

SEE
TEXT
ONLY

BUILDING A
FOUNDATION
FOR SECURE,
TRUSTED
COMPUTING
BASES

CHERI

C
A
P
A
B
I
L
I
T
Y

H
A
R
D
W
A
R
E

E
N
H
A
N
C
E
D

R
I
S
C

I
N
S
T
R
U
C
T
I
O
N
S

by Brooks Davis



BSD operating systems have been around since the 1980s, and the history of UNIX extends all the way back to 1969, but despite orders of magnitude growth in performance, storage, and memory capacity, we still use CPUs with computing models that are remarkably similar to the PDP-11 on which the early versions of UNIX were run. We have a flat virtual address space (now 64 bits instead of 16), TLB-based process virtualization of memory, and permissions at page granularity.

This model has many advantages and has served us well for many years, but it also suffers from serious disadvantages. Multiple classes of memory corruption bugs including buffer overflows are consistent sources of vulnerabilities despite years of work to address them. As a community, we employ many mitigation techniques such as address-space layout randomization (ASLR) and compartmentalization (also known as privilege-separation), but these have significant costs. ASLR has a non-negligible performance overhead, provides protection that is statistical not absolute, and certain common programming models such as pre-fork servers allow efficient automated attacks. Compartmentalization limits the impact of failures of other mitigations by confining risky parts of programs to environments where their privilege is limited. On compartmentalizing software, programmers transform programs into distributed computations, increasing context switch overhead and TLB pressure while converting single-address-space programs into distributed systems with all the attendant complexity. This means compartmentalization is commonly used only where it is a trivial match to the application's programming model (`uniq(1)`) or where vulnerabilities would be the most critical (`sshd(1)`, Chrome, or Firefox).

Developed under the auspices of the DARPA CRASH (Clean-slate design of Resilient, Adaptive, Secure Hosts) program, CHERI (Capability Hardware Enhanced RISC Instructions) is a hardware/software co-design project that challenges the assumptions we've made about hardware software interfaces for the last 40-plus years. We have developed extensions to the MIPS64 ISA that provide robust, fine-grained, hardware-enforced, process-scope memory *capabilities* that enforce object boundaries and permissions in a manner compatible with C pointers. We have further extended these capabilities to provide robust, efficient, in-process compartmentalization. Our extensions are implemented as a MIPS coprocessor allowing incremental deployment of CHERI features.

To develop and demonstrate these features, we have implemented an FPGA soft-core CPU, ported FreeBSD to it, and extended FreeBSD with support for CHERI capabilities [Woodruff]. We have added support to LLVM and Clang to use capabilities in C, both a hybrid mode where specially annotated pointers become capabilities and another where all pointers are capabilities [Chisnall]. Additionally, we have extended our FreeBSD port (CheriBSD) to support in-process compartmentalization of code and demonstrated the viability of the approach with the `tcpdump` program and `zlib` library [Watson].

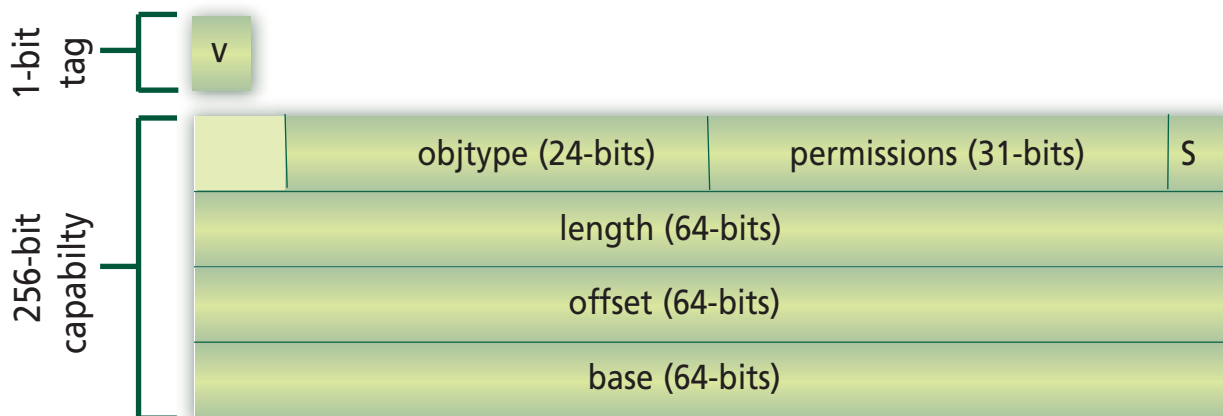
CHERI Capabilities

CHERI capabilities are unforgeable references to regions of virtual address space. They are stored in memory accompanied by a tag bit that verifies their validity (vs. some arbitrary arrangement of bits) and are manipulated in special capability registers. Capabilities contain a *base*, *length*, *offset* relative to the base, *permissions*, and a *type*. Instructions that manipulate capabilities may only shrink the region bounded by the base and length or reduce permissions. Attempts to increase the scope or permissions of a capability raise a hardware exception. Thus, capabilities provide monotonically decreasing rights.

All memory accesses are performed through a capability. This may occur directly via new capability-based load and store instruction or indirectly via the default data capability (DDC). When running capability-unaware or hybrid programs, DDC is set to a capability with rights to the whole address space. This allows all legacy load and store instructions to work as expected in hybrid or pure-MIPS binaries.

Because all memory accesses are via capabilities, the portion of address space that is reachable by a given thread is the transitive closure of the address space reachable by the set of capabilities in the register file. That is to say, all memory that can be accessed by a capability currently in the register file or via a capability that can be loaded from the memory the register file grants access to. Thus, compartmentalization can be achieved by transforming the contents of the register set. This is achieved with the `CCa11` instruction, which takes a paired code and data capability having the same type, stores the current register contents, and sets up a new register set executing in the code capability and having access to argument registers and registers specified in the data capability. To exit a sandbox, a `CReturn` instruction restores the register set from a trusted stack.

In our primary prototype, capabilities are 256 bits and strongly aligned with a tag bit stored in a separate, inaccessible portion of DRAM. The current layout is shown in the figure below.



The primary overhead of using capabilities comes from the increased memory and especially cache footprint of code where pointers become capabilities. Pointer-intensive benchmarks have measurable overhead, but thus far real-world programs such as `tcpdump` show no significant penalty.

C Support

Unlike many past capability systems, we designed CHERI capabilities with the explicit goal of using them as C pointers. This has had considerable impact on our ISA and the contents of our capabilities. Most significantly, our capabilities include not only base and length, but offset, because real-world C programs often temporarily address values outside their allocated range.

Our initial focus on supporting capabilities in C was adding a new annotations `__capability` to pointers that we wished to restrict. For example, we annotated a version of `tcpdump` where we protected the packet pointer preventing out-of-bounds access from producing spurious results. The thousands of lines requiring annotation and the thousands of merge conflicts produced in the first import of a new `tcpdump` code base convinced us that a pure-capability mode was required for CHERI to be usable. We can currently compile nearly all C code in pure-capability mode and do so for most compartmentalized code. We continue to use hybrid mode in support code in `libc` that must be capability aware, and in transitions between MIPS64 and pure-capability code (usually around the edges of compartmentalized code).

Supporting CHERI capabilities as pointers also has a number of ABI subtleties that are beyond the scope of this article.

CheriBSD

To ensure that our ideas are truly viable, we have made it a goal from the start to run a real operating system and application stack on CHERI and to allow incremental adoption of CHERI features in software. To this end, we first brought up FreeBSD on CHERI adding support for those features required to support our prototype board without support for capabilities [Davis]. We separately added kernel support for running programs containing capabilities. This includes process startup, context switch code, signal handling, and debugging. The table below shows a breakdown of the modest set of changes required.

Component	Files Modified	Lines Added	Lines Removed
Headers	19	1,424	11
CHERI initialization	2	49	4
Context management	2	392	10
Exception handling	3	574	90
Memory copying	2	122	0
Virtual memory	5	398	27
Object capabilities	2	883	0
System calls	2	76	0
Signal delivery	3	327	71
Process monitoring/debugging	3	298	0
Kernel debugger	2	264	0

To support hybrid capability programs, we modified `libc` slightly to make memory manipulation function (`memcpy()`, `memmove()`, `qsort()`) capability aware to allow structures to contain capabilities. With these changes we continue to support unmodified MIPS64 binaries, and, in fact, most of our userspace programs are capability aware only in so far as it is easier (and potentially faster due to copying more bytes per instruction) to unconditionally use capability-aware memory manipulation functions. We have also provided capability aware variants of string and memory manipulation functions. For example, a capability aware (and thus memory-safe) `strcpy_c()` implementation is shown below.

```
__capability char *
strcpy_c( __capability char * __restrict to,
          __capability const char * __restrict from)
{
    __capability char *save = to;

    for (; (*to = *from); ++from, ++to);

    return(save);
}
```

Due to the initially weak support for MIPS64 in `clang` we modified the FreeBSD build system to allow select libraries to be built with our modified `clang` version while compiling the rest of the system with the

base `gcc`. We have also modified the build system to build pure-capability versions to all libraries similar to the way the normal build system builds 32-bit versions of libraries on 64-bit systems. This both allows us to more thoroughly test compiler support and to link unmodified libraries into compartmentalized sections of code that use pure-capability mode.

libcheri

Compartmentalization in CheriBSD is currently implemented in the `libcheri` library. It provides an interface to load sandbox classes and create sandbox objects. Sandbox objects are effectively mini-address spaces within a process—`libcheri` maps a region of address space, loads a compartmentalized object into it, and securely calls methods implemented by that object. In our current implementation, compartmentalized code is typically pure-capability code in order to take advantage of memory safety guarantees and to simplify passing of capabilities from outside the sandbox, but other models are possible. For example, a bit of wrapper code could allow an unmodified 32-bit library to run in a sandbox within a 64-bit program.

Conceptually, `libcheri` sandboxes are libraries with the added twist that multiple instances of each library can be instantiated and those instances can fail and be reset independently. This allows risky code such as `tcpdump` packet decoding or `zlib` decompression to be placed in a sandbox where failure has reduced consequences as the sandboxed code has no direct ability to make system calls and greatly reduced ability to impact the main process.

We have used `libcheri` compartmentalization to protect the main `tcpdump` process (often run as root!) from the packet dissection and printing code (handcrafted C to process untrustworthy and frequently corrupt data from the network). We are able to protect against a wide range of attack models from simple crashes to infinite loop-based denial of service attacks and even supply chain attacks where a packet triggers a bug in a malicious dissector. With these changes active, the impact of a crashing or denial of service bug is limited to the loss of formatted output for the given packet and a brief slow-down as we reload the sandbox instance.

In addition to protecting applications like `tcpdump` from their internals, we have also implemented library compartmentalization where we present an API- and ABI-compatible interface to a compartmentalized `zlib` library. This allows completely unmodified, dynamically linked `gzip`, `gif2png`, and similar programs to gain the benefits of a compartmentalized and memory safe `zlib` without even recompiling. This strategy of library compartmentalization allows the effort of compartmentalization to benefit as many consumers as possible with the least effort.

Contributions to FreeBSD

In the course of our work on Cheri and CheriBSD, we have contributed a number of changes back to FreeBSD. During our initial bring-up, we improved support for CFI flash devices, ported support for Flat Device Trees and the FreeBSD boot loaders to MIPS, and generally improved MIPS support. We have merged this work upstream as well as drivers for specific Altera and Terasic hardware on our Terasic DE4 reference board.

As we've worked on Cheri, our stricter interpretation of the C standard has found occasional correctness issues in code in FreeBSD. As we've found general issues, we've merged these changes as they will benefit both potential Cheri-based platforms and other memory safety systems like Intel's MXP once C language support is implemented.

We have also supported the development of QEMU user mode support to allow us to build packages for use on our FPGAs. Building packages on embedded boards like the EdgeRouter™ Lite is slow, but building them on a 100Mhz FPGA with slow I/O is completely impractical and so we've had to support the creation of new infrastructure.

Finally, should hardware implementations of Cheri emerge, CheriBSD stands ready as a reference platform and a base for mainline Cheri support in FreeBSD.

Future Work

We are currently exploring a new system call interface where system call argument pointers are capabili-

ties. This will allow us to run significant numbers of unmodified programs in pure-capability mode. Our hope is that we can build and safely run a FreeBSD userspace with full spatial memory in the near future. We expect to find a number of subtle bugs as we enforce bounds on such a large codebase.

Library compartmentalization is a powerful tool for improving the safety of diverse codebases. We have implemented a number of compartmentalization prototypes, but need to port a wider range of libraries. Doing so will help us determine the sorts of tools we need to develop to ease the process. We have already done some work in this area with our SOAAP project [Gudka], but have mostly focused on application compartmentalization. Library compartmentalization poses a similar but related set of issues. In some cases, it may be practical and advisable to implement process-based library compartmentalization for some libraries. Libraries with buffer-based interfaces such as `zlib` are ill-suited to this, but libraries with stream-oriented interfaces may achieve reasonable performance even in a process-oriented environment.

As cache pressure is one of the main performance impacts of CHERI, we are currently perusing a 128-bit compressed capability model for CHERI. Our experiments show this lowers the overhead significantly at the cost of a loss of granularity for large objects. We hypothesize that the loss of granularity largely maps to existing practices of rounding up allocations, but work is ongoing to confirm this.

Further Reading

This article only scratches the surface of our work on CHERI. Our three conference papers cover it in much greater detail and show the evolution of our ideas as we have developed more software support. *The CHERI capability model: Revisiting RISC in an age of risk* [Woodruff] covers the key memory safety properties of CHERI. *Beyond the PDP-11: Processor support for a memory-safe C abstract machine* [Chisnall] shows changes required to implement C on top of CHERI. Finally, *CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization* [Watson-Oakland] details our compartmentalization strategy.

Those interested in the nuts and bolts of the ISA may find *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture* [Watson-ISA] to be of interest.

Trying CHERI

We have released a QEMU implementation of CHERI which tracks our our CheriBSD, Clang, and LLVM progress on our Github repository <https://github.com/CTSRD-CHERI/>.

We have made our CHERI CPU FPGA implementation and snapshots of related software available as open source at <http://chericpu.org>. FPGA releases are infrequent and lag the current state of development. If you have a need for the FPGA version, please contact us.

BROOKS DAVIS is a Senior Software Engineer in the Computer Science Laboratory at SRI International and a Visiting Research Fellow at the University of Cambridge Computer Laboratory. He has been a FreeBSD user since 1994, a FreeBSD committer since 2001, and was a core team member from 2006 to 2012. Brooks earned a Bachelor's Degree in Computer Science from Harvey Mudd College in 1998. His computing interests include security, operating systems, networking, high-performance computing, and, of course, finding ways to use FreeBSD in all these areas. When not computing, he enjoys cooking, brewing, gardening, wood-working, blacksmithing, and hiking.

REFERENCES

[Chisnall] David Chisnall, Colin Rothwell, Brooks Davis, Robert N. M. Watson, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, and Michael Roe. "Beyond the PDP-11: Processor support for a memory-safe C abstract machine," Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015), Istanbul, Turkey. (March 2015) <http://www.cl.cam.ac.uk/research/security/ctsrld/pdfs/201503-asplos2015-cheri-cmachine.pdf>

[Davis] Brooks Davis, Robert Norton, Jonathan Woodruff, and Robert N. M. Watson. "Bringing Up MIPS," FreeBSD Journal. (January/February 2015)

[Gudka] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. "Clean Application Compartmentalization with SOAAP," Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015), Denver, Colorado. (October 2015) <http://www.cl.cam.ac.uk/research/security/ctsrld/pdfs/2015ccs-soaap.pdf>

CONTINUES NEXT PAGE

REFERENCES CONTINUED

[Watson-Oakland] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," Proceedings of the 36th IEEE Symposium on Security and Privacy ("Oakland"), San Jose, California. (May 2015) <http://www.cl.cam.ac.uk/research/security/ctsr/pdfs/201505-oakland2015-cheri-compartmentalization.pdf>

[Watson-programmers-guide] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. "Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide," Technical Report UCAM-CL-TR-877, University of Cambridge, Computer Laboratory. (September 2015) Current CHERI programmer's guide <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-877.pdf>


[Watson-ISA] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, and Stacey Son. "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture," Technical Report UCAM-CL-TR-876, University of Cambridge, Computer Laboratory. (September 2015) Current CHERI ISA specification <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-876.pdf>

[Watson-BERI-software] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. "Blue spec Extensible RISC Implementation: BERI Software Reference," Technical Report UCAM-CL-TR-869, University of Cambridge, Computer Laboratory. (April 2015) <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-869.pdf>

[Watson-BERI-hardware] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markets, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Robert Norton, and Michael Roe. "Blue spec Extensible RISC Implementation: BERI Hardware Reference," Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory. (April 2015) <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-868.pdf>


[Woodruff] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk," Proceedings of the 41st International Symposium on Computer Architecture (ISCA 2014), Minneapolis, Minnesota. (June 14-16, 2014) <http://www.cl.cam.ac.uk/research/security/ctsr/pdfs/201406-isca2014-cheri.pdf>



This paper is approved for public release; distribution is unlimited. It was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under Contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies of the U.S. Department of Defense.




Rack-mount networking server


Designed for BSD and Linux Systems
Up to **5.5Gbit/s** routing power!

Made for  FreeBSD





 **PERFECT FOR**


- BGP & OSPF routing
- Firewall & UTM Security Appliances
- Intrusion Detection & WAF
- CDN & Web Cache / Proxy
- E-mail Server & SMTP Filtering
- Anti-DDoS and clean pipe filtering

 **KEY FEATURES**


- 6 NICs w/ Intel igb(4) driver w/ bypass
- Hand-picked server chipsets
- Netmap Ready (FreeBSD & pfSense)
- Up to 14 Gigabit expansion ports
- Up to 4x10GbE SFP+ expansion




DESIGNED FOR



DESIGNED FOR



DESIGNED FOR



DESIGNED FOR

Designed. Certified. Supported

1 Gbit/s Copper	Ports	Chipset
L800-G808-1	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G808-2	8x Gbe RJ-45 ports	8x Intel i210 AT; PEX8618
L800-G428-1	4x Gbe RJ-45 ports	1x Intel i350 AM4
L800-G428-2	4x Gbe RJ-45 ports	1x Intel i350 AM4
1 Gbit/s SFP (Fiber)	Ports	Chipset
L800-S406-1	4x Gbe SFP ports	i350-AM4
10GbE Copper	Ports	Chipset
L800-T202-1	2x 10GbE RJ-45 ports	Intel X540
L800-T203-1	2x 10GbE RJ-45 ports	Intel X540
10GbE SFP+ (Fiber)	Ports	Chipset
L800-X204-1	2x 10GbE SFP+	Intel 82599ES
L800-X205-1	2x 10GbE SFP+	Intel 82599ES
L800-X405-1	4x 10GbE SFP+	Intel 82599ES; PEX8724

contactus@serveru.us | www.serveru.us | 8001 NW 64th St. Miami, LF 33166 | +1(305) 421-9956