chunk with write permission.
`memset`	`libc`	Writes a value to each word in a chunk of memory given a capability with write permission.
`strlen`	`libc`	Finds the length of the string given a read capability.
`gmtime`	`libc`	Converts time representations.
`abort`	`libc`	Aborts execution of the process.
`atof`	`libm`	Converts string to float.
`frexp`	`libm`	Writes a float as a fractional part times a power of two.
`modf`	`libm`	Splits a float into integer and fractional parts.
`pow`	`libm`	Calculates a power.

The most restrictive policy is to disallow any calls to external references, requiring the sandboxed library to be a "leaf library" without any dependencies. This is not feasible for the vast majority of software libraries: even simple libraries tend to depend on standard library functions such as `memcpy()`, `memset()`, etc.

I support policies where the sandboxed object is allowed to link against a *whitelist* of symbols that are considered "safe". For example, copying a chunk of memory via architectural support through `memcpy()` is considered safe, but functions for deleting files, killing processes, or accessing private keys are considered unsafe. Table 3.1 shows a minimal list of symbols I identified by looking at dependencies of the libraries I wanted to sandbox, as opposed to a complete list of safe symbols.

A notable design choice is the use of heap memory. Most libraries make use of the heap internally, therefore I allowed sandboxes to allocate and free memory on the caller's heap. The security implications of this are discussed in § 4.2.2.

I implemented a mechanism in RTLD to enforce the policy. When a library is loaded in a sandbox at run time (via `dlopen_sandbox()`), RTLD walks through the dynamic symbol table in its ELF and checks all external symbols – function or object – against the whitelist.

Supporting more flexible policies is possible. We might want `libc` itself to annotate in its ELF which symbols are safe for sandboxing. Alternatively, we could create a linker namespace mechanism, similar to the one implemented by the Android run-time linker described in § 2.1.5. The libraries in the sandbox namespace would only contain safe symbols, e.g., `sandbox/libc` would contain a safe subset of `system/libc`, and the library to link against would be picked depending on whether the caller is from a sandbox or not.

3.2.2 System calls

System calls provide a mechanism for user-space programs to invoke functionality provided by the kernel. They are performed by executing a special instruction that triggers the supervisor bit to be set and control to be passed to the kernel's system call handler. Allowing system calls to be made by the sandbox would grant it excessive privileges: e.g., making an `execve()` system call grants arbitrary code execution capability.

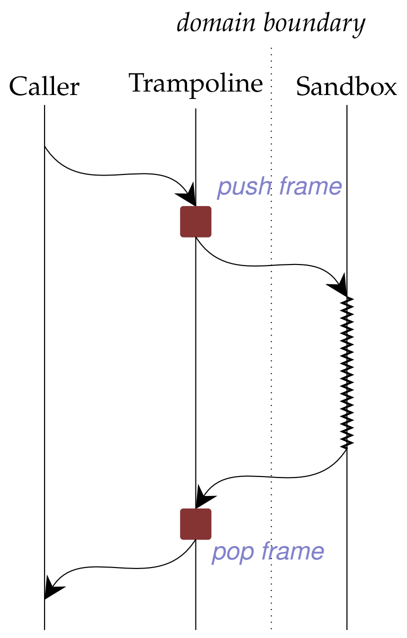
I prevented sandboxed library code from making "raw" system calls; only safe system calls can be made through the trusted `libc` wrappers for them. CheriBSD provides a mechanism to control the permission to make system calls: the PCC has a permission bit that the kernel's system call handler checks, and if not set, the system call fails. Clearing this system call permission bit in the code segments of the sandboxed shared object ensures that no raw system calls can be made. Therefore, any system calls made through the `libc` wrappers will appear in the ELF dynamic symbol table, making it amenable to be managed by the dependency policy described above.

3.3 Trampolines

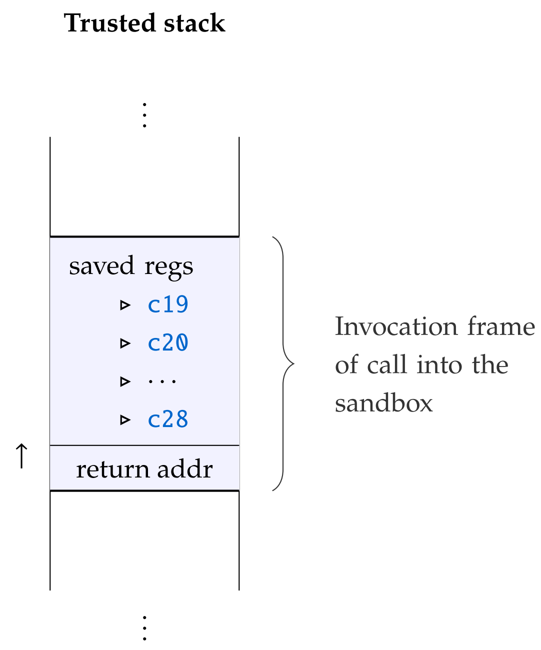
In the previous section, I described how it is guaranteed that the sandboxed library can only call safe functions. In this section, we focus on the pieces of domain transition code interposing cross-domain function calls and returns, which is the mechanism for restricting the unwanted flow of capabilities into the sandbox.

Trampoline is a generic term for pieces of code affecting control flow in a way that is invisible at the source-code level. They often interpose function calls and do some additional work before jumping into the function. For example, lazy binding (§ 2.1.2) is implemented in a trampoline that calls RTLD's symbol lookup and binding code, then directly jumps to the resolved function and at the source-code level, it seems as if the function had been called normally.

I used trampolines to interpose transitions between the sandboxed and the trusted domain, restricting and restoring privileges as appropriate. The following sections describe the contents of these trampolines and the design choices made.



(a) *When calling a sandboxed function from the trusted domain a trampoline sets up the restricted security domain and restores the privileges on return*



(b) *Layout of the call-into-sandbox trusted stack frame*

Figure 3.2: *Trampoline interposing calls into the sandbox*

3.3.1 Trusted stack

We do not want to push a stack frame on the conventional call stack for the trampolines because of the Morello function calling convention [4]: function arguments and return values might be passed on the stack in certain scenarios, and injecting an extra stack frame in-between the caller and the callee would disrupt this mechanism. Therefore, I used a *trusted stack* to store stack frames of cross-domain function invocations.

Maintaining a trusted stack also has the advantage that the stack can be unwound across different domains if a sandboxed library faults. Although an important future direction, recovery from faults is not within the scope of this project.

A trusted stack is also necessary if we were to provide different stacks to different protection domains, a suggestion we will see in § 4.2.2; a central stack is needed to keep track of function invocations, but this is also out of scope.

I made use of an existing implementation of a per-thread stack by Dapeng Gao (see § 1.3), mapping fresh pages into memory on demand.

3.3.2 Call into sandbox

The first trampoline is invoked when calling or returning from a sandboxed function. Its high-level assembly code can be seen in listing 3.1.

On Morello, being an Arm architecture, the function's return address is stored in register `c30`. For the target function to return into the trampoline the return value is saved on

```
call  push_trusted_stack_frame

// Push return address (c30)
str   c30, [c9, #16]!

// Save callee-saved capability registers on the trusted stack
stp c19, c20, [c9, #16]!
...
stp c27, c28, [c9, #32]!

// Clear caller-saved capability registers (non-argument temporaries)
mov x9, xzr
...
mov x18, xzr
// Clear callee-saved registers that are stored on the trusted stack
mov x19, xzr
...
mov x28, xzr

// Set Restricted SP and FP
... Calculate desired bounds
scbnds  RCSP_EL0, csp, bounds
cpyvalue  c29, RCSP_EL0, c29

// Jump to target function in sandbox
blrr  function_in_sandbox

// Restore Executive SP (csp) and FP (c29)
cpyvalue  c29, csp, c29
cpyvalue  csp, csp, RCSP_EL0

call  pop_trusted_stack_frame
// c9 stores the popped stack frame
// Restore return address
ldr c30, [c9, #16]!

// Restore callee-saved registers from the trusted stack
ldp c19, c20, [c9, #16]!
...
ldp c27, c28, [c9, #32]!

// Return to caller
ret c30
```

Listing 3.1: *High-level assembly code of the trampoline interposing calls into the sandbox*

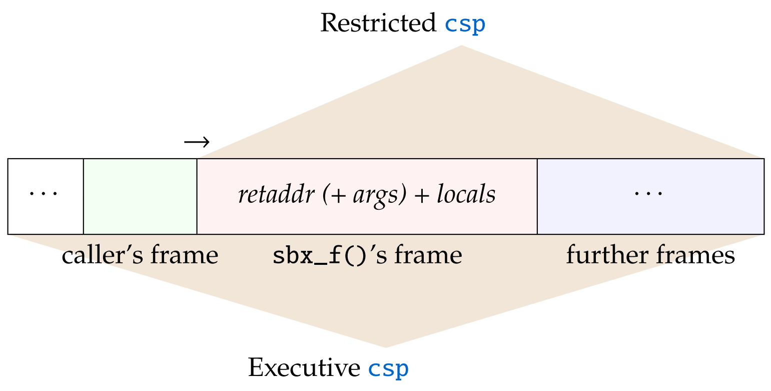


Figure 3.3: *The trampoline restricts the capability bounds of the stack pointer (`csp`) and frame pointer when calling a sandboxed function (`sbx_f()`), restoring them on return; it guarantees* spatial *safety but not* temporal *safety of the stack*

the trusted stack and the address of the trampoline's return side is loaded into `c30`.

Non-argument temporary registers can potentially leak unwanted capabilities into the sandbox, therefore they are cleared by the trampoline. Caller-saved temporary registers will have been saved by the caller and can simply be cleared (`c9`-`c18`), whereas callee-saved temporaries need to be saved on the trusted stack before being cleared, and restored before returning into the caller (`c19`-`c28`).

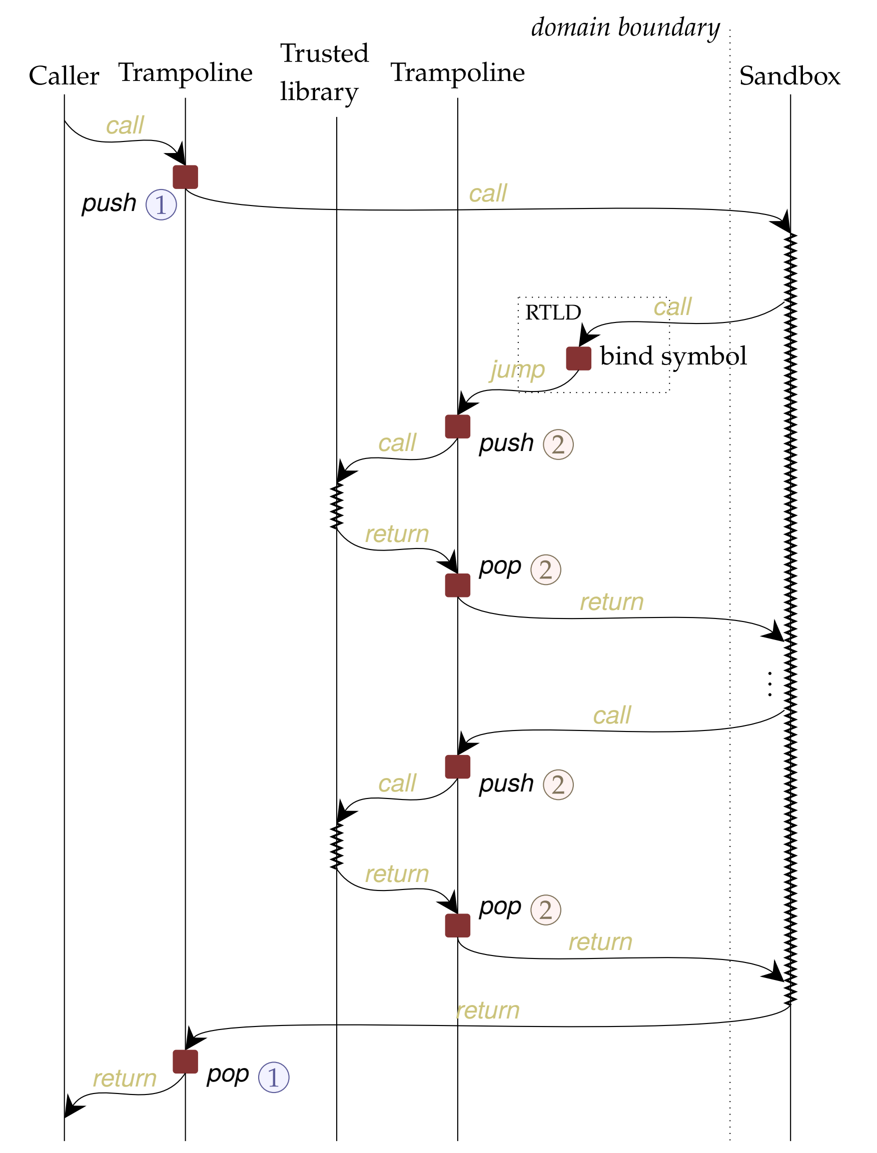
Architectural capabilities might leak on the stack: either from stack frames below the current one (e.g., local variables of the caller), or from no-longer-active stack frames overlapping with or above the current one. The former can be prevented by restricting the bounds of the stack and frame pointers before entering the sandbox and restored on return (shown in figure 3.3), the latter temporal safety issue is not handled by this prototype (a suggestion is presented in § 4.2.2).

For efficiency, we make use of the Morello Executive–Restricted modes (§ 2.1.4). Sandboxed library code runs in Restricted mode, everything else runs in Executive mode. While sandboxed code is running, the stack pointer with full bounds is stored in the Executive version of the banked stack pointer. To implement this, the Executive permission bit in the sandboxed library's code segments is cleared when mapped into memory.

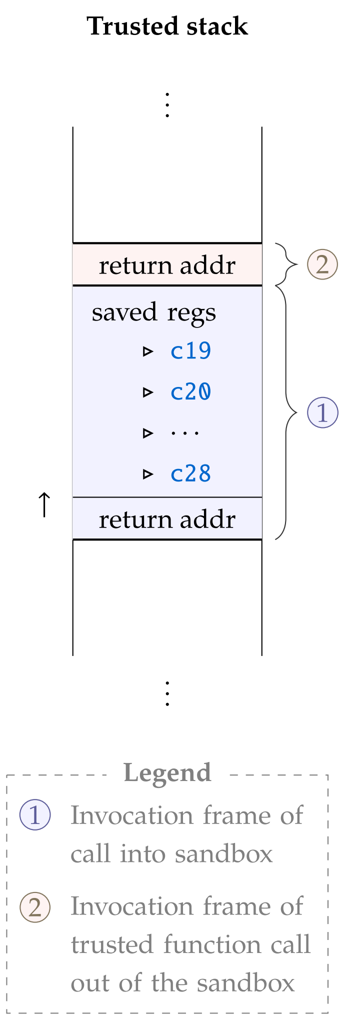
When functions from the sandboxed library are resolved via `dlsym()`, RTLD wraps the function with this trampoline.

3.3.3 Call out of the sandbox

The second trampoline is invoked when calling a trusted symbol from within the sandbox or returning from it. It does similar things as the first trampoline but in a different order. Its high-level code is shown in appendix A. First, the stack bounds are restored, then the return address from the sandbox is pushed onto the trusted stack, and the target trusted function is invoked. On return, the return address is popped from the trusted stack, and before passing control back to the sandbox the stack bounds are restricted and the



(a) *When calling a trusted function from within the sandbox a trampoline sets up the trusted domain, and restricts the privileges on return; a scenario is shown with lazy binding enabled*


Trusted stack
return addr
saved regs
▹ c19
▹ c20
▹ ...
▹ c28
return addr
2
1
Legend
1 Invocation frame of call into sandbox
2 Invocation frame of trusted function call out of the sandbox

(b) *Trusted stack state when inside a trusted function called by the sandbox*

Figure 3.4: *Trampolines when calling out of the sandbox*

caller-saved temporary registers (c9-c18) are cleared to prevent leakage of capabilities into the sandbox.

When binding the function by installing the trusted function pointer into the sandboxed object's GOT, RTLD first wraps it with this trampoline.

3.3.4 Allocating trampolines per function

To help branch prediction, I made the design choice to stamp out multiple copies of trampolines, one for each target function. Branch predictors try to recognise branches and predict their targets and outcomes in the first few pipeline stages to help reduce branch penalties. They consist of three main components:

- the Branch Target Buffer, mapping program counter values corresponding to branch instructions to branch target addresses (and potentially branch type, i.e., conditional or unconditional);
- the component predicting if the branch will be taken or not (often using local and global branch history),
- and the Return Address Stack, maintaining a stack of return addresses for return instructions to be predicted correctly.

If a single copy of the trampoline were to handle jumps to different functions, the Branch Target Buffer would not be able to associate a single target address with the trampoline's branch instruction. One copy of a trampoline per function ensures that one branch address is associated with a single target.

The use of "branch and link" instruction in the trampoline triggers the Return Address Stack to save the trampoline's address, so that it can predict the return address of the target function.

I reused some existing code by Dapeng Gao (see § 1.3) to implement stamping out multiple copies of the trampolines, allocating fresh pages on demand.

3.3.5 `-Bsymbolic`

By default, if shared object code calls an exported function or references exported data from the *same* shared object, this access goes through the GOT. This gives more flexibility for run-time pre-emption of symbols and makes sure that pointer comparison on exported symbols works as expected. There are static linker options to change this behaviour: `-Bsymbolic` and its variants cause intra-library symbol accesses to be performed via direct PC-relative addressing, avoiding the extra indirection through the GOT. For intra-library function calls not to go through trampolines, libraries need to be compiled with these options.

3.4 Lazy binding

My implementation supports lazy binding (§ 2.1.2) of trusted symbols referenced by the sandboxed library. To make lazy binding work, I needed to do two things.

Firstly, the symbol-binding code should wrap the trusted symbol with a trampoline before binding it, i.e., installing it into the sandboxed library's GOT, so that the domain transition code is executed on invocation and return of the trusted function.

Secondly, observe that the (Executive-mode) symbol-binding trampoline – which is invoked when the function is called for the first time – might now be invoked both from the trusted or the sandboxed domain. I needed to make sure that this domain transition, occurring before the trampoline is executed, does not result in an inconsistent state. In particular, the Executive stack pointer needs to be restored if coming from a sandbox in order not to corrupt the stack. Therefore, a branch is performed on the permissions of the calling object[2], and the stack and frame pointers are manipulated if it is sandboxed.

3.5 GOT entries

A source of privilege is the GOT of the sandboxed library (§ 2.1.2). We have to make sure that the sandbox cannot gain excessive privileges through its GOT entries.

The GOT contains function and data pointers to external symbols, installed by the linker. The dependency policy (§ 3.2) only allows safe function or data pointers to be installed into the GOT of the sandboxed library.

Function pointers are *sealed*, making them unmodifiable; the "branch and link" instruction unseals the capability and installs it into the PCC (§ 2.1.1). This way, although the function pointer authorises a range of instructions to be executed, the sandbox can only jump to the function's pre-defined entry point, and cannot fiddle with the pointer (e.g., attempting return-oriented programming). This is an example of controlled non-monotonicity provided by CHERI to be used for privilege escalation.

However, there are two additional GOT entries on Morello used to support lazy binding: a (sealed) function pointer to RTLD's symbol-binding trampoline, and a pointer capability to the shared object's descriptor internal to RTLD, which is passed to the symbol-binding code to determine which object's GOT entry needs binding.

The latter unsealed pointer might provide excessive privileges to the sandbox: it allows traversing the DAG of the needed libraries, getting `libc`'s object descriptor, through which the sandbox could call `execve()` without it being present in the ELF, achieving arbitrary code execution outside the sandbox, e.g., by spawning a shell.[3]

To prevent this, I made RTLD seal it before installing it into the GOT, and unseal it on lazy symbol binding. Sealing or unsealing a capability requires a *seal capability* authorising the operation which can only be obtained from the kernel through a `sysctl()` system call: RTLD obtains such a capability on start-up and stores it in its internal state, not exposing it to the sandbox.

3.6 Callbacks

Some real-world libraries tend to offer callback-oriented APIs for customisation. For example, `libpng` allows callers to register callbacks ranging from custom memory allocators and warning handlers to status callbacks called after a row has been read or written. These are implemented by C function pointers.

[2]Note that the sandbox might try to "forge" a non-sandboxed object pointer. This does not create a vulnerability because the lack of `RETR` instruction in the bound function induces a crash whenever a non-wrapped function attempts to return into the sandbox, preventing any capability leakage into the sandbox.

[3]This exploit was found when performing an adversarial security evaluation of the sandboxing mechanism.

```
void png_set_read_status_fn(
        png_structrp png_ptr,
        void (*read_row_fn)(png_structp, png_uint_32, int));
```

Listing 3.2: *Callback API example from `libpng`*

By default, the sandbox trying to call an injected function pointer crashes on return because of the lack of `RETR` instruction required for Executive–Restricted mode transition.

Therefore, callbacks passed into the sandbox should also be wrapped with a trampoline, just like the trusted `libc` symbols. I exposed an extra API in RTLD doing this, `dlwrap_callback()`, which the caller needs to use on function pointers passed as a callback.

Care must be taken by the caller not to accidentally pass in a function pointer with excessive privileges into the sandbox, as this might be exploited.

```
png_set_read_status_fn(read_ptr, dlwrap_callback(read_row_callback));
```

Listing 3.3: *Code change example required for callback APIs*

3.7 Repository overview

The repository is made up of my fork of the CheriBSD operating system, the adjusted evaluation corpus, benchmarks and analysis. Other than the benchmarks, the rest of my repository extends existing software, referenced in § 2.1.3 and § 4.1, and my contributions are only additions and modifications. Figure 3.5 on the following page shows some directories and files that I worked with. The structure was mostly predetermined.

```
/
├─ cheribsd-rtld-sandbox/ ........ Fork of CheriBSD with the modified RTLD and various bits
│  ├─ libexec/rtld-elf/
│  │  ├─ aarch64/ ........ Morello-specific files such as trampolines, etc.
│  │  ├─ tests/sandbox/ ........ Test libraries and unit test suite (own contribution)
│  │  ├─ rtld.c ........ Core RTLD file
│  │  └─ ... ........ Various other RTLD files were modified or added
│  ├─ lib/libc/ ........ Version of libc with weak symbols introduced
│  ├─ sys/contrib/zlib/ ........ A version of zlib was produced without its file API
│  └─ ...
├─ freebsd-rtld-sandbox/ ........ Fork of FreeBSD with initial non-CHERI specific sandboxing changes
├─ libpng/ ........ Sandboxing-compatible fork of libpng (§ 4.1.3)
│  ├─ scripts/
│  │  └─ pngusr-sandbox.dfa ........ Configuration file specifying which features to be turned on or off
│  ├─ pngtest.c ........ Adapted version of pngtest
│  └─ ...
├─ imagemagick/ ........ ImageMagick fork using sandboxed libpng (§ 4.1.4)
│  ├─ coders/png.c ........ The PNG coder adapted
│  └─ ... ........ Various files changed for CHERI compilation
└─ benchmark/ ........ Benchmarks, files, and scripts to run and evaluate benchmarks (own contribution)
   ├─ evaluation/
   │  └─ micro-bechmark-analysis.ipynb ........ Notebook analysing and visualising benchmark results
   ├─ dlsym-benchmark.c
   └─ ...
```

Figure 3.5: *Repository structure*

Chapter 4: Evaluation

In this chapter, I present the evaluation of the sandboxing mechanism described in chapter 3. The evaluation considers three main aspects: the security guarantees of the mechanism (§ 4.2), the performance overhead incurred (§ 4.3), and the code changes required to the applications and libraries (§ 4.1.6).

4.1 Evaluation corpus

This section introduces the libraries and test applications chosen for evaluation (§§ 4.1.1 to 4.1.4), the code changes required to adapt these (§ 4.1.6), and the data corpus used (§ 4.1.5).

4.1.1 Motivation

Data processing libraries have been the main focus of this project. These libraries – ranging from decompression to image and video processing, etc. – often date back to the last century, rely on legacy code and have a history of a wide range of past vulnerabilities[1]. Despite this, many applications still rely on them: for example, a web browser uses various libraries for decompression, image processing and rendering, data parsing, etc.

Watson, et al. [25] demonstrated that by running these libraries on CHERI, most vulnerabilities affecting integrity and confidentiality are transformed into crashes through memory safety. For example, a buffer overflow vulnerability that reads potentially confidential data out-of-bounds is transformed into a capability bounds fault. However, there are still advantages to sandboxing.

Firstly, arbitrary code execution vulnerabilities are still relevant. MRSC [14] found that at least 18% but potentially a third of memory safety vulnerabilities are not mitigated by CHERI, and some artificially injected vulnerabilities in the QtWebKit JavaScript Core can be exploited to achieve arbitrary command execution. Back doors might also be present in software (supply chain attack). Even if arbitrary code execution is achieved, the successful attacker's privileges are reduced, preventing integrity and confidentiality violations.

Secondly, sandboxing makes it possible to handle availability vulnerabilities (denial-of-service attacks) by recovering from faults in the libraries without crashing the whole process. Returning to the web browser example, the browser can display a "broken image" icon, rather than crashing the whole tab when loading a website containing a maliciously crafted image. While this latter aspect regarding availability is outside the

[1]E.g., 43 CVE vulnerabilities were reported in `libpng` between 2007 and 2020, 9 of which were code execution vulnerabilities. URL: `https://www.cvedetails.com/vendor/7294/Libpng.html` (accessed 7-May-2022)

scope of this project, the implementation with trusted stacks does lend itself to fault recovery through stack unwinding.

4.1.2 `zlib`

Library `zlib` [10] provides compression and decompression based on the DEFLATE algorithm. It has two sets of APIs: one operating on in-memory streams, and another one operating on `gzip` files. For internal memory allocations, it supports a user-defined custom allocator, defaulting to `libc`'s heap allocators. Otherwise, it is a relatively simple library with no dependencies other than `libc`. To make it conform to the dependency policy, I built a version without the `gzip` file API and using the default heap allocators. This took around 30 lines of code and build configuration change.

4.1.3 `libpng`

Library `libpng` [22] contains the reference implementation of the Portable Network Graphics format, providing functions for reading and writing PNG images using a variety of different options. It has a dependency on `zlib` for compression and decompression specified by the format.

The API is highly flexible, allowing users to set custom memory allocators, read and write functions, error handlers, status callbacks invoked after reading or writing a row, etc. For example, in FreeBSD's userspace, out of the 46 ELFs (libraries or executables) referencing `libpng`, 28 of them set custom read or write functions, and even more set custom `longjmp` style error recovery functions. This heavily callback-oriented design necessitated support for passing callback function pointers into the sandbox (§ 3.6).

I built a version of `libpng` suitable for sandboxing. The build system of `libpng` provides a scripting file format for specifying which features a custom build should include. I turned off sandbox-unfriendly APIs – e.g., file APIs and `longjmp` – but kept features still general enough for most applications. To pass the dependency policy, I also had to statically link `zlib` into `libpng`, as opposed to being a shared-object dependency. This required around 70 lines of code change in the build configuration files.

There is a test program provided with `libpng`, `pngtest`, which reads in and writes out a PNG file and verifies its properties. I adapted this program to use the `dl*` API, resolving every `libpng` function used by the program through `dlsym()`, taking 68 lines of code change. To work with the sandboxed `libpng`, I further needed to explicitly wrap callbacks with a trampoline in 11 places (§ 3.3). I used this program both to verify the correctness of the build and sandboxing and as a workload to be benchmarked.

4.1.4 ImageMagick

ImageMagick [16] is a program to edit, create or convert digital images supporting a variety of file formats and transformations. I adjusted it to build for CHERI by fixing a few cases of pointer-integer confusion (see [27]), requiring 13 code changes.

Table 4.1: *Summary of lines of code and configuration changes required to adapt evaluation stack*

Library	Program	Adaptation for		
		CHERI	dl*	sandboxed
zlib		existed		30
libpng		existed		70
	pngtest	0	68	11
	ImageMagick	13	90	6

ImageMagick uses `libpng` for its PNG coder that reads and writes PNG images. I adapted this coder module to use the sandboxed `libpng` described above. I needed to first adapt it to use the `dl*` API, requiring 90 lines of code change and minor build configuration changes, then wrap callbacks with trampolines in 6 places. Its command-line interface has been verified to work correctly on a few example commands.

ImageMagick is a high-level application using `libpng`, which can be seen as a representative example of application programs using data processing libraries when considering realistic performance overheads.

4.1.5 Data corpus

I scraped around 5,000 PNG images from the publicly accessible Cambridge University departmental websites. This corpus has been used in § 4.3.2 to estimate a realistic overhead through image size distribution, and as data input to ImageMagick macro-benchmarks.

4.1.6 Potential for wider use

One of the design goals of this project has been compatibility. When adapting this `zlib`–`libpng`–ImageMagick stack, the source code changes required were modest. The majority of them were due to using the `dl*` API, which can be avoided if implicit sandboxing based on ELF entries is implemented in the future. Although handling callbacks is disruptive to source code, a typical application registers callbacks with the library only a few times on set-up, therefore major code changes are not required.

It is worth noting that security achieved by sandboxing a library is limited by the security of the API itself: if a lot of potentially confidential data is required to be passed as function arguments, sandboxing cannot ensure confidentiality and integrity without changing the API.

The current prototype is quite restrictive in its dependency policy, its limited control-flow interactions between domains, e.g., not supporting `longjmp` or C++ exception handling, etc. Notably, file APIs and many other resources are not provided to the sandbox, supporting which would raise many questions about ownership and access control.

The trade-off between restrictiveness, disruptiveness and security achieved by the prototype fulfils both the design goals and the success criteria of the project.

4.2 Security evaluation

4.2.1 Threat model

We assume a threat model where the caller, i.e., the main program together with a set of libraries, is trusted, and one or more libraries are *untrustworthy* and assumed to be vulnerable to arbitrary code execution attacks. Besides trusting the caller, we have to further assume a trusted computing base (TCB) including the operating system, the run-time linker and the whitelisted `libc` and `libm` symbols. The data operated on by the application is potentially malicious; in fact, we trust the sandboxed libraries until the point when they first operate on untrusted data. We would like to guarantee that attackers cannot violate the confidentiality and integrity of the rest of the application, even if arbitrary code execution is achieved. The attacker's ability to impact the availability of the application by crashing the sandbox is considered acceptable.

An example setup is a web browser rendering pages downloaded from an untrusted web server. To parse HTML and XML files, render images, decode videos, etc., it uses a variety of libraries operating on untrusted data. A malicious server could serve files designed to achieve arbitrary code execution through exploiting vulnerabilities or back doors in such libraries. Having achieved arbitrary code execution capability, a successful attacker might want to leak confidential data from the process address space, such as information about other tabs, session cookies, etc., or violate integrity by maliciously changing the rendering of the page, making unwanted HTTP requests, etc.

4.2.2 Security guarantees

According to § 2.2, the privileges of running code are determined by the set of architectural capabilities reachable from the register file. This section explores each source of privilege, and thoroughly analyses whether and how unwanted capabilities might leak into the sandbox, assuming the implementation from chapter 3.

Capability leakage through the GOT

The symbols in the GOT of the object can be called by the sandbox. The dependency policy (§ 3.2) limits what external functions and objects are accessible through the GOT of the object, guaranteeing that RTLD will not bind any non-whitelisted symbols. The function symbols in the GOT are sealed (§ 2.1.1), preventing any fiddling, e.g., attempting return-oriented programming.

Additional entries are introduced by the lazy binding implementation: to determine which object invoked the lazy binding code, the object's descriptor is stored in the GOT and is passed to RTLD's symbol binding code. This descriptor grants excessive privileges: e.g., by traversing the DAG of the needed libraries the library code could get `libc`'s object descriptor through which it could call `execve()` without it being present in the ELF, allowing arbitrary code execution. To prevent this, this descriptor capability is sealed by RTLD, as described in § 3.5, making it unmodifiable and un-dereferenceable by the sandbox.

This vulnerability was discovered through performing this security evaluation from an adversarial point of view.

Capability leakage through general-purpose registers

The function calls and returns into the sandbox are interposed by trampolines (§ 3.3), limiting the flow of capabilities through the general-purpose capability registers and the stack. Function calls or returns are passed into the sandbox through the argument registers `c0`-`c8` and the return-value register `c9`. Unused argument or return-value registers might be used as temporaries; since the trampoline does not know statically which registers will be used as arguments and return values, it cannot clear the unused ones, potentially leading to unwanted capabilities leaking into the sandbox. To fix this, the compiler could clear unused argument and return-value registers before passing control to the sandbox if cross-domain functions were annotated.

Capability leakage through the stack

The stack and frame pointers authorise access to the stack. As shown in figure 3.3, the trampoline restricts the bounds of these pointers before passing control to the sandbox, so that the sandbox cannot access stack frames below its own frame, e.g., it cannot access local variables of its caller, granting spatial safety. However, stack frames are overwritten with new stack frames, and the lack of temporal safety means that capabilities might leak into the stack frame of the sandboxed function.

To prevent this, since the costs of clearing stack frames are high without dedicated architectural support, one needs to allocate different stacks to different domains, which could be supported through the trusted stack mechanism. This would be a natural further extension of this project.

Capability leakage through signal handling

When an asynchronous signal is delivered to the process, the signal handler registered by the process is invoked in one of the threads of the process. By default, the signal handler uses the thread's stack for saving and restoring the register file, and for local variables of the signal handler. This might lead to capability leakage even if different domains have different stacks, either if the register file of another domain is saved on the stack, or if the signal handler itself leaves capabilities around. UNIX provides an *alternative signal stack* mechanism, which allows a thread to provide an alternative stack to be used when handling some of the signals. Requiring each signal to be handled on an alternative stack would ensure that no capabilities from another domain leak onto the domain's stack.

Capability leakage through the heap

Since the dependency policy allows the sandboxed library to use the caller's heap through the `libc` API, if `libc`'s heap allocator does not guarantee temporal safety, i.e., clearing capabilities before reallocating memory, capabilities might leak into the sandbox. To fix this, `libc` could be adapted to guarantee heap temporal safety, for which there were attempts in [9].

Capability leakage through the PCC

The PCC can authorise operations, too. It grants access to the GOT, it can authorise system calls, and it can authorise access to the Executive versions of the banked registers. The implications of the GOT have been discussed above. The sandbox performing system calls could have detrimental effects: making an `execve()` call would lead to arbitrary code execution. By removing the system call and Executive permission bits of code pointers in the library's mapped code segments, the PCC inside the library will authorise neither system calls nor access to the Executive registers. Therefore, system calls can only be made through the `libc` wrappers which are controlled by the dependency policy.

4.2.3 Security conclusions

The above analysis shows that the capabilities of a successful attacker achieving arbitrary code execution are effectively limited by the implementation. Except for the shortcomings mentioned, any code maliciously executed by attackers can only access the internal library state and function arguments and can call safe `libc` and `libm` functions. Although the shortcomings potentially provide excessive capabilities to the attacker in our threat model, ways have been discussed to address them, which could be implemented in future work.

A further consideration not addressed is *vertical compartmentalisation*. The implemented mechanism provides sandboxing based on *code* (horizontal compartmentalisation): it separates caller code and library code by limiting accesses across the boundary. Another desirable mechanism would be compartmentalisation based on *data* (vertical compartmentalisation): e.g., creating separate domains for each image processed by `libpng`, disallowing access to the other images processed by the library. This is not addressed by the implementation, potentially leading to integrity or confidentiality violations; an effective solution would most likely require substantial code changes in the libraries and is outside the scope of this project.

Regarding availability, crashes in the library currently lead to crashes of the whole application, which the threat model is not concerned with, but it is an important future direction.

4.3 Performance evaluation

One of the hypotheses of the CHERI project is that besides memory safety, it provides scalable intra-process compartmentalisation. Process-based compartmentalisation – e.g., putting `libpng` into its own process – incurs high overheads because of the process-creation costs and the inter-process communication overheads. In this section, I demonstrate that the CHERI-enabled run-time linker approach to sandboxing incurs measurable, but acceptable performance overheads.

Performance overheads are evaluated through benchmarking. The hardware used is an Arm Morello board, with 4 out-of-order superscalar cores running the Morello architecture (§ 2.1.4). This board is somewhat similar in its specifications to those used in smartphones, making it a representative benchmarking setup.

4.3.1 Micro-benchmarks

Micro-benchmarks measure the relative performance of a single well-specified task or component. Below I describe a few micro-benchmarks associated with the overhead incurred by sandboxing: setting up a library, resolving a symbol, and invoking a sandboxed function. Further tasks are also impacted: binding a trusted function – either eagerly or lazily – has similar overheads to resolving a sandboxed symbol with `dlsym()`, and calling a trusted function from inside the sandbox has similar overheads to function calls into the sandbox, therefore their results are omitted here.

To evaluate performance, the following architectural and micro-architectural performance counters were used:

- `INST_RETIRED`: counts the instructions executed.
- `CPU_CYCLES`: counts the CPU cycles to execute the code; might be lower than the number of instructions because of stalls, or higher because of superscalar execution.
- `BR_MIS_PRED`: counts the mispredicted branches. Branch prediction is key in pipelined cores supporting speculative execution. Mispredicted branches might lead to stalls, reduce the number of available instructions and so the amount of exploitable instruction-level parallelism (ILP). Therefore, the number of mispredicted branches contributes significantly to the executed instructions per cycle.
- `L1D_CACHE_REFILL`: counts the allocations in the Level 1 data cache. This is a metric about the cache-friendliness of the code, although not a perfect one. Allocations in the cache occur when a memory access results in a cache miss, and the instruction needs to stall. However, they might be caused by speculatively executed load instructions, or prefetching, which do not necessarily stall execution in an out-of-order superscalar core.

To amortise the probing overhead, warm up the resources, and decrease variance, the micro-benchmark kernels were run in a tight loop, ranging from 100 to $10,000$ iterations depending on the kernel. Micro-benchmarks were pinned to a single CPU core and made sure not to compete with any I/O. Because of the non-determinism stemming from interrupts, CPU loads, memory system response time, etc., $1,000$ data points were collected for each micro-benchmark.

Library set-up

A library is set up in a sandbox if it is opened with the `dlopen_sandbox()` call. This is compared against the existing `dlopen()` call as a baseline. There are overheads associated with setting up a library in the sandbox: removing Executive and system call permissions from the object's mapped code segments, and checking the object's symbol table against the dependency policy, i.e., against the whitelist of symbols. The micro-benchmark opens the library in a sandbox and closes it, simulating an application's start-up and tear-down behaviour.

The micro-benchmark was run on three libraries: `libhello_world` is a simple library exporting a single function symbol, `libz_nofio` exports 60 function symbols and depends on 5 external ones, and `libpng` exports around 150 function symbols and depends on 13. Figure 4.1 shows the absolute and relative overheads of medians. The overhead, which is almost insignificant in the simple library case, scales up to around 20% for the largest of the three libraries. This is explained by the need to scan through the dynamic symbol table of the library's ELF to check against the dependency policy, i.e., that only whitelisted symbols are referenced by the library, which is linear in the length of the dynamic symbol table. The `dlopen_sandbox()` call has worse branch predictability than the baseline, caused by the unpredictability of the matches in the whitelist. The worse

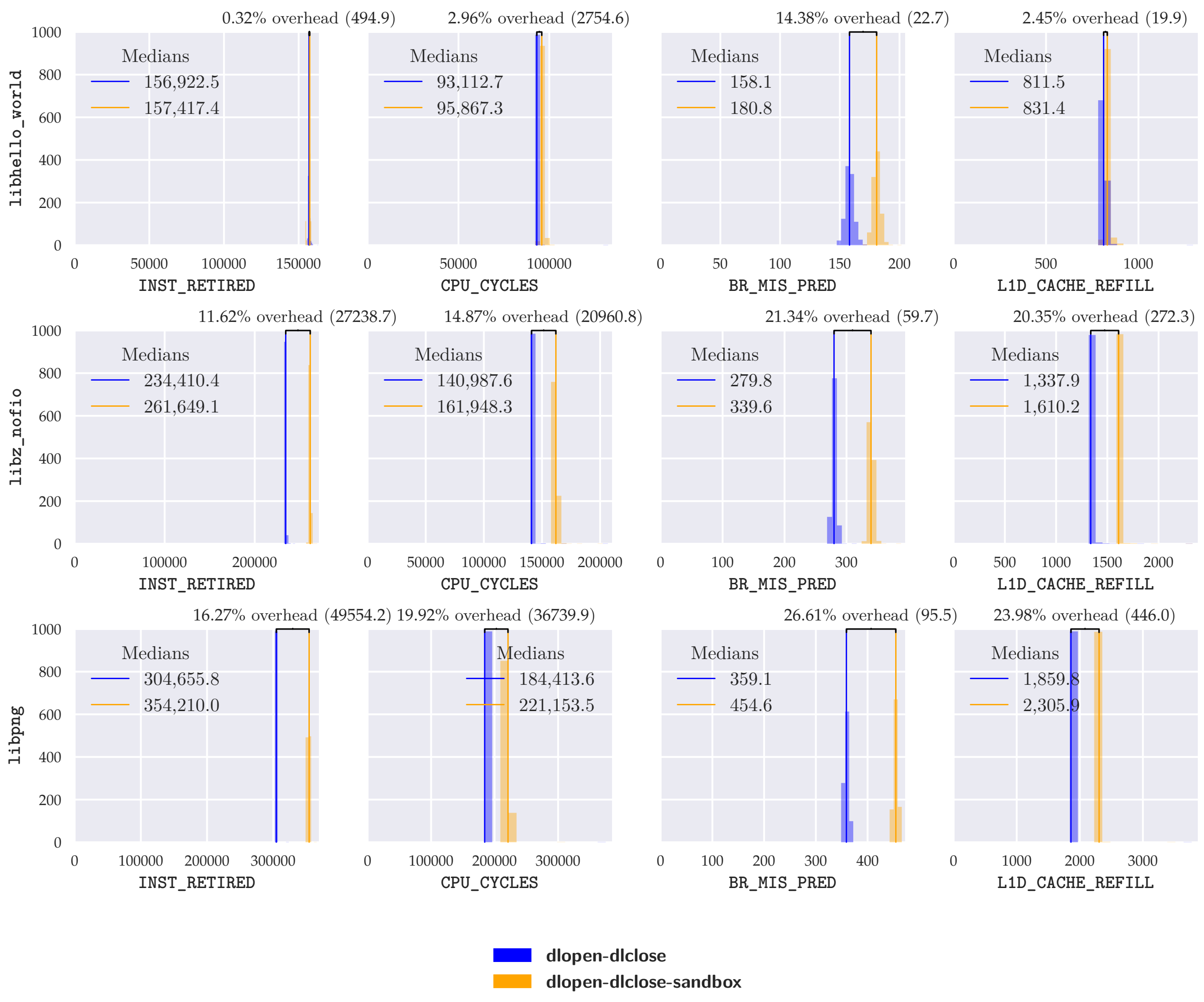


Figure 4.1: *Micro-benchmarking performance overheads associated with setting up the sandbox*

cache behaviour can also be explained by the need to keep the whitelist in the cache and access the whole dynamic symbol table and the corresponding string table entries of the library's ELF.

`dlsym`

Function or object symbols are retrieved from the `dlopen()`-ed library using the `dlsym()` function. When the library is set up in a sandbox, `dlsym()` needs to allocate and set up a fresh copy of the trampoline to intercept the call into the sandbox (§ 3.3.4). Had the design choice been made to use a single copy of the trampoline for all function symbols, this overhead would be avoided, but branch targets from the trampoline would be unpredictable so we would expect higher overall overheads.

The resulting slowdown by a factor of 2 shown in figure 4.2 is significant, but given that programs tend to only resolve each function symbol once, it looks to be an acceptable overhead. The need to `memcpy()` the trampoline code into the fresh trampoline allocation leads to worse cache behaviour, and also introduces an unpredictable branch.

Function call

The overheads associated with function calls are perhaps the most important because they are performed frequently by applications interacting with a sandboxed library. It is also the place where most sandboxing guarantees are enforced. This micro-benchmark considers function calls into a test function returning the address of a static variable, consisting of three instructions. If the function is sandboxed the function call goes through a trampoline, as opposed to the conventional single-instruction function call if the function is not sandboxed. Because the kernel of this micro-benchmark is so small, probing and looping overheads become significant; e.g., the baseline function call takes 19 instructions, even though it only takes four instructions to call and execute the function. The overheads are shown in figure 4.3.

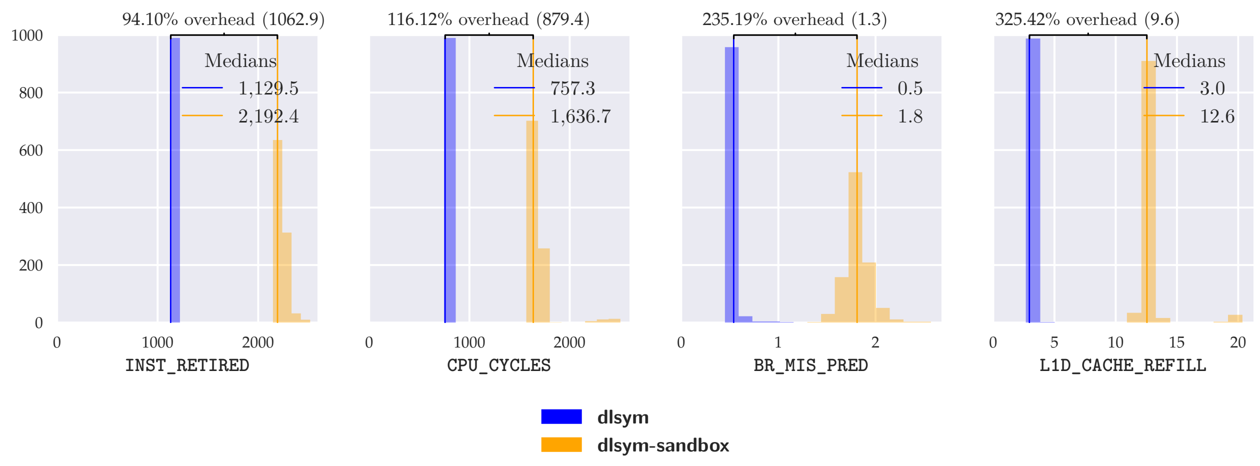


Figure 4.2: *Micro-benchmarking performance overheads associated with resolving a symbol in the sandboxed library*

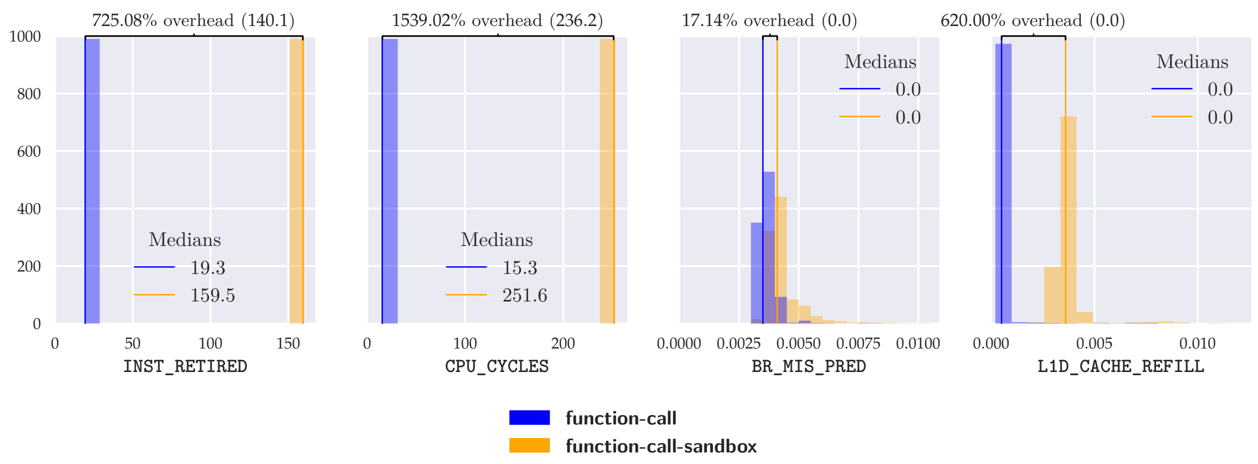


Figure 4.3: *Micro-benchmarking performance overheads associated with calling into the sandbox*

To find the root causes of the overheads, I performed an *ablation study*. I built six intermediate versions of the linker, doing increasingly more work in the function call and return trampolines by each version:

- **empty-call-trampoline**: the call is intercepted by a trampoline which jumps instantly to the target function, and the function return is not intercepted;
- **clear-callee-saved-regs**: the call trampoline clears all caller-saved temporary registers (`c9`–`c18`);
- **return-trampoline-retaddr-on-caller-stack**: function return is also interposed by the trampoline, pushing the original return address onto the normal stack[2];
- **trusted-stack**: records of function invocations across sandbox boundaries are stored on a separate "trusted" stack (§ 3.3.1), containing the return address;
- **clear-and-save-callee-saved-regs**: additionally, callee-saved temporary registers are stored on the trusted stack and cleared before entering the sandbox;
- **restricted-mode**: the sandbox runs in Restricted mode, and the trampoline has to copy stack pointers between the banked stack registers corresponding to the two modes;
- **restrict-sp-fp-bounds**: the bounds of stack and frame pointers are restricted before jumping into the sandbox, and restored after exiting the sandbox (figure 3.3); this reaches the final version.

The results in figure 4.4 show that the single largest source of overhead is associated with maintaining the trusted stack. It requires acquiring the trusted stack pointer and pushing and popping trusted stack frames, mapping and unmapping memory pages on demand. This code introduces some branches, and needs cachelines for the top of the trusted stack,

[2]As discussed in § 3.3, this does not work when function arguments or return values are passed on the stack, but it does not incur the overheads of maintaining the trusted stack

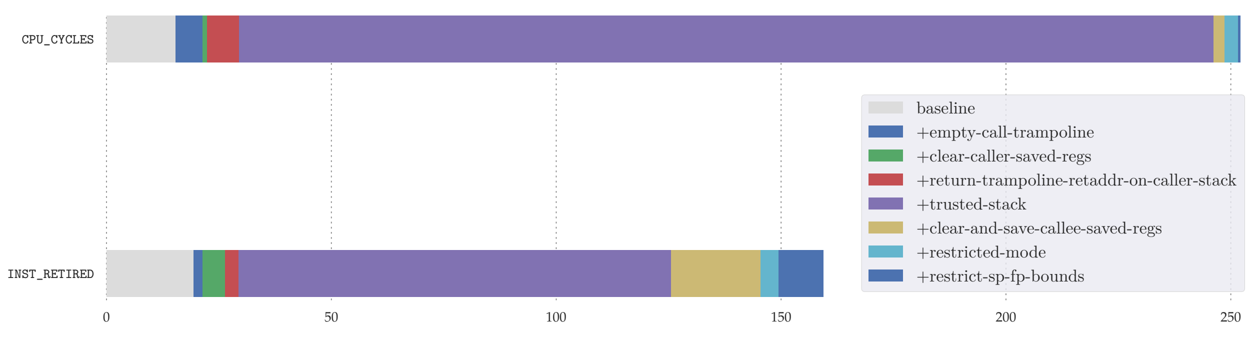


Figure 4.4 and Table 4.2: *Ablation study decomposing the function call overhead; the counter values shown are medians over 1,000 runs*

Stage	INST_RETIRED	CPU_CYCLES	BR_MIS_PRED	L1D_CACHE_REFILL
baseline	19.33	15.35	0.0035	0.0005
+empty-call-trampoline	21.33	21.42	0.0034	0.0008
+clear-caller-saved-regs	26.33	22.36	0.0032	0.0007
+return-trampoline-retaddr-on-caller-stack	29.33	29.48	0.0031	0.0007
+trusted-stack	125.48	246.15	0.0044	0.0033
+clear-and-save-callee-saved-regs	145.48	248.58	0.0048	0.0035
+restricted-mode	149.48	252.08	0.0046	0.0035
+restrict-sp-fp-bounds	159.48	251.60	0.0047	0.0035

although these seem to be predictable.

Note the non-monotonicity in the number of CPU cycles: the last trampoline version – adding the SP and FP bounds manipulation – has more instructions but takes fewer cycles than the previous version. It is due to the out-of-order superscalar design of the core, which can lead to counter-intuitive behaviour. In Morello, in particular, more serialisation seems to be introduced around branches due to pragmatic micro-architectural design choices, so the relative positions of branch instructions affect throughput.

4.3.2 Macro-benchmarks

The micro-benchmarks above show the overheads introduced to the primitive operations such as symbol binding or function call. Although these provide insight into the performance of the implementation, larger, compound, more realistic workloads are needed to evaluate the realistic overheads experienced by end-users when running application programs. To do this, we consider some macro-benchmark workloads from the stack of evaluation libraries introduced in § 4.1.

`pngtest`

This macro-benchmark evaluates sandboxing overheads on `pngtest`, `libpng`'s test program reading, writing and verifying a PNG image. The baseline – using `libpng` through the `dlopen()` API – is compared against the one using the sandboxed `libpng`. The program was run on resized versions of a test image to power-of-two sizes between 1×1 pixels and 1024×1024 pixels, allowing us to see how the sandboxing overhead is amortised as the amount of data processed by the library increases. It is worth noting that the number of symbols resolved is constant, but the number of function calls across the boundary scales with the image size due to callbacks being invoked after each row has been read or written by the library, therefore the asymptotic overhead is non-zero.

Figure 4.5 shows for each image the distribution of the overhead in the number of CPU cycles, i.e., the distribution of the random variable S/B where S and B are the random variables denoting the cycles taken by the sandboxed and baseline versions, respectively. Because it is not possible to get paired samples, i.e., the CPU cycles for both versions under the exact same conditions, S and B are independent which leads to high variance. The fliers show big outliers in both directions; these are explained by the heavy-tailed service time distribution of the non-deterministic interrupt events. Variance decreases as the image size increases because the program itself takes longer to run on larger images, and the random variations introduced by interrupts, CPU and memory subsystem loads (which can be thought of as a sum of independent, identically distributed indicator variables) are averaged out over time, decreasing the variance of the ratio.

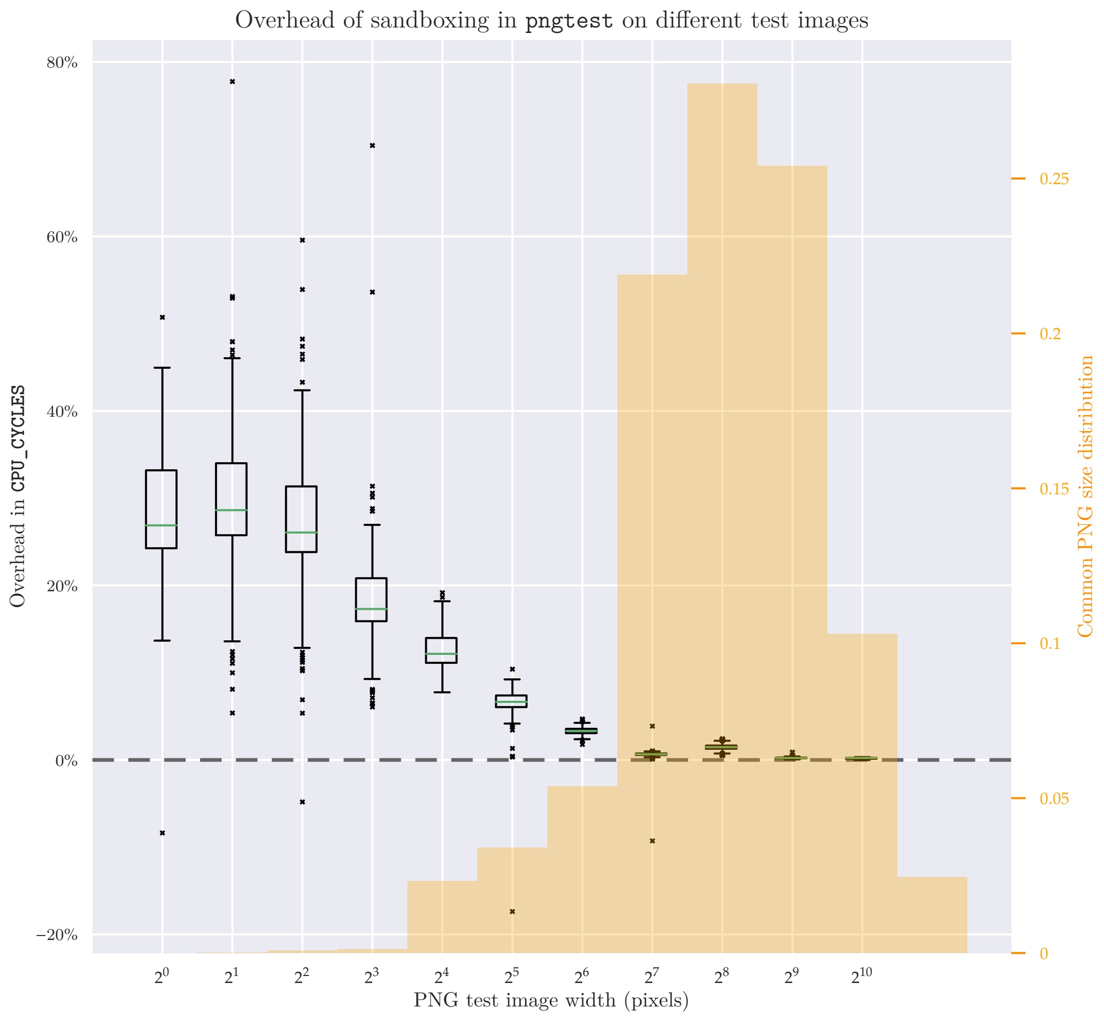


Figure 4.5: *The `pngtest` macro-benchmark on a test image resized to different power-of-two sizes; the PNG size distribution comes from the data corpus in § 4.1.5*

The overlaid histogram shows the PNG image sizes found in the data corpus described in § 4.1.5. While for some small icons the overhead can be over 10%, for most images rendered by the browser the overhead is less than 1%, indicating that the effective overhead that would be experienced by a browser using a sandboxed `libpng` in its rendering code is very minor.

ImageMagick

ImageMagick (§ 4.1.4) is the application-level program of the evaluation corpus, aimed to demonstrate end-user tasks using `libpng` under the hood, giving realistic overheads of the sandboxing mechanism that an end user would experience. Three workloads were selected requiring an increasing amount of compute:

- **read-write**: a basic workload reading a PNG image using `libpng`, converting it into ImageMagick's internal representation, and then writing it out using `libpng`;
- **read-resize-write**: a workload doing the above and resizing the PNG image to be 512 pixels wide;
- **read-resize-improved-write**: the above resizing workload, extended with several filters and transformations to improve the resized image quality, taken as an example from [16], defined by the command below.

```
magick INPUT -colorspace RGB +sigmoidal-contrast 11.6933 -define
   filter:filter=Sinc -define filter:window=Jinc -define filter:lobes=3
   -resize 512 -sigmoidal-contrast 11.6933 -colorspace sRGB OUTPUT
```

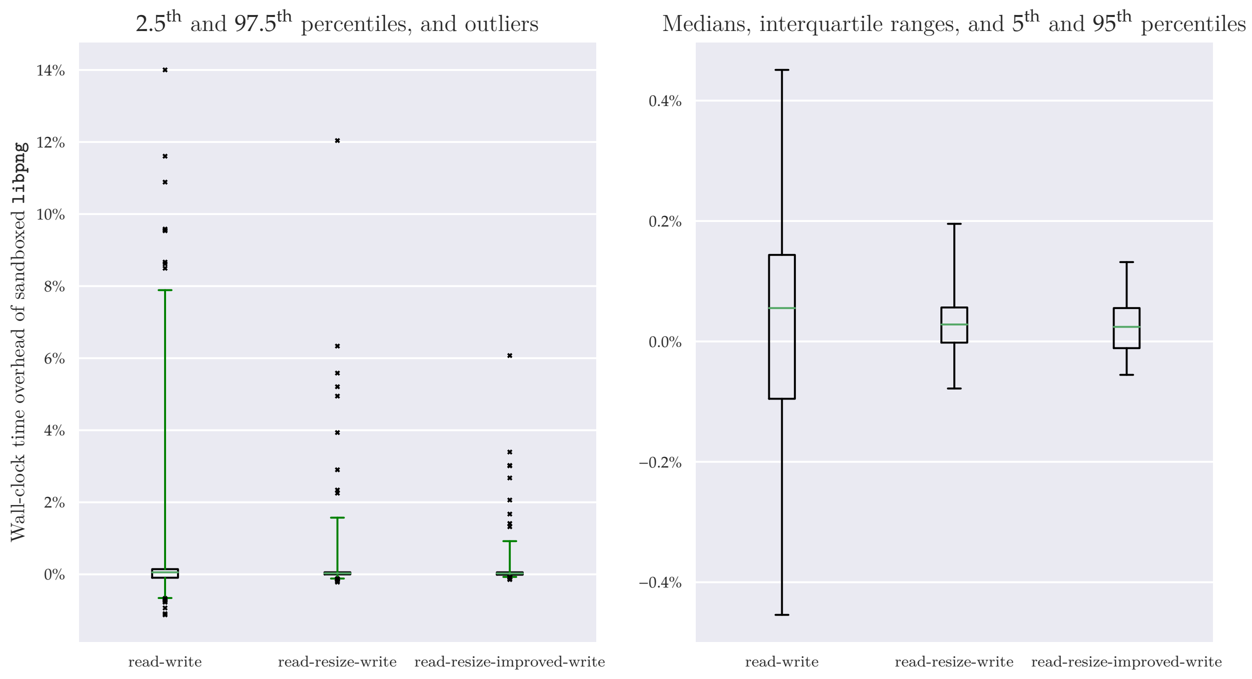


Figure 4.6: *ImageMagick macro-benchmark*

These workloads were run on 359 randomly selected PNGs from the data corpus (§ 4.1.5). Figure 4.6 shows the wall-clock time overhead measured for each workload. Overall, the overhead is insignificant, and the median overhead decreases as the amount of compute grows, as expected due to amortisation. As in the previous macro-benchmark, variance decreases with the time taken to complete the workload, and huge outliers are present. However, the outliers are more skewed towards the large overhead direction because occasionally small images are encountered that experience higher relative overhead. In conclusion, most of the time the end-user overhead is on the order of 0.1% with the occasional outliers, especially if the data operated on is small.

4.4 Limitations of the evaluation

Naturally, the evaluation performed has some limitations. Firstly, the evaluation corpus might not be representative of other data processing libraries or arbitrary software that we might want to sandbox. Although a few other data processing libraries have been inspected, such as `libxml2` or Turbo JPEG, and found to have relatively similar APIs, time did not permit evaluating a more extensive corpus of libraries.

Secondly, performance evaluation was performed on experimental CHERI hardware which can sometimes have odd behaviour due to pragmatic micro-architectural design choices. As an example, we experienced an unexpected non-monotonicity in the number of CPU cycles in the ablation study in § 4.3.1.

Thirdly, the GOT object descriptor vulnerability was only found when performing the security evaluation after benchmarking was performed, therefore the performance results do not include this feature described in § 3.5. The effects of this change are minimal: introducing an extra system call on RTLD start-up, an extra sealing instruction on `dlopen_sandbox()`, and an extra unsealing instruction on lazy binding.

Chapter 5: Conclusions

In conclusion, the previous chapters demonstrate that the success criteria of the project have been met: the approach proposed in chapter 1 has been implemented, as described in chapter 3, and evaluated for security, performance, and code disruption in chapter 4. The implementation has been found to provide fairly strong security guarantees within the assumed threat model, and the sandboxing overheads on end-user applications are projected to be minimal based on the ImageMagick benchmarks (§ 4.3.2).

This work demonstrates that the approach of taking shared objects as the basis of sandboxing is feasible, and provides a scalable mechanism to defend against arbitrary code execution in existing software.

5.1 Future directions

The prototype designed and implemented by this project is not complete. Implementing the suggested changes in § 4.2.2 would provide even stronger security guarantees, restricting the flow of capabilities into the sandbox even further.

The implementation only supports sandboxing libraries with minimal dependencies. An extension would design and implement more flexible and scalable dependency policies (§ 3.2).

Further directions include implementing load-time sandboxing as opposed to the `dl*` API, exploring the availability aspects to support recovery from faulting libraries, or considering a mutually distrusting threat model.

It is an ongoing research effort within the CHERI group to explore this idea further, and with the large interest in CHERI from both academia and industry, there is a chance that CHERI-enabled sandboxing will see deployment in the future.

5.2 Personal reflections

Throughout this project, I learned about CHERI's novel architectural ideas and interacted with its wonderful research community. I also learned a lot about run-time linking internals and systems research in general. One critique of my project management in hindsight is that I should have started implementation earlier, instead of trying to understand all the details first.

Bibliography

[1] Andoid Open Source Project. *Linker Namespace*. [Online; accessed 26-April-2022]. URL: https://source.android.com/devices/architecture/vndk/linker-namespace.

[2] Apple. *MacOS App Sandbox*. [Online; accessed 26-April-2022]. 2016. URL: https://developer.apple.com/library/archive/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html.

[3] Arm. *Arm Architecture Reference Manual Supplement – Morello for A-profile Architecture*. [Online; accessed 12-May-2022]. Sept. 2020. URL: https://developer.arm.com/documentation/ddi0606/ak.

[4] Arm. *Morello extensions to Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)*. [Online; accessed 13-May-2022]. Mar. 2022. URL: https://github.com/ARM-software/abi-aa/blob/60a8eb8c55e999d74dac5e368fc9d7e36e38dda4/aapcs64-morello/aapcs64-morello.rst.

[5] Arm. *Morello Program*. [Online; accessed 26-April-2022]. 2019. URL: https://www.arm.com/architecture/cpu/morello.

[6] The Kyua Authors. *Kyua*. [Online; accessed 1-May-2022]. URL: https://github.com/jmmv/kyua.

[7] The CHERI community. *CheriBSD*. [Online; accessed 26-April-2022]. 2014. URL: https://github.com/CTSRD-CHERI/cheribsd.

[8] Brooks Davis et al. "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 379–393. ISBN: 9781450362405. DOI: 10.1145/3297858.3304042. URL: https://doi.org/10.1145/3297858.3304042.

[9] Nathaniel Wesley Filardo et al. "Cornucopia: Temporal safety for CHERI heaps". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 608–625.

[10] Jean-Loup Gailly and Mark Adler. "Zlib compression library". In: (2004). URL: https://www.zlib.net.

[11] Li Gong et al. "Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2". In: *USENIX Symposium on Internet Technologies and Systems (USITS 97)*. Monterey, CA: USENIX Association, Dec. 1997. URL: https://www.usenix.org/conference/usits-97/going-beyond-sandbox-overview-new-security-architecture-java-development-kit-12.

[12] Seung-Soon Im. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. Tech. rep. TIS Committee, May 1995. URL: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[13] The NetBSD Foundation Inc., Google Inc., and Julio Merino. *Automated Testing Framework*. [Online; accessed 1-May-2022]. URL: https://github.com/jmmv/atf.

[14] Nicolas Joly, Saif ElSherei, and Saar Amar. *Security analysis of CHERI ISA*. Tech. rep. [Online; accessed 5-May-2022]. Microsoft Security Response Center (MSRC), 2020. URL: https://github.com/microsoft/MSRC-Security-Research/blob/1372d4ff97ba61886497a97c340b354896881b8f/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf.

[15] R. Levin et al. "Policy/Mechanism Separation in Hydra". In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. SOSP '75. Austin, Texas, USA: Association for Computing Machinery, 1975, pp. 132–140. ISBN: 9781450378635. DOI: 10.1145/800213.806531. URL: https://doi.org/10.1145/800213.806531.

[16] ImageMagick Studio LLC. *ImageMagick*. [Online; accessed 26-April-2022]. URL: https://imagemagick.org.

[17] Peter Loscocco and Stephen Smalley. "Integrating flexible support for security policies into the Linux operating system". In: *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. Boston, MA: USENIX Association, June 2001. URL: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/integrating-flexible-support-security-policies.

[18] Mark Samuel Miller. "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control". PhD thesis. Baltimore, Maryland, USA: Johns Hopkins University, May 2006.

[19] Kyndylan Nienhuis et al. "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1003–1020.

[20] Oracle. *Java SE7 Documentation*. [Online; accessed 26-April-2022]. 1995. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html.

[21] Jerome H. Saltzer and Michael D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[22] Guy Eric Schalnat et al. *Libpng*. [Online; accessed 26-April-2022]. URL: http://www.libpng.org/pub/png/libpng.html.

[23] The FreeBSD Project. *FreeBSD Operating System*. 1995. URL: https://www.freebsd.org.

[24] Robert N. M. Watson. "A Decade of OS Access-Control Extensibility". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 52–63. ISSN: 0001-0782. DOI: 10.1145/2408776.2408792. URL: https://doi.org/10.1145/2408776.2408792.

[25] Robert N. M. Watson, Ben Laurie, and Alex Richardson. *Assessing the viability of an open-Source CHERI desktop software ecosystem*. Tech. rep. Capabilities Limited, 2021. URL: http://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf.

[26] Robert N. M. Watson et al. *An Introduction to CHERI*. Tech. rep. UCAM-CL-TR-941. University of Cambridge, Computer Laboratory, Sept. 2019. DOI: 10.48456/tr-941. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[27] Robert N. M. Watson et al. *CHERI C/C++ Programming Guide*. Tech. rep. UCAM-CL-TR-947. University of Cambridge, Computer Laboratory, June 2020. DOI: 10.48456/tr-947. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf.

[28] Robert N. M. Watson et al. "Cheri: A hybrid capability-system architecture for scalable software compartmentalization". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 20–37.

Appendix A: Second trampoline

This is the high-level assembly code of the call-out-of-sandbox trampoline described in § 3.3.3, presented to complement the high-level code of the first trampoline (listing 3.1).

```
// Restore Executive SP (csp) and FP (c29)
cpyvalue  c29, csp, c29
cpyvalue  csp, csp, RCSP_EL0

call  push_trusted_stack_frame

// Push return address (c30)
str   c30, [c9, #16]!

// Jump to target function in sandbox
blr trusted_function

call  pop_trusted_stack_frame
// c9 stores the popped stack frame
// Restore return address
ldr c30, [c9, #16]!

// Restore Restricted SP and FP
... Calculate desired bounds
scbnds  RCSP_EL0, csp, bounds
cpyvalue  c29, RCSP_EL0, c29

// Clear caller-saved capability registers (non-argument temporaries)
mov x9, xzr
...
mov x18, xzr

// Return back into the sandbox
retr  c30
```

Appendix B: Project proposal

The original project proposal follows on the next page.

1 Description

1.1 Introduction

The CHERI architecture allows C pointers to be represented by hardware capabilities, with encoded bounds and access rights information, as well as tags, which are checked by the hardware when dereferencing a pointer. This can be used to "sandbox" code, granting it access to just the necessary code and data structures. This has the potential to prevent many kinds of common software vulnerabilities stemming from C programming mistakes being exploited, e.g., to leak confidential data, or maliciously modify data structures accessible in the address space. Data processing libraries (e.g., for compression or packet analysis) have traditionally been especially prone to these kinds of vulnerabilities, and they will be the primary focus of this project.

To date, the CHERI project focus has been on accelerating process sandboxing (using multiple processes), whereas in this project, I will implement *intra-process library sandboxing* in the run-time linker. When loading libraries into process memory, the linker will set them up to have their own protection domains, having access only to specific parts of the process address space, such as their internal data structures in the static data segment, the stack and some shared memory for arguments passing. Calls into the libraries will be implemented with CHERI domain transition instructions.

This work will extend the existing CHERI-aware run-time linker in the open-source CheriBSD operating system, using the open-source CHERI Clang/LLVM/LLD and GDB toolchain.

1.2 Evaluation

This project will identify a corpus of libraries for evaluation, with a history of vulnerabilities (e.g., data processing libraries such as libpng, giflib, libjpeg-turbo, etc.). I will analyse the impact of sandboxing on vulnerabilities violating confidentiality or integrity, e.g., using a list of past relevant vulnerabilities reported in these libraries, such as the corpus explored in the 2021 CHERI open-source desktop report [1] which white-boarded (but did not implement) library compartmentalization for the purposes of its evaluation.

Quantitative evaluation will be conducted through micro- and macro-benchmarking, measuring the performance overhead incurred by loading, and calling into these sandboxed libraries. Factors to consider are latency, memory footprint, and architectural and microarchitectural measures:

- What is the performance impact on library load and unload – and how does it scale with library complexity?
- What is the direct performance impact on function calls into and out of a library, and how does that scale with argument size (e.g., for image data buffers).
- How is system memory allocation affected by the potential duplication of code or data due to sandboxes having their own instances?
- How do tools such as Imagemagick, which directly wrap targeted libraries, behave with real-world collections of images at greater scale?

Benchmarking will be done on FPGA boards running CHERI-RISC-V, or, if opportunity and time permit, on ARM Morello boards (the first pre-production Morello boards are in use by Arm and Cambridge currently, in October 2021, and my supervisor expects production boards to be available from February 2022).

1.3 Extensions

The core project makes some simplifying assumptions, such as

- Libraries do not have their own sandboxed stacks,
- Only "leaf" libraries without any dependencies are supported,
- I am only concerned with threat models regarding integrity and confidentiality, and not availability; in particular, I don't consider recovery from faulting libraries and simply allow the application to crash – still a preferred outcome to arbitrary code execution,
- Each piece of code has only one corresponding GOT so that PC-relative addressing can be used.

As extensions, it can be investigated what happens when removing one or more of these assumptions.

It is also possible to perform analysis of trade-offs between security guarantees and restrictiveness or disruptiveness, depending on design choices, e.g., how we pass arguments.

2 Starting Point

I have a basic understanding of the CHERI project's main ideas, and of linkers on a IB Compilers level. I do not have experience with FreeBSD or systems programming. No code has been written for the project.

3 Success Criterion

In order for this project to be considered successful, the following criteria must be met:

- A run-time linker is implemented that is capable of sandboxing no-dependency libraries.
- The impact of sandboxing on software vulnerabilities is evaluated.
- Performance overhead of sandboxing is evaluated on the identified corpus of libraries.

4 Work Plan

- **Deadline** 5 October: Phase 1 Report
- **Deadline** 8 October: Draft proposal
- **Deadline** 18 October: Submit proposal
- 16–29 October: *Background reading and initial experiments*
  Do background reading, and set up the development environment.
  Milestone: Prepare a document explaining key concepts.
  Milestone: Get the CHERI-RISC-V platform up and running in QEMU.
- 30 October–12 November: *Prepare evaluation corpus*
  Gather, analyse and adjust a set of libraries, to be used for evaluation. Maybe only consider a suitable subset of their APIs to reduce dependencies (e.g., keep only APIs for image processing that use buffers, and not files).
  Milestone: A list of libraries is identified for evaluation.

- 13–26 November: *Design core deliverable*
  Develop the model for linker-imposed sandboxing, describing the approach and constraints.
  Milestone: Prepare a design document about the approach.
- 27 November–10 December: *Implement core deliverable*
  Implement the model designed above.
  Milestone: Run-time linker implemented.
- 11–24 December: *Two week break for Christmas*
- 25 December–7 January: *Core project evaluation*
  Evaluate the implementation on the identified corpus of libraries.
  Milestone: Benchmarking data measured on FPGA.
- 8–21 January: *Complete core project*
- 22 January–4 February: *Revise evaluation based on feedback, write Progress Report*
  Deadline 4 February: Progress Report
- 5–18 February: *Prepare presentation, slack time or start extension*
  Deadline 10, 11, 14, 15 February: Progress Report Presentations
- 19 February–4 March: *Complete extension*
- 5–18 March: *Start writing up*
  Milestone: Submit the first chapter to the supervisor.
- 19 March–1 April: *Continue writing up*
  Milestone: Submit the second chapter to the supervisor.
- 2–15 April: *Complete writing up and deliver draft dissertation to supervisor*
  Milestone: Submit the rest of the chapters to the supervisor.
- 16–29 April: *Two week break while supervisor reads draft*
- 30 April–13 May: *Revise dissertation and prepare for submission*
  Deadline: Dissertation submission

5 Resource Declaration

My main development environment is going to be my personal laptop (MacBook Air 2018, 8GB RAM, Intel i5 CPU). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure, such as using Git version control, regularly creating back-ups to GitHub and iCloud drive. As a fallback option, my supervisor will provide a replacement laptop.

If needed, my supervisor will grant access to a build server, but that does not count as a critical resource.

For performing benchmarking and evaluation, access to an FPGA will be required to run the CHERI enabled architecture. My supervisor will provide either physical access to a board or AWS access, to be billed to a grant. Evaluation will potentially be performed on ARM Morello boards instead, but they do not count as a critical resource.

Although the CHERI software stack that this project builds on is still in the research phase, it is widely used in academia and industry, giving confidence that it will be a reliable baseline for my project, and the large research team will be able to provide support if needed.

References

[1] Watson, Robert N. M., et al. *Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem.* Version 1, 17 September 2021. Capabilities Limited CHERI Desktop Report, `http://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf`. Accessed 14 October 2021.