CHERIoT

BY DAVID CHISNALL

The CHERI project has always had a close relationship with FreeBSD. It began from observing that Capsicum-based compartmentalization was great for new code but retrofitting it to existing libraries (with one process per library instance) was difficult for two reasons:

First, libraries want to share complex data structures, which imposes a lot of serialization overhead when turning the interfaces into messages sent over some inter-procedural communication (IPC) channel. A function call in a normal library would simply share a data structure by passing a pointer to an object. A privilege-separated library would need to authorize everything moved between the caller and callee. Libraries also often want long-term sharing, which imposes additional synchronization overhead.

Second, processes are isolated using a memory management unit (MMU), which provides a virtual-memory abstraction with mappings from addresses in a virtual address space to the underlying physical memory. Modern MMUs are fast because they have a translation look-aside buffer (TLB), a fast cache of translations. The TLB caches virtual to physical address translations. If a single page is shared between ten processes, it will take ten TLB entries. MMUs are great for isolation but poor for sharing.

CHERI is a set of architectural extensions that provide fine-grained memory safety for everything from assembly code on up.

These two problems led to the general observation: Isolation is easy, sharing is hard.

CHERI is a set of architectural extensions that provide fine-grained memory safety for everything from assembly code on up. CHERI, like Capsicum, is a capability system. In a capability system, every action must be accompanied by a capability, an unforgeable token of authority, that authorizes the action.

In contrast, a lot of other optimized mechanisms have a notion of *ambient authority*: you can perform an action simply because you are you. For example, if a process runs with UID 0 on a traditional UNIX system (without something like the FreeBSD MAC framework or SELinux), it can do a lot of privileged operations, whether it intended them or not.

Processes running as root are often security problems because it's easy to trick them into doing things that they shouldn't. In a capability system, a privileged process would instead hold a set of tokens that each authorize specific actions. It would need to choose the one to use at each point, preventing accidental use of elevated privileges.

MMUs have a similar notion of ambient authority. A running program can access any memory for which there is a valid mapping. If you have a buffer overflow, the MMU doesn't care that you didn't mean to access an adjacent object: you have the right to access that memory and so you can. The relevant piece of code may also hold a pointer to the other object, and so be authorized to access that object, but it didn't mean to at that point.

Capsicum extends file descriptors to be capabilities.

A Capsicum file descriptor has a rich set of permissions and, after entering capability mode (via the `cap_enter` system call) a process cannot access anything outside of its own memory without providing a valid file descriptor (capability), with the correct permissions, to a system call.

For example, in a normal POSIX process, you can call `open` to access any file that the user (or some MAC policy involving the user, the process, and maybe other things) has access to. In contrast, a capability-mode process must call `openat` and pass it a capability that authorizes access to a specific directory. This enforces the *principle of intentionality* and avoids a large class of vulnerabilities.

For example, if you mean to access something in a temporary directory, but happen to be given a path to some more important directory that you should not be accessing (for example, `../etc/rc.conf`), `openat` with a file descriptor to your temporary directory will fail (correctly, preventing an exploit) but `open` would succeed.

CHERI protects memory accesses in a way analogous to how Capsicum protects filesystem accesses. A conventional instruction set architecture has load, store, and jump instructions (sometimes, as with x86, combined with more complicated operations) that take an address as a base. This provides a model of memory that's similar to conventional UNIX's view of the filesystem: Any load or store can work if the process is allowed to load or store at that location. Buffer overflows or use-after-free bugs are to memory what path-traversal vulnerabilities are to the filesystem.

In a CHERI system, this address is replaced by a CHERI capability, an unforgeable value that authorizes access to a range of the address space. These values are stored in registers and memory and are protected by the hardware from tampering.

How does this make it easy to program?

Does a programmer have to track capabilities as some additional thing?

No, not at all.

It turns out that most programming languages already have an abstraction for a token that authorizes you to access a region of memory. They call it a pointer (or, in some cases, a reference). As a programmer, targeting a CHERI system, you mostly don't think about CHERI capabilities at all, you simply think about pointers.

If you do some arithmetic that takes a pointer out of the range of the object, you can't use it for loads or stores anymore. And, because the hardware knows precisely which things in memory are pointers and which aren't, it is possible to invalidate pointers when objects are freed. This means that you can share objects with a library on a CHERI system simply by passing pointers to those objects as arguments to a function exposed from the library.

Buffer overflows or use-after-free bugs are to memory what path-traversal vulnerabilities are to the filesystem.

The very earliest operating system for CHERI was a tiny microkernel but the vast majority of it was done on FreeBSD. CheriABI (a CHERI userspace ABI for FreeBSD) demonstrated a complete memory-safe userspace and a kernel on a friendly fork of FreeBSD (CheriBSD), which aims to upstream the changes before the 16.x release. A CHERI base architecture is currently being standardized as part of RISC-V, and FreeBSD 16 should have first-class support for all of the upcoming application-class CHERI cores implementing this instruction set.

FreeBSD was essential to the CHERI project. CHERI was a long-term hardware-software co-design project, which required modifying both hardware and software parts of the stack to explore where and how various ideas could best be implemented. This required a production-grade operating system that was easy to modify. The clear structure and well-defined abstractions of FreeBSD made this easy. We've seen with later Linux-on-CHERI work that the effort of adapting FreeBSD was far lower, and the project would probably not have had enough software engineers to have done the same on Linux as an initial target. The permissive license also made it easy to show vendors of other operating systems how various hardware features were used. FreeBSD was also an early adopter of LLVM, which has similar advantages in terms of ease of modification and license. It's easy to compile the entire FreeBSD base system with a modified LLVM, which makes testing new CPU features trivial. Brooks Davis wrote a much longer article about the benefits of FreeBSD for CHERI research in the May / June 2023 edition of the *FreeBSD Journal*.

In 2019, a few of us at Microsoft decided to see whether we could scale down the same abstractions to the smallest systems.

Scaling down CHERI

CheriABI demonstrated that you could run real POSIX applications, including large programs such as Chromium, on a full KDE desktop with Wayland and 3D drivers, in a memory-safe world. This could coexist with existing binaries, via a COMPAT64 layer, which worked much like the COMPAT32 layer that allows FreeBSD to run 32-bit programs on 64-bit systems. Most systems from mobile phones up to servers use the set of abstractions that were shown to work.

In 2019, a few of us at Microsoft decided to see whether we could scale down the same abstractions to the smallest systems.

We had three questions:

- Can you make the things that work well with CHERI on 64-bit systems work on 32-bit ones?
- If you have CHERI, what can you discard?
- What does an operating system look like if you can assume CHERI from the ground up?

The first is a slightly non-obvious problem. The abstractions that CHERI provides don't seem to be dependent on the size of an address, except for one thing: all of the metadata for a CHERI capability must fit in the same number of bits as the address. This means that on a 32-bit system we have half as much space for metadata.

CHERI uses a compressed encoding for the bounds that takes advantage of the fact that there is a lot of redundancy between the base of an object, the top of an object, and the address of a pointer to that object. On a smaller system, there is less redundancy and less space for the bounds. Fortunately, on embedded systems, the total amount of address space tends to be smaller, so there's less need to be able to precisely represent very large regions of the address space. A high-end microcontroller typically has well under 4 MiB of total RAM, so most objects are very small.

We also had fewer bits available for permissions than on a 64-bit system and had to compress our permission encoding, eliminating combinations that were either bad for security, not useful, or not meaningful.

How I used FreeBSD in developing CHERIoT

When we built the first CHERIoT prototypes, there were three core components:

- A CPU core written in BlueSpec SystemVerilog.
- A port of LLVM.
- A clean-slate RTOS.

BlueSpec SystemVerilog is a Haskell-based high-level hardware description language, which makes rapid prototyping easy. The BlueSpec compiler required a few small tweaks to build on FreeBSD, which we upstreamed, and FreeBSD is now a supported platform. It isn't yet in ports, but that would be great to see. With this, we could build a simulation of the core that ran on FreeBSD. Later, we moved to a production-quality code implemented in SystemVerilog and used verilator (from packages) to build a simulator on FreeBSD.

Like FreeBSD, CHERIoT RTOS is a permissively licensed, community-developed, operating system.

LLVM development on FreeBSD is very easy. FreeBSD is a first-class target for LLVM and LLVM is trivial to build. In some ways, it was too easy: when we needed to support people on LTS releases of some Linux distributions, we found that bootstrapping the version of LLVM that we were using was hard because it used a newer version of C++ than their stock toolchain supported.

For cutting-edge work, FreeBSD was far easier: multiple versions of GCC and LLVM were available in ports. This also made it easy to reproduce some of the build failures on other systems, by simply installing the old version of GCC that they shipped and configuring a build to try using it. Once those were working, cross-compiling and testing the RTOS was easy.

Lessons learned from FreeBSD

Like FreeBSD, CHERIoT RTOS is a permissively licensed, community-developed, operating system.

Most importantly, CHERIoT aims to copy FreeBSD's model of designing features before implementing them. The easiest time to change code is before it's written. This is how FreeBSD ended up with features like Jails, kqueue, and Capsicum: careful thought and design iteration, rather than throwing an API at users, hoping it works, and then living with the consequences.

We learn from Capsicum that we can make things that look to programmers like conventional file descriptors or handles into capabilities. Our software abstractions follow this model.

We also learned from `kqueue` that a single, simple, unified way of polling for any blocking event is easy. The `kqueue` design does not map particularly well to a privilege-separated RTOS, but the core idea does. Our scheduler exposes futexes (a simple atomic compare-and-wait-if-equal operation) as the only blocking event source. Interrupts are mapped to futexes, so a thread waits for an interrupt by simply waiting on a futex. The scheduler then layers a multiwaiter API on top, allowing a thread to wait for multiple events, either from hardware or software.

Perhaps most importantly, we've learned from FreeBSD that documentation is king. It doesn't matter how amazing your system is if no one can figure out how to use it. Between well-written man pages and the Handbook, FreeBSD is easy to learn. Writing a book for developers wanting to pick up CHERIoT RTOS was a priority and it was published earlier this year. On top of that, we have doc comments for every API, which are parsed by modern IDEs (and vim with our version of `clangd` as the language-server protocol implementation).

Lessons FreeBSD could learn from CHERIoT

CHERIoT RTOS is currently written entirely in C++. C++ has a lot of advantages over C for systems programming. It's easy to create rich abstractions that are checked at compile time. For example, we have a message queue design that uses a single counter for each of the producer and consumer pointers and has to handle the cases where these values wrap.

In C++, we can define `constexpr` functions to do the increments, and then write templates that `static_assert` over their behavior. Every time that we compile the file that defines these, the compiler will exhaustively check the overflow behavior on every possible value of producer and consumer pointers for some small queue sizes.

Using rich types also lets us avoid a lot of errors by construction. For example, we have a `PermissionSet` class that manages the set of permissions on a CHERIoT capability. This is a `constexpr` set that lets you construct permissions by name and will generate the bitmap that the instructions that operate on permissions expect.

Using rich types also lets us avoid a lot of errors by construction.

In the loader, we use this to describe both the permissions that we want on a capability and the permissions that each of the roots have. We will get a compile failure if we try to derive a permission that is not present in the original, which is far easier to debug than a later instruction failing because it lacks an expected permission in one of its operands.

We make a lot of use of a pattern where we have an inline templated function that does some compile-time checks and is optimized away, and a calls a type-erased function. Our type-safe logging works like this, for example. We have a `printf`-like function that takes an array of the arguments to print and the union discriminators. This is constructed by some templates that generate the discriminator values based on the types, so whether you want to log a pointer, an enumeration value, or a MAC address, you'll get the correct output, without having to correctly match the type in the string and with compile-time checks.

This is extensible. MAC addresses, for example, are not built in, the network stack defines a callback for them and a template specialization that handles the mapping.

We don't yet have a Rust compiler (it's coming soon!) but when we do, we expect to start using Rust for some components as well. Richer types in systems programming languages let us both avoid bugs at compile time and also write a lot less code.

In a lot of key places in the RTOS, we'd need much more source code if we used C, and that code would be harder to maintain. It would also be far harder to find developers who are familiar with the language. Today, the number of lines of new code that we see written for systems languages are ordered:

1. C++
2. C
3. Rust

It's easier today to find C++ developers than either of the other two, but that ignores the trend. C++ has been seeing fairly slow growth since C++11 was introduced. C has been seeing a much sharper steady decline in the same time. Rust was seeing steady growth, but it's accelerated over the last three years. I expect that it will be easier to find Rust developers than C developers very soon and probably easier to find Rust developers than C++ within a decade or so.

This is unsurprising.

Developers tend to favor languages in which they can be more efficient. There is ongoing work in FreeBSD to support Rust in the base system but there's a much simpler path to adopting modern C++, which can make it far simpler to express complex concepts than C and provide a lot more compile-time checks.

FreeBSD has also done some fantastic work using Lua for things that are not absolutely performance critical, including userspace parts, policies in the bootloader, and ZFS channel programs. Lua is a far simpler language to learn than any of the above list and is easy for relatively inexperienced programmers to be productive in. We use Lua in our build system (and for typesetting the CHERIoT book!), but sadly the Lua VM takes about as much memory as a typical microcontroller has in total, so we can't use it in the RTOS.

Richer systems programming languages are important but FreeBSD's use of Lua is a good reminder that a lot of things—even in the kernel—don't actually need a systems programming language.

Developing for CHERIoT on FreeBSD

The CHERIoT toolchain is in ports on FreeBSD, as is the `xmake` build tools, and so you can install them simply with:

```
# pkg ins cheriot-llvm xmake-io git
```

That gives you the prerequisites for building CHERIoT firmware. Note that, at the time of writing, the version of CHERIoT LLVM in the quarterly branch is 18, whereas the version in the latest branch is 20 (and about to be updated to 21), so it's a good idea to use the latest branch for development.

You should now be able to try cloning the RTOS and building a simple firmware image.

First, clone the RTOS:

```
$ git clone --recurse https://github.com/CHERIoT-Platform/cheriot-rtos
$ cd cheriot-rtos
```

Next, you need to configure the build:

```
$ cd examples/01.hello_world/
$ xmake config --sdk=/usr/local/llvm-cheriot
checking for platform ... cheriot
checking for architecture ... cheriot
Board file saved as  build/cheriot/cheriot/release/hello_world.board.json
Remapping priority of thread 1 from 1 to 0
generating /tmp/cheriot-rtos/sdk/firmware.rocode.ldscript.in ... ok
generating /tmp/cheriot-rtos/sdk/firmware.ldscript.in ... ok
generating /tmp/cheriot-rtos/sdk/firmware.rwdata.ldscript.in ... ok
$ xmake
...
[100%]: build ok, spent 13.796s
```

If you have a lowRISC Sonata board, you can add `--board=sonata` to the end of the `xmake config` line and you'll get an ELF file targeting their board. The board shows up as a USB mass-storage device with a FAT filesystem that can load a UF2 file. If you do `xmake run`, it will point you to the missing Python package that you need to install from `pip` to convert the ELF to a UF2 file. Once this is installed, it will either tell you which file to copy, or copy it with no further interaction if the `SONATA` filesystem is mounted in a common location.

If you don't, then you can run the resulting firmware in a simulator. By default, these examples will target the Sail simulator. This requires OCaml and does build on FreeBSD, but requires some manual steps. The project also ships a Linux dev container, which works very nicely in the FreeBSD Linuxulator with Podman:

```
# pkg ins podman-suite
# podman run --rm -it --os linux -v path/to/cheriot-rtos:/home/cheriot/cheriot-rtos
ghcr.io/cheriot-platform/devcontainer:x86_64-latest
```

This will drop you into an ephemeral container with your clone of the RTOS source code mounted in `/home/cheriot/cheriot-rtos`. Note that the current version of Podman in ports doesn't like using the tag that refers to a multi-arch container when the OS doesn't match the host. We provide binaries for x86-64 and AArch64, so if you're on AArch64 simply replace `x86_64` with `aarch64` in the above line.

The dev container has all of the tools installed in `/cheriot-tools`, so you can try building the example again in the same way as earlier:

```
$ cd examples/01.hello_world/
$ xmake f --sdk=/cheriot-tools
...
$ xmake
...
[100%]: build ok, spent 13.524s
$ xmake run
Board file saved as  build/cheriot/cheriot/release/hello_world.board.json
Remapping priority of thread 1 from 1 to 0
Running file hello_world.
ELF Entry @ 0x80000000
tohost located at 0x80006448
```

```
Hello world compartment: Hello world
SUCCESS
```

Congratulations, you've run memory-safe C++ code in a simulator built from a formal model of the ISA!

The dev container contains three other simulators:

- The cycle-accurate verilator simulator for the Ibex core with a minimal set of peripherals.
- The simulator for lowRISC Sonata boards.
- The MPact simulator from Google that provides a high-performance simulator with an integrated debugger

The MPact simulator is compatible with the Sail (simplified) machine.

You can try running it directly:

```
$ /cheriot-tools/bin/mpact_cheriot build/cheriot/cheriot/release/hello_world
Starting simulation
Hello world compartment: Hello world
Simulation halted: exit 0
Simulation done: 106508 instructions in 0.2 sec (0.5 MIPS)
Exporting counters
```

You can do the development in FreeBSD and run just the simulators in the container, or do all of the development in the container. On other platforms, most developers use an editor such as Visual Studio Code that has dev container integration. This should also be possible on FreeBSD, configuring Podman to run the Linux version of the container.

Alternatively, you can build all of the simulators natively for FreeBSD. The instructions for this are too long for this article, but look at the documentation in the RTOS repository for guidance. All of the dependencies for them are in ports and hopefully the simulators themselves will be soon!

DAVID CHISNALL's background spans operating systems, compilers, hardware, and security. He is the author of the Definitive Guide to the Xen Hypervisor, has been an LLVM committer since 2008 and was a member of the FreeBSD Core Team from 2012 to 2016. He joined the CHERI project to lead the compilers and languages thread of the research at the University of Cambridge in 2012. He continued to work on CHERI, including leading the creation of CHERIoT, at Microsoft from 2018 to 2023. He is co-founder and Director of Systems Architecture at SCI Semiconductor, which makes CHERIoT SoCs, and co-maintainer of the CHERIoT Platform open-source project.