CerisEmu: A capability machine emulator

Group 10
Philipp Haas **Léo Gamboa dos Santos**

May 31, 2024

Abstract

This report outlines the design and implementation of an emulator for a capability-based machine, inspired by the Cerise specification. It aims to achieve fine-grained memory protection and software compartmentalization through hardware capabilities. The project, implemented in Rust, consists of a compiler and an emulator.

A proof-of-concept operating system, running on the emulator, demonstrating safe compartmentalization and execution of untrusted code. The OS kernel initializes the system, sets up interrupts, and runs user programs in isolation. The project includes a comprehensive suite of unit and integration tests to ensure reliability and correctness.

Contents

1 Introduction

Memory safety and secure software compartmentalization are critical aspects of modern computer systems, particularly in the context of increasing complexity and security threats. Hardware capabilities offer a promising approach to achieving these goals, by associating pointers with specific memory ranges and permissions. This technique has evolved over the years, with CHERI (Capability Hardware Enhanced RISC Instructions)[Woodruff, 2014] being a recent development that provides a robust framework for fine-grained memory protection.

This report outlines the design and implementation of an emulator for a capability-based machine inspired by the Cerise specification. Cerise [Georges* et al., 2024] is a formally-verified subset of the CHERI capability machine. Our emulator is designed to follow the Cerise specifications closely, while implementing modifications that ensure the unforgeability of capabilities via RSA cryptographic signing. This method ensures that user programs cannot create or alter capabilities arbitrarily, thereby upholding the security gurantees established by the formal proofs of Cerise.

The emulator is implemented in Rust, because of its powerful type-checking capabilities, which serve to prevent many common programming errors and improve overall safety. The project is divided into two main components: the compiler and the emulator. The compiler translates assembly code into a format understandable by the emulator, while the emulator executes the translated program within a simulated capability-based environment.

Additionally, we have developed a proof-of-concept operating system to showcase the practical uses of our emulator. Essential routines such as memory copying (memcpy) and dynamic memory allocation (malloc) are integrated into the operating system, complemented by a micro-kernel engineered to safely execute user programs in isolated environments. During the development process, we rigorously tested our implementation to guarantee its correctness and reliability.

The remaining sections of this report are structured as follows. Section 2 details the research upon which our implementation is based. In section 3, we discuss the design choices that we adopted. In section 4 we provide an in-depth overview about the implementation specifications of our project, while section 5 elaborates on the approach taken to implement our proof-of-concept operating system. Included are details about the example programs memcpy and malloc, along with specifics about the kernel implementation. In section 6, we summarize our report and draw a conclusion from our findings.

2 Related work

2.1 Hardware Capabilities

Hardware Capabilities are a technique to ensure memory safety. They were first introduced in 1966 by Dennis and Van Horn [1966] and have been developed since then, with the last update of *CHERI* coming out in 2019 [Watson et al., 2019]. They offer a way to implement fine-grained memory protection as well as scalable software compartmentalization. They essentially follow the security technique of least privilege. The way Hardware Capabilities achieve memory safety, through least privilege, is through adding a range and a permission to a pointer. That means the pointer is only allowed to access memory within the given range and only can perform actions on the memory it has the permission for. Furthermore, it is only possible to make the capabilities more restrictive, by for instance reducing the given permission or shrinking the bounds of the memory it is able to access. This removes the problem of, for example, dereferencing null pointers or pointers that point out of bounds.

2.2 Cerise

Cerise [Georges* et al., 2024] is a formally-verified capability machine specification based on a subset of the *CHERI* [Woodruff, 2014] capability machine. The emulator designed in this project aims to closely emulate the specifications of the Cerise capability machine.

Some changes have been made to these specifications, to either extend them or to better fit them into our implementation of the emulator. However, no major changes have been made, as these could potentially introduce unforeseen security flaws which the formal proofs of Cerise wouldn't cover anymore.

3 Design

The following section describes the design choices we have made, as well as the specifications that we created and followed when implementing our project.

3.1 Unforgeable capabilities

The model of our capabilities closely follows the one from Cerise, as evident in definition 1.

$$
\begin{array}{lll}
\text{Capability} : (p, b, e, a) & & \\
\text{where} \quad p \in \{\text{O}, \text{E}, \text{RO}, \text{RX}, \text{RW}, \text{RWX}\} & & \text{permission} \\
\quad\quad b \in \text{Address} & & \text{base address} \\
\quad\quad e \in \text{Address} & & \text{end address} \\
\quad\quad a \in \text{Address} & & \text{current address}
\end{array} \tag{1}
$$

One of the conditions for Hardware Capabilities to be secure, is that they need to be unforgeable. If this cannot be ensured, every process could create their own capabilities, making the security guarantees obsolete. In Cerise, this unforgeability is guaranteed by the specifications, as no instructions exist to create new capabilities. In our emulated machine, however, it is technically possible to represent capabilities inside arbitrary programs or in the memory of the machine. While the compiler and specifications *should* still disallow new capabilities from being created, we opted to add a cryptographic signing approach to ensure capabilities are unforgeable.

To this extent, we define `SignedCapability`, where each capability is wrapped in a signed type, with the additional attribute `signature` (definition 2).

$$
\begin{array}{lll}
\text{SignedCapability} : (c, s) & & \\
\text{where} \quad s \in \mathbb{Z} & & \text{signature} \\
\quad\quad c \in \text{Capability} & & \text{capability}
\end{array} \tag{2}
$$

To sign and verify the capabilities, we have decided to use RSA [Rivest et al., 1978]. The key to sign and verify the capabilities is created at the time of the machine creation 3.2.

To sign and verify capabilities, the functions `sign(`c`)` and `verify(`(c, s)`)` are added to the machine. In `sign`, the machine uses its signing key `sk` in combination with the data inside the capability c, to append a signature s to the capability and return a signed capability (c, s). In `verify`, the signature s is verified against the data of the capability c using the verifying key `vk`, and if they match, the inner capability c is returned.

If at any point a capability cannot be verified during execution, the machine fails it current operation.

3.2 Machine design

Just like in Cerise, our machine specifications abstract away from the hardware bits and represent all definitions in an abstractly-typed way. We however also remove the need to encode capabilities and instructions as integers, and instead make them fully typed like our other definitions.

Our machine consists of a number of different attributes to make it work. As most conventional computers and similarly to Cerise, it has an execution state, registers, an interrupt table and memory. In addition to that, our machine needs a signing key and a verifying key to ensure that the capabilities are unforgeable, as detailed in section 3.1.

$$
\begin{array}{lll}
\text{Machine } \varphi : \quad (\mathit{execState},\ \mathit{reg},\ \mathit{mem},\ \mathit{itrTable},\ \mathit{sk},\ \mathit{vk}) & & \\
\text{where} \quad \mathit{execState} \in \text{State} & & \text{current state} \\
\quad\quad \mathit{reg} \in \text{Registers} & & \text{registers} \\
\quad\quad \mathit{mem} \in \text{Memory} & & \text{memory} \\
\quad\quad \mathit{itrTable} \in \text{InterruptTable} & & \text{interrupt table} \\
\quad\quad \mathit{sk} \in \text{RsaPrivateKey} & & \text{signing key} \\
\quad\quad \mathit{vk} \in \text{RsaPublicKey} & & \text{verifying key}
\end{array}
$$

$$
\begin{array}{rcl}
s \in \text{State} & ::= & \text{Running} \mid \text{Halted} \mid \text{Failed} \mid \text{Interrupted}(itr \in \text{Interrupt}) \\
reg \in \text{Registers} & \triangleq & \text{RegName} \to \text{Word} \\
mem \in \text{Memory} & \triangleq & \text{Address} \to \text{Row} \\
itrTable \in \text{InterruptTable} & \triangleq & \text{Interrupt} \to \text{Address} \\
itr \in \text{Interrupt} & ::= & \text{Halt} \mid \text{Fail} \\
row \in \text{Row} & \triangleq & \text{Instruction} \cup \text{Word} \\
w \in \text{Word} & \triangleq & \mathbb{Z} \cup \text{Char} \cup \text{SignedCapability} \cup \text{Permission} \\
sc \in \text{SignedCapability} & \triangleq & \{(c, s) \mid c \in \text{Capability} \wedge s \in \mathbb{Z}\} \\
c \in \text{Capability} & \triangleq & \{(p, b, e, a) \mid p \in \text{Permission} \wedge b, e, a \in \text{Address}\} \\
p \in \text{Permission} & ::= & \text{O} \mid \text{E} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX} \\
a \in \text{Address} & \triangleq & [0, \text{AddrMax}) \\
r \in \text{RegName} & ::= & \text{PC} \mid \text{R}_0 \mid \text{R}_1 \mid \ldots \mid \text{R}_{255} \\
i \in \text{Instruction} & ::= & \text{jmp } r \mid \text{jnz } r\ r \mid \text{mov } r\ \rho \mid \text{load } r\ r \mid \text{sto re } r\ \rho \mid \text{add } r\ \rho\ \rho \mid \text{sub } r\ \rho\ \rho \mid \\
 & & \text{lt } r\ \rho\ \rho \mid \text{lea } r\ \rho \mid \text{restrict } r\ p' \mid \text{subseg } r\ \rho\ \rho \mid \text{isptr } r\ r \mid \text{getp } r\ r \mid \\
 & & \text{getb } r\ r \mid \text{gete } r\ r \mid \text{geta } r\ r \mid \text{fail} \mid \text{halt} \\
 & & \text{where} \quad r \in \text{RegName} \wedge \rho \in (\text{RegName} \cup \text{Word}) \wedge p' \in \text{Permission}
\end{array}
$$

A note on vocabulary: In our specification and the rest of the report, we distinguish between *rows* of memory and *words*. A `Row` represents the smallest unit in the memory of our machine, and can be either an `Instruction` or a `Word`. Meanwhile, a `Word` represent a data value, and can be either an integer, a single character, a (signed) capability or a permission.

3.3 Execution

The execution specification of our capability machine is quite similar to the one defined for Cerise. We define here the new `execMachine`(φ) function, which, given a new capability machine φ, emulates it until possible termination.
We mark some minor changes to the specification from Cerise as well as new addition in *teal*. Changes related to our capability signing approach are marked in *red*.

$$\text{execMachine}(\varphi) = \text{loop}(\text{execSingle}(\varphi[\text{reg.PC} \mapsto \text{sign}((\top, 0, \text{AddrMax}, 0)),\ \text{execState} \mapsto \text{Running}]))$$

$$
\text{loop}((s, \varphi)) = \begin{cases}
\text{loop}(\text{execSingle}(\varphi[\text{execState} \mapsto \text{Running}])) & \text{if } s = \text{Running} \\
\text{loop}(\text{execSingle}(\varphi[\text{execState} \mapsto \text{Interrupted(Halt)}, \\ \quad \text{reg.PC} \mapsto \text{updatePcPerm}(w)])) & \text{if } s = \text{Halted} \wedge \varphi.\text{execState} \neq \text{Interrupted}(\_) \wedge \\ & \quad w = \varphi.\text{mem}(\varphi.\text{itrTable(Halt)}) \\
\text{loop}(\text{execSingle}(\varphi[\text{execState} \mapsto \text{Interrupted(Fail)}, \\ \quad \text{reg.PC} \mapsto \text{updatePcPerm}(w)])) & \text{if } s = \text{Failed} \wedge \varphi.\text{execState} \neq \text{Interrupted}(\_) \wedge \\ & \quad w = \varphi.\text{mem}(\varphi.\text{itrTable(Fail)}) \\
\varphi[\text{execState} \mapsto originalState] & \text{if } s \in \{\text{Halted} \mid \text{Failed}\} \wedge \\ & \quad \varphi.\text{execState} = \text{Interrupted}(originalState) \\
\varphi & \text{otherwise}
\end{cases}
$$

$$
\text{execSingle}(\varphi) = \begin{cases}
\text{execInstruction}(\varphi, i) & \text{if } \text{verify}(\varphi.\text{reg(PC)}) = (p, b, e, a) \wedge b \leq a < e \wedge \\ & \quad p \succeq \text{RX} \wedge \varphi.\text{mem}(a) = i \wedge i \in \text{Instruction} \\
(\text{Failed}, \varphi) & \text{otherwise}
\end{cases}
$$

$$
\text{getWord}(\varphi, \rho) = \begin{cases}
\rho & \text{if} \rho \in \text{Word} \\
\varphi.\text{reg}(\rho) & \text{if} \rho \in \text{Register}
\end{cases}
$$

$$
\text{updPC}(\varphi) = \begin{cases}
(\text{Running}, \varphi[\text{reg.PC} \mapsto \text{sign}((p, b, e, a + 1))]) & \text{if } \text{verify}(\varphi.\text{reg(PC)}) = (p, b, e, a) \\
(\text{Failed}, \varphi) & \text{otherwise}
\end{cases}
$$

$$
\text{updatePcPerm}(w) = \begin{cases}
\text{sign}((\text{RX}, b, e, a)) & \text{if } w = \text{verify}((\text{E}, b, e, a)) \\
w & \text{otherwise}
\end{cases}
$$

Instruction	Conditions	New State and Effects
i		execInstruction(φ, i)
fail		(Failed, φ)
halt		(Halted, φ)
mov r ρ	$w = \text{getWord}(\varphi, \rho)$	updPC(φ[reg.r $\mapsto$ w])
load r_1 r_2	verify($\varphi.reg(r_2)$) $= (p, b, e, a)$ $\wedge$ $p \succeq \text{RO} \wedge$ $b \leq a < e$ $\wedge$ $w = \varphi.mem(a)$	updPC(φ[reg.r$_1$ $\mapsto$ w])
store r ρ	verify($\varphi.reg(r)$) $= (p, b, e, a)$ $\wedge$ $p \succeq \text{RW} \wedge$ $b \leq a < e$ $\wedge$ $w = \text{getWord}(\varphi, \rho)$	updPC(φ[mem.a $\mapsto$ w])
jmp r	$newPc = \text{updatePcPerm}(\varphi.reg(r))$	(Running, φ[reg.PC $\mapsto$ $newPc$])
jnz r_1 r_2	$newPc = \text{updatePcPerm}(\varphi.reg(r_1))$	if $\varphi.reg(r_2) \neq 0$ then (Running, φ[reg.PC $\mapsto$ $newPc$] else updPC(φ))
restrict r p'	verify($\varphi.reg(r_2)$) $= (p, b, e, a) \wedge$ $p' \preceq p$ $\wedge$ $w = \text{sign}((p', b, e, a))$	updPC(φ[reg.r $\mapsto$ w])
subseg r ρ_1 ρ_2	verify($\varphi.reg(r_2)$) $= (p, b, e, a)$ $\wedge$ $z_1 = \text{getWord}(\varphi, \rho_1) \wedge$ $z_2 = \text{getWord}(\varphi, \rho_2)$ $\wedge$ $b \leq z_1 < \text{AddrMax} \wedge$ $0 \leq z_2 \leq e$ $\wedge$ $p \neq \text{E}$ $\wedge$ $w = \text{sign}((\rho, z_1, z_2, a))$	updPC(φ[reg.r $\mapsto$ w])
lea r ρ	verify($\varphi.reg(r)$) $= (p, b, e, a)$ $\wedge$ $z = \text{getWord}(\varphi, \rho) \wedge$ $p \neq \text{E}$ $\wedge$ $w = \text{sign}((\rho, b, e, a + z))$	updPC(φ[reg.r $\mapsto$ w])
add r ρ_1 ρ_2	$z_1 = \text{getWord}(\varphi, \rho_1)$ $\wedge$ $z_2 = \text{getWord}(\varphi, \rho_2) \wedge$ $z_1 \in \mathbb{Z}$ $\wedge$ $z_2 \in \mathbb{Z}$ $\wedge$ $z = z_1 + z_2$	updPC(φ[reg.r $\mapsto$ z])
sub r ρ_1 ρ_2	$z_1 = \text{getWord}(\varphi, \rho_1)$ $\wedge$ $z_2 = \text{getWord}(\varphi, \rho_2) \wedge$ $z_1 \in \mathbb{Z}$ $\wedge$ $z_2 \in \mathbb{Z}$ $\wedge$ $z = z_1 - z_2$	updPC(φ[reg.r $\mapsto$ z])
lt r ρ_1 ρ_2	$z_1 = \text{getWord}(\varphi, \rho_1)$ $\wedge$ $z_2 = \text{getWord}(\varphi, \rho_2) \wedge$ $z_1 \in \mathbb{Z}$ $\wedge$ $z_2 \in \mathbb{Z}$ $\wedge$ if $z_1 < z_2$ then $z = 1$ else $z = 0$	updPC(φ[reg.r $\mapsto$ z])
getp r_1 r_2	verify($\varphi.reg(r_2)$) $= (p, \_, \_, \_)$	updPC(φ[reg.r $\mapsto$ p])
getb r_1 r_2	verify($\varphi.reg(r_2)$) $= (\_, b, \_, \_)$	updPC(φ[reg.r $\mapsto$ b])
gete r_1 r_2	verify($\varphi.reg(r_2)$) $= (\_, \_, e, \_)$	updPC(φ[reg.r $\mapsto$ e])
geta r_1 r_2	verify($\varphi.reg(r_2)$) $= (\_, \_, \_, a)$	updPC(φ[reg.r $\mapsto$ a])
isptr r_1 r_2	if $\varphi.reg(r_2) \in Capability$ then $z = 1$ else $z = 0$	updPC(φ[reg.r $\mapsto$ z])
_	otherwise	(Failed, φ)

3.4 Interrupts

To be able to manage interrupts effectively, we employ an *Interrupt Table*. The instantiation of the interrupt table occurs at machine initialization, assigning each interrupt to its corresponding redirection address. The implemented interrupts consist of `Halt` and `Fail`. Should either of the corresponding instructions be executed or if the machine encounters a failure, the machine attempts to recover by setting its current state to `Interrupted` and setting the `PC` register to the value at the address specified in the interrupt table. If it fails to recover, the machine is promptly terminated.

The interrupt mechanism is useful for programs to recover and managing errors from subroutines, e.g. an OS handling errors during the execution of untrusted program code. A more detailed explanation of the technique can be found in section 5.

4 Implementation

In the following section, we outline the implementation details of our project. We opted for Rust as our programming language due to its strong type checking features.

Throughout the implementation, we serialize and de-serialize our internal types, powered by the *serde* library [Open-source contributors, 2024]. The target format for this is *Rusty Object Notation* (RON), a format inspired by JSON but adapted for Rust.

The implementation is divided into two main components: the *compiler* (section 4.1) and the *emulator* (section 4.3). We also go over the configuration mechanism in section 4.2, and the *testing* in section 4.4.

4.1 The Compiler

The first implementation step involved translating the input code into a format comprehensible to our emulator.

Since we opted to adhere closely to Cerise for the instruction set, we decided to make our assembly language faithful to the one used in the code listings in the Cerise paper. Moreover, most code listings from the Cerise paper should be compilable with our compiler.

The compiler pipeline is divided into four stages; lexing, parsing, processing and code generation, as shown in figure 1.

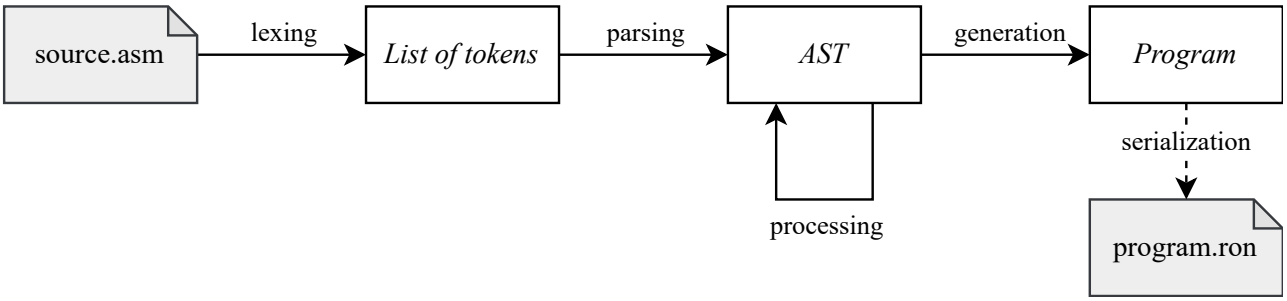

Figure 1: The different stages of the compiler.

- The lexing step involves tokenizing the input assembly code via a lexer. To simplify this task, the *Logos* [Hirsz, 2024] library was used. Every token is defined using a string or a regex expression, and Logos creates the appropriate lexer which creates a list of tokens from the input assembly.
- The parsing step involves parsing the list of tokens into an *Abstract Syntax Tree* (AST). A simple recursive descent parser is used for parsing, and the resulting AST is of type `Ast`, which is essentially a list of `AstRows`.
- The processing step applies multiple transformations to the AST, which brings it closer to the final representation of the compiled program. The transformations are: desugaring strings into individual characters, desugaring `GOTO` statements into `lea` instructions, extracting the relative position of labels definitions (i.e. init: ...) and finally evaluating math expressions inside brackets `[ ]`.
- Finally the generation step maps the AST to a compiled `Program`, while making sure the AST is properly processed.

All these stages are combined and carried out sequentially within the compiler part of the project.

4.2 Configuring a machine for emulation

To facilitate creating arbitrary machines for different purposes and to allow complex integration testing (see section 4.4), we have implemented a *Machine Configuration* system. The idea is that a `MachineConfig` can be created from scratch or serialized from a RON file, and a proper `Machine` could be built from it, ready to be emulated/executed.

A `MachineConfig` requires at least a `size`, which will set the memory size of the `Machine`. One can also specify `programs`, each of which will load a given `ProgramConfig` at the specified address; `registers`, which initialize each given register with the given value; and an `interrupt_table`, which specifies the addresses for each interrupt type.

`ProgramConfig`s are an additional utility, which simplifies specifying a program in different forms (as a file, raw string, uncompiled, compiled, serialized, etc.). When building the `Machine` from a `MachineConfig`, each `ProgramConfig` will automatically load, de-serialize and/or compile any file or source where necessary.

Figure 2 illustrates how a `MachineConfig` can be created in multiple ways.

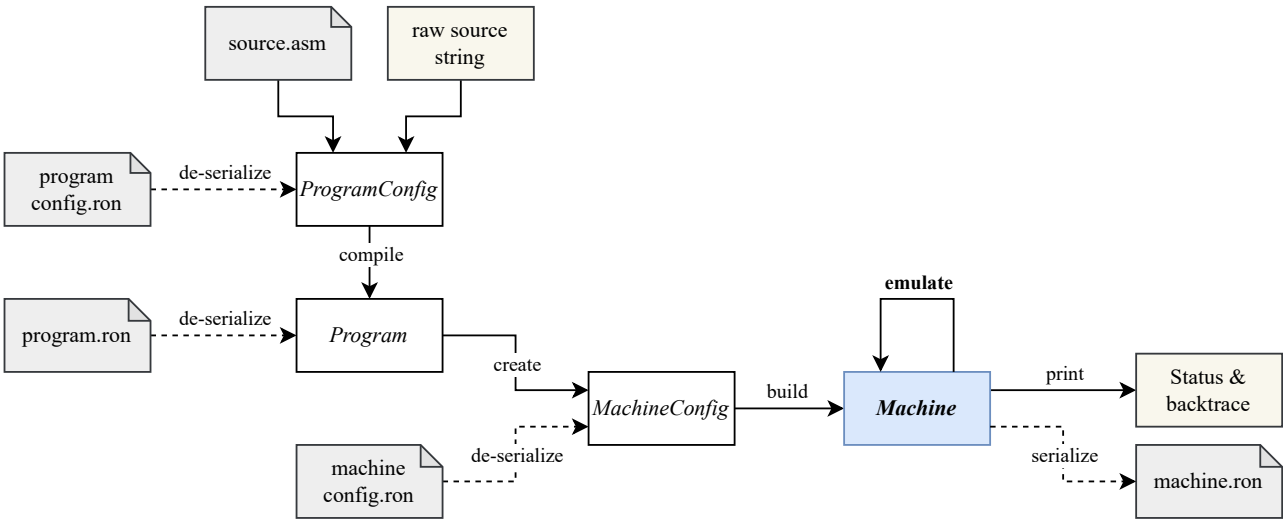

Figure 2: The different stages of the emulator, and how to build a machine to emulate.

4.3 The Emulator

The emulator is the main part of our project; it is capable of emulating capability machine programs as defined in Cerise. As the emulator follows closely the specifications laid out in sections 3.2 and 3.3, the code of the emulator is straightforward to understand and requires little description.

There is a single difference between the implementation of the emulator and the execution specs from section 3.3. Instead of passing the machine (φ) around to static functions, all the execution functions and utility functions are implemented as methods of the `Machine` type, properly encapsulating them.

Here, the rust type-system combined with the way we defined the `Machine` type and subtypes forces a lot of pattern matching. However, it also gives us excellent type safety, making runtime bugs in the emulator code less likely. It also ensures we are never using unverified capabilities during execution, as the inner capability of a `Signed<Capability>` is private and is only accessible through verification.

To help with debugging the emulated programs, we also added a *backtrace* system to track the execution of the machine. At every action or error in the emulator, a corresponding message is added to the machine's backtrace. After execution, the backtrace can be formatted into a neat table, greatly assisting in the finding of bugs in our assembly code. Listing 7 shows such a backtrace.

4.4 Testing

To ensure our implementation creates predictable results and hopefully contains few bugs, we have created a suit of unit and integration tests, where we test each part of the implementation individually.

For the compiler, the tests are mainly unit tests about math expressions and that they produce the correct results.

For the emulator, we have made some unit tests checking the correctness of the permission lattice, as well as some complete integration tests checking the assembly implementations of malloc (5.3) and memcpy (5.2). In addition, we created isolated test cases for most instructions.

In total, there are 70 test cases for the project. We would have liked to have even better test coverage for this project, however due to time constraints we had to focus on the most important and error-prone parts of the code.

The integration tests for malloc, memcpy and the individual instructions were the original reason for creating `MachineConfig`, as it made it much simpler to create reliable and predictable tests for the emulator.

5 Capability machine-based OS

An additional objective of the project is to create a proof-of-concept Operating System, written in our capability machine assembly language, and running in our emulator. As part of this OS, we have written a `memcpy` 5.2 and a `malloc`[1] 5.3 subroutine, as well as a micro-`kernel` 5.4 capable of safely running user programs. The code for all these can be read in the appendix, section 7.1.

The techniques used for calling untrusted code and memory compartmentalization are laid out by the Cerise paper [see Georges* et al., 2024, section 2], and were used extensively in the implementation of this OS proof-of-concept.

5.1 Machine configuration for the OS

The machine to emulate the OS is set up using a `MachineConfig`, the serialized representation for which can be read in listing 1. The machine is set to have a memory size of 65536 words, and the interrupts are set at the very end of the memory; `0xFFFE` and `0xFFFF` for the `Fail` and `Halt` interrupts respectively. The kernel is loaded at the beginning of memory, and the routines `memcpy`, `malloc` and `hello_world` are loaded at addresses `0x0100`, `0x0200` and `0x0300` respectively.

5.2 memcpy

`memcpy` is a short subroutine that copies memory in bulk. For that it needs two capabilities, with the source one needing read permission, while the destination needs write permissions. Furthermore, the range of both permissions need to be the same size. The subroutine makes sure to check these conditions and failing if necessary. If all checks are successful, the subroutine enters a loop. At each iteration, a single word is loaded from the source into a buffer register, and then stored at the destination. When all words are copied, the loop ends and the subroutine transfers control back to the caller.

5.3 malloc

`malloc` is a subroutine that allocates heap memory on demand, and returns a capability to the newly-allocated memory section. As this subroutine require proper compartmentalization, it is split up in two parts; the `init` subroutine and the actual `malloc` subroutine. `init` should be called by the system to initialize the memory for the heap. It expects a PC with write access, as well as a capability with write access that delimits the memory to be used for the heap. `init` then stores the heap pointer in its local program memory at `[heap]`, and creates

[1] The Cerise paper already proposes an implementation for malloc, but we decided to reimplement our own.

a read-write pointer to that local storage, which is then stored at `[heap_access]`. Finally, a pointer to the `malloc` subroutine is created and returned.

When `malloc` is called, all it has to do is use its read-only PC to read the pointer at `[heap_access]`. This gives the subroutine the priviledge of reading and writing the heap pointer. The heap pointer is then incremented and stored back into memory, while a copy of it is sub-segmented to the right size and finally returned to the caller.

5.4 The kernel

In this proof-of-concept OS, the kernel is responsible for setting up the standard library (memcpy and malloc), setting up interrupts, handle errors and run programs in isolation.

To do this, it first goes through an os-initialization phase. During that phase, its first actions are to save the `PC` register (which contains the *Master Capability* at the beginning), and modifying it to create a pointer to the `internal_err` section which is then set as the `FAIL` interrupt. This is done as early as possible in the initialization phase to make sure it can catch as many potential internal errors as possible. Next, the other interrupt pointers are created, and stored to local memory. The next action during the initialization phase is to create pointers to `memcpy` and `malloc`. Malloc additionally requires its `init` subroutine to be called first, so that is also done before creating the pointer to `malloc`. Finally, at the end of the initialization phase, the PC register is downgrade from `RWX` to `RX` (thus removing write-access), the registers are cleared of their values, and the kernel can now start a program.

To start a program given its address in memory, the kernel needs to create an appropriate sentry capability with the right bounds and the `E` permission, load the pre-computed pointers to `memcpy` and `malloc` into their registers, and set the proper interrupt addresses in case the program were to halt or fail.

When one of these interrupts is triggered, the kernel gains back execution at the respective section; potentially `program_fail` or `program_halt`. There, the kernel sets an *exit code* in register `R0`, unbinds the `HALT` interrupt to avoid an infinite loop, and finally halts the machine.

6 Conclusion

In conclusion, this project has successfully realized an emulator for a capability-based machine, inspired by Cerise, providing fine-grained memory protection and secure software compartmentalization. By closely following the Cerise specification, and additionally introducing mechanisms for cryptographic signing of capabilities, our emulator ensures the unforgeability and integrity of hardware capabilities, thereby reinforcing the security guarantees provided by hardware capabilities.

The decision to use Rust as the programming language for implementation proved advantageous, utilizing its strong type-checking features to mitigate common programming errors and enhance overall code safety. In conclusion, we have implemented a robust emulator capable of running assembly code within a simulated capability-based environment, accurately reproducing the behavior outlined by Cerise.

Furthermore, the creation of a proof-of-concept operating system shows the practical use-cases of our emulator. It showcases its ability to support critical operations, while maintaining strong security barriers between user programs.

Throughout the project, we extensively tested our implementation to ensure its correctness and reliability. Unit tests, integration tests, and comprehensive test suites were employed in identifying and resolving potential issues, leading to a more robust and stable emulator.

Looking forward, future work could focus on additional optimization of the emulator to enhance performance, as well as exploring additional security features and capabilities to increase the overall security of the system. In conclusion, this project makes a step towards realizing the vision of secure and trustworthy computing environments through hardware capabilities.

References

J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, mar 1966. ISSN 0001-0782. doi: 10.1145/365230.365252. URL `https://doi.org/10.1145/365230.365252`.

A. L. Georges*, A. Guéneau*, T. Van Strydonck, A. Timany, A. Trieu*, D. Devriese, and L. Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. *J. ACM*, 71(1), feb 2024. ISSN 0004-5411. doi: 10.1145/3623510. URL `https://doi.org/10.1145/3623510`.

M. Hirsz. Logos. `https://github.com/maciejhirsz/logos`, 2024.

Open-source contributors. serde. `https://github.com/serde-rs/serde`, 2024.

R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978. ISSN 0001-0782. doi: 10.1145/359340.359342. URL `https://doi.org/10.1145/359340.359342`.

R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, June 2019. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf`.

J. D. Woodruff. CHERI: A RISC capability machine for practical memory safety. Technical Report UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014. URL `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf`.

7 Appendix

7.1 Operating System code

```
MachineConfig(
        size: 0x10000,
        programs: {
                // OS Kernel
                0x0000: SourceFile("asm/kernel.asm"),

                // Standard library
                0x0100: SourceFile("asm/memcpy.asm"),
                0x0200: SourceFile("asm/malloc.asm"),

                // Actual program
                0x0300: SourceFile("asm/hello_world.asm"),
        },
        interrupt_table: {
                Fail: 0xFFFE,
                Halt: 0xFFFF,
        },
)
```

Listing 1: os.ron

```
; PROGRAM LENGTH: 33 rows

; r0: return
; r1: source memory (RO, a, b, a)
; r2: destination memory (RW, c, d, c)

memcpy:
   ; make sure that address of r1 is =base
   geta r3 r1
   getb r4 r1
   sub r3 r4 r3 ; if address is bigger than base by x, now r3=-x
   lea r1 r3 ; done

   ; make sure that address of r2 is =base
   geta r3 r2
   getb r4 r2
   sub r3 r4 r3 ; if address is bigger than base by x, now r3=-x
   lea r2 r3 ; done

   ; check that the size of the memory zones r1 and r2 are the same
   getb r3 r1
   gete r4 r1
   sub r3 r4 r3 ; r3=end-base
   getb r4 r2
   gete r5 r2
   sub r4 r5 r4 ; r4=end-base
   sub r3 r4 r3 ; r3 should now be zero. if not, fail
      mov r4 PC
      lea r4 4
      jnz r4 r3
   GOTO ok
      fail
   ok:


   ; r3 will be our buffer
   loop:
      ; copy source into the buffer
      load r3 r1
      ; copy buffer into destination
      store r2 r3
      ; i++
      lea r1 1
      lea r2 1

      ; check if the loop should continue
      gete r4 r1
      geta r5 r1
      sub r4 r5 r4

      ; if r4 is zero, we break the loop
      mov r5 PC
      lea r5 4
      jnz r5 r4
      GOTO break
      GOTO loop
   break:

   ; we're done!
   jmp r0
```

Listing 2: memcpy.asm

```
; PROGRAM LENGTH: 39 rows

; PC: (RWX, ...)
; r0: integer that will be used to return to the caller routine
; r1: (RWX, low, high, pointer)
; returns r1: pointer to malloc subroutine
init:
   mov r2 PC ; important: expects PC to be RWX at least
   lea r2 [heap - init]
   store r2 r1 ; store at [heap]
   mov r3 r2
   lea r3 [-1]
   store r3 r2 ; store at [heap_access]

   ; return the pointer for malloc
   marker0:
   mov r1 PC
   lea r1 [malloc - marker0]

   ; clear registers just in case
   mov r2 0
   mov r3 0

   ; Jump back to caller
   marker1:
   sub r0 r0 [marker1 - init + 1]
   lea PC r0


heap_access:
   empty ; Will contain (RWX, _, _, heap)
heap:
   empty ; Will contain (RWX, low, high, pointer)



; PC: (RX, ...)
; r0: return
; r1: integer determining the number of words to allocate
;
; returns r1: capability to the allocated memory
;
; fails if size <= 0 or if it does not have enough space left
malloc:
   ; get access to the heap
   mov r4 PC
   lea r4 [-2]
   load r4 r4 ; r4 = heap_access: (RWX, _, _, heap)
   load r5 r4 ; r5 = heap: (RWX, low, high, pointer)

   ; check that size > 0
   lt r3 0 r1 ; if everything is correct, r3 = 1
   mov r2 pc
   lea r2 4
   jnz r2 r3 ; if r3 = 0, that means it's bad, so don't jump ahead and fail
   fail

   mov r2 r1 ; r2 = number of words to allocate

   geta r7 r5 ; r7 = future base of the return capability

   mov r6 r5 ; r6 = copy of (RWX, low, high, pointer)
   lea r6 r2 ; increment the heap capability
   store r4 r6 ; store the heap capability back at [heap]

   geta r8 r6 ; r8 = future end of the return capability
   subseg r5 r7 r8

   ; copy to r1, we're done
   mov r1 r5

   ; clear registers just in case
   mov r2 0
   mov r3 0
   mov r4 0
   mov r5 0
   mov r6 0
   mov r7 0
   mov r8 0

   ; return
   jmp r0
```

Listing 3: malloc.asm

```
; The kernel returns an exit code in r0 after terminating.
;   0: everything went well and the program halted correctly
;   1: the program failed
;   2: the kernel encountered an unexpected internal error

; The machine starts out with PC = (RWX, 0, MAX, 0) (the master capability)

os_init:
   ; R255 = MASTER CAPABILITY
   mov R255 PC

   ; Save master capability to memory
   mov R1 R255
   lea R1 [master]
   store R1 R255

   ; Setup interrupt pointers
   ; Setup internal_err
   mov R2 R255
   lea R2 [internal_err]
   mov R1 R255
   lea R1 [p_internal_err]
   store R1 R2

   ; Set internal_err as the FAIL interrupt
   mov R1 R255
   lea R1 0xFFFE
   store R1 R2

   ; Setup the rest of interrupt pointers
   ; Setup program_fail
   mov R2 R255
   lea R2 [program_fail]
   mov R1 R255
   lea R1 [p_program_fail]
   store R1 R2

   ; Setup program_halt
   mov R2 R255
   lea R2 [program_halt]
   mov R1 R255
   lea R1 [p_program_halt]
   store R1 R2

   ; Setup p_invalid
   mov R2 R255
   subseg R2 0x0 0x0
   restrict R2 0
   mov R1 R255
   lea R1 [p_invalid]
   store R1 R2



   ; Setup memcpy
      ; Create pointer to memcpy()
      mov R2 R255
      lea R2 0x100 ; memcpy() is at 0x100
      subseg R2 0x100 0x200
      ; Save pointer to memcpy()
      mov R1 R255
      lea R1 [p_memcpy]
      store R1 R2

   ; Setup malloc
      ; Setup args for init()
         ; r0: integer that will be used to return to the caller routine
         mov R0 [-(0x200 - malloc_init_return) - 1] ; init() is at 0x200
         ; r1: (RWX, low, high, pointer)
         mov R1 R255
         lea R1 0x1000
         subseg R1 0x1000 0xF000 ; heap will be in the range [0x1000..0xF000[
      ; Call init()
      ; (A lea-jump is used instead of a jmp so that PC stays RWX instead of E->RX, and same to return
          back from init())
      malloc_init_call:
         lea PC [0x200 - malloc_init_call - 1]
      malloc_init_return:
      ; Fix up the pointer to malloc
      subseg R1 0x200 0x300
      restrict R1 E
      ; Save pointer to malloc()
      mov R2 R255
      lea R2 [p_malloc]
      store R2 R1

   ; Downgrade PC to RX
   restrict PC RX

   ; Delete master capability from R255
   mov R255 0

   ; Clear other registers
   mov R0 0
   mov R1 0
```

```
95      mov R2 0
96
97      ; Setting up OS done!
98
99      ; Now go run a program
100     GOTO run_program
101
102
103
104 os_storage:
105     master:          empty, ; MASTER CAPABILITY (RWX, 0, MAX, 0)
106
107     p_memcpy:        empty, ; pointer to memcpy
108     p_malloc:        empty, ; pointer to malloc
109
110     p_internal_err: empty, ; pointer to internal_err
111     p_program_fail: empty, ; pointer to program_fail
112     p_program_halt: empty, ; pointer to program_halt
113
114     p_invalid:       empty, ; purposefully invalid capability (0, 0, 0, 0)
115
116
117
118 internal_err:
119     ; Set "exit code" to 2, the OS had an internal error
120     mov R0 2
121     GOTO exit
122
123 program_fail:
124     ; Set "exit code" to 1, a program failed
125     mov R0 1
126     GOTO exit
127
128 program_halt:
129     ; Set "exit code" to 0, all good
130     mov R0 0
131     GOTO exit
132
133 exit:
134     ; Unbind the halt interrupt to make sure we can actually exit
135
136     mov R1 PC
137     lea R1 [master - exit]
138     load R2 R1 ; r2 = master capability
139     lea R1 [p_invalid - master]
140     load R1 R1 ; r1 = p_invalid
141     lea R1 R0 ; add "exit code" to address of p_invalid so it shows up in the backtrace :)
142     lea R2 0xFFFF ; halt interrupt
143     store R2 R1 ; write p_invalid to the halt interrupt
144
145     ; Goodbye world!
146     halt
147
148
149
150 run_program:
151     ; We are assuming:
152     ; - the program we are running is located at address 0x300
153     ; - the program is smaller than 255 rows (not a hard limit, but we are restricting its PC space to
          [0x300..0x400[ )
154     ; - the program expects memcpy() in R32
155     ; - the program expects malloc() in R33
156
157     mov R0 PC
158     mov R3 R0
159     mov R4 R0
160     mov R32 R0
161     mov R33 R0
162
163     ; Setup the jump address in R0
164     lea R0 [-run_program + 0x300]
165     subseg R0 0x300 0x400
166     restrict R0 E
167
168     ; Load memcpy into R32
169     lea R32 [-run_program + p_memcpy]
170     load R32 R32
171
172     ; Load malloc into R33
173     lea R33 [-run_program + p_malloc]
174     load R33 R33
175
176     ; Load master capability in R3
177     lea R3 [-run_program + master]
178     load R3 R3
179
180     ; Set interrupt for fail
181     lea R4 [-run_program + p_program_fail]
182     load R5 R4
183     lea R3 0xFFFE ; interrupt for fail is at 0xFFFE
184     store R3 R5
185
186     ; Set interrupt for halt
187     lea R4 [-p_program_fail + p_program_halt]
188     load R5 R4
189     lea R3 1 ; interrupt for halt is at 0xFFFF
```

```
190     store R3 R5
191
192     ; Delete master capability from R3 and cleanup R4 & R5
193     mov R3 0
194     mov R4 0
195     mov R5 0
196
197     ; Run program
198     jmp R0
```

Listing 4: kernel.asm

7.2 Example programs

```
; This program copies a hello world string from its local memory into the heap.
; It also uses register r42 as a marker that can be seen in the backtrace,
;   r42 = '+' on start,
;   r42 = '-' at the end.

; We expect from the OS:
; R32 = memcpy()
; R33 = malloc()

program:
   geta R10 PC
   mov R42 '+'

   ; Call malloc
   mov R1 [payload_end - payload]
   mov R0 pc, lea R0 4, restrict R0 E
   jmp R33

   mov R2 R1

   ; Prep source memory
   marker0:
   mov R1 PC
   lea R1 [payload - marker0]
   add R11 R10 [payload]
   add R12 R10 [payload_end]
   subseg R1 R11 R12
   restrict R1 R0

   ; Call memcpy
   mov R0 pc, lea R0 4, restrict R0 E
   jmp R32

   mov R42 '-'
   halt

payload:
   "Hello World!"
payload_end:
```

Listing 5: hello_world.asm

```
; This program loads some numbers into registers r1 and r2,
; adds r1 and r2 together and saves the result to r0

; We expect from the OS:
; R32 = memcpy()
; R33 = malloc()

program:
   mov r1 pc
   mov r2 r1

   lea r1 [num_r1 - program]
   lea r2 [num_r2 - program]

   load r1 r1
   load r2 r2

   add r0 r1 r2

   halt

num_r1: 42
num_r2: 31415926
```

Listing 6: add.asm

7.3 Example backtrace

```
Machine backtrace:
> State: Running | PC: $(RWX, 0x0, 0xa, 0x0) | Instruction: mov R1 PC    | R1 = $(RWX, 0x0, 0xa, 0x0) | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x1) | Instruction: mov R2 R1    | R2 = $(RWX, 0x0, 0xa, 0x0) | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x2) | Instruction: lea R1 8     | R1 = $(RWX, 0x0, 0xa, 0x8) | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x3) | Instruction: lea R2 9     | R2 = $(RWX, 0x0, 0xa, 0x9) | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x4) | Instruction: load R1 R1   | R1 = 42                    | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x5) | Instruction: load R2 R2   | R2 = 31415926              | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x6) | Instruction: add R0 R1 R2 | R0 = 31415968              | New State: Running
> State: Running | PC: $(RWX, 0x0, 0xa, 0x7) | Instruction: halt         | New State: Halted
Machine status: Halted at address 0x7
```

Listing 7: The backtrace of the machine after running the program 'add.asm'.