CheriABI
Hardware enforced memory safety for FreeBSD


SRI International, University of Cambridge, Microsoft Research, Google, Inc

Approved for public release; distribution is unlimited. This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 (“CTSRD”) and HR0011-18-C-0016 (“ECATS”). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Punchline: it really does work

• Full FreeBSD operating system with spatial and referential memory safety
  • Covers programs, libraries, and linkers
  • Kernel access to user memory
• Performance is generally acceptable
• Significant 3rd-party software works: PostgreSQL database, Webkit
Introduction to CHERI

- CHERI introduces a new register type: the **capability**
  - In addition to integer and floating point
- CHERI capabilities grant access to bounded regions of virtual address space
  - Protected by tags

Watson, et al. **CHERI: a research platform deconflating hardware virtualization and protection.** RESoLVE 2012.

Architectural CHERI capabilities extend pointers with:

- **Tags** protect capabilities in registers and memory
- **Bounds** limit range of address space accessible via a pointer
- **Permissions** limit operations – e.g., load, store, instruction fetch
128-bit compressed capabilities

- **Compress bounds** relative to 64-bit virtual address
  - Floating-point bounds mechanism constrains bounds alignment
  - Security properties maintained (e.g., provenance, monotonicity)
  - Strong C-language support (e.g., for out-of-bound pointers)
- DRAM tag density from 0.4% to 0.8% of physical memory size
- Full prototype with full software stack on FPGA
- Implications for memory allocators, object alignment, etc
CHERI memory operation

• All memory access via CHERI capabilities
  • Explicit (new instructions):
    • Capability load, store, branch, jump
  • Implicit (legacy MIPS ISA):
    • via Default Data Capability (DDC) or Program Counter Capability (PCC)
CHERI capability manipulation

• Capabilities are used and manipulated in capability registers with capability instructions
  • Manipulations are monotonic (can only reduce bounds and permissions)
    • CAndPerm cd, cb, rt
    • CSetAddr cd, cs, rs
• Capabilities can be stored in memory, protected by tags
  • Non-capability stores clear tags
Capabilities as C pointers

• CHERI capabilities are designed for use as C pointers
  • Allowed to be out of bounds between dereferences
  • Can store 64-bit integers (untagged)
  • No protection tables or privileged operations

• Two compilation modes:
  • Hybrid: __capability annotation applied to select pointers
  • Pure-capability: all pointers are capabilities

CheriABI: Pure-capability process environment

- Built on CheriBSD (FreeBSD modified for CHERI)
- All program pointers are capabilities
  - Including syscall arguments and return values
- Goal: Bounds are minimized
  - C-language objects
  - Pointers provided by the kernel
- Goal: run pure-capability programs with simple recompilation


Implementation: kernel

- CheriABI is implemented as a compat layer (i.e. freebsd32)
- The kernel is a hybrid CHERI-C program
  - Pointers to userspace are annotated with __capability and are capabilities.
  - Select data structures (e.g. struct iovec, signal bits) converted to store capabilities.
- All userspace access via capabilities
  - Capability aware versions of userspace access functions: copyin_c/copyout_c/fueword_c, etc
  - Non “_c” versions return error for CheriABI processes
  - Capabilities not copied to/from userspace by default
    - Special copyincap/copyoutcap used to ensure copy is intentional
Abstract capabilities

How should the systems programmer think about bounds?

New concept: abstract capability
• Set of permissions of the process
• Tracks ghost state across swapping, etc
• Constructed and maintained by a collaboration of the kernel and language runtime
## System startup

### Power-on state

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DDC</strong></td>
<td>RWX 0x0 - 0xFF...FF</td>
</tr>
<tr>
<td><strong>PCC</strong></td>
<td>RWX 0x0 - 0xFF...FF</td>
</tr>
<tr>
<td><strong>C1-31</strong></td>
<td>NULL</td>
</tr>
</tbody>
</table>

### Early boot

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DDC</strong></td>
<td>RW- 0x0 - 0xFF...FF</td>
</tr>
<tr>
<td><strong>PCC</strong></td>
<td>R-X 0x0 - 0xFF...FF</td>
</tr>
<tr>
<td><strong>C1-31</strong></td>
<td><strong>Working set</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>All tags clear</strong></td>
<td></td>
</tr>
<tr>
<td><strong>UserRoot</strong></td>
<td>RWX 0x0-0xFFFF</td>
</tr>
<tr>
<td><strong>SwapRoot</strong></td>
<td>RWX 0x0 - 0xFF...FF</td>
</tr>
</tbody>
</table>
Execve

Initial register values

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDC</td>
<td>NULL</td>
</tr>
<tr>
<td>PCC</td>
<td>RWX</td>
</tr>
<tr>
<td>CSP</td>
<td>RW-</td>
</tr>
<tr>
<td>C03</td>
<td>RW-</td>
</tr>
</tbody>
</table>

UserRoot: RWX 0x0-0x0000007F...FF

Kernel

Userspace

Program binary

Run-time linker

Thread Stack

Process arguments

auxargs

environ

argv

Arg & environ strings
Virtual-memory system

• Programmer visible:
  • Provides capabilities to newly mapped regions via \texttt{mmap()} and \texttt{shmat()}
  • Alters and frees mappings

• Abstract capability maintenance:
  • Ensures correct virtual to physical mappings
  • Preserves stored capabilities in swapped pages
Virtual-memory system: mmap

• `mmap()` allocates virtual address space and changes mappings
• In CheriABI returns a bounded pointer
  • Imprecise mapping requests rejected
  • User must round-up unpresentable requests
• Permissions are set based on page permissions
  • `PROT_MAX()` extension allows `PROT_NONE` mappings for reservation
Virtual-memory system: swap
Run-time linker

- Loads and links dynamic libraries
- Resolves symbols and synthesizes capabilities
- Jumps to program entry point
- Provides on-demand loading of libraries and supports exception handling
C runtime

• Objects allocated by `malloc()` are bounded to requested size
• `realloc()` adjusts bounds or allocates new storage
• Thread-local storage is bounded
  • Currently to per-thread storage
• Compiler generated code sets bounds on stack, automatic, and global objects as required
System calls

```c
read(fd, buffer, nbyte);
```

copyout(kaddr, buffer, len);
...

kern_readv(td, fd, {buffer, nbyte});
cheriabi_read(td, uap);
Kernel code changes: read()

```c
int user_read(struct thread *td, int fd, void * __capability buf, size_t nbyte)
{
    struct uio auio;
    kiovec_t aiov;
    if (nbyte > IOSIZE_MAX)
        return (EINVAL);
    IOVEC_INIT_C(&aiov, buf, nbyte);
    auio.uio_iov = &aiov;
    ...
    return (kern_readv(td, fd, &auio));
}
```

Called by `sys_read()` and `cheriabi_read()`

New init macro for struct iovec
if ((nstrings = realloc(we->we_strings, we->we_nbytes)) == NULL) {
    error = WRDE_NOSPACE;
    goto cleanup;
}
for (i = 0; i < vofs; i++)
+    if (we->we_wordv[i] != NULL) {
+        we->we_wordv[i] += nstrings - we->we_strings +
+            (we->we_wordv[i] - we->we_strings);
+    }
we->we_strings = nstrings;
Required changes: summary

• Userspace: 1% (~200) of files required changes
  • Concentrated in libraries
  • Most programs require no changes
• Kernel: <6% of files (~750) required changes
  • Pervasive changes to iovec, signal handlers, network interface ioctl handlers
  • A pure-capability kernel could reduce changes

• Many changes improve code quality
  • We have upstreamed many to FreeBSD (compat32 improvements, etc)
Capability bounds minimization (OpenSSL)

Most capabilities bound small regions (<<1 page)

Small number of whole shared-object references remain in startup code

Stack references

Better
• Micro-benchmark performance generally acceptable
  • <10% overhead in most cases
  • Graph excludes crypto and bit-manipulation outliers
Reflections on using FreeBSD for CheriABI

• Good:
  • Well-abstracted process ABI infrastructure
    • SysV stack ABI somewhat baked in
  • Central, generated system call tables, stubs, etc
  • Single, hackable build system

• Bad
  • Centralized copyin/copyout for ioctl divorces copy from types
  • Tests require ports/packages (kyua)
    • No easy way to build kyua static
Work in progress

- Porting ISA from MIPS64 to RISC-V
- New compressed capability format
- Temporal memory safety
- Make CheriABI the default ABI
  - Add a compat/freebsd64
- Pure-capability kernel
Future work on FreeBSD

• More compatX cleanup
  • Code deduplication
  • Remove separate syscalls.master

• Rework ioctl interface
  • Konrad Witaszczyk (def@) is working in this area

• Refactor use of initial stack for arguments
  • Needed for CheriABI, likely helpful for ASLR

• Upstream CHERI/CheriABI support
  • Hardware platform required, but hopefully coming
Conclusions

• Full UNIX-like operating system with spatial and referential memory safety
  • Covers programs, libraries, and linkers
  • Kernel access to user memory
• Some fundamental operating system changes required
  • Generally non-disruptive
• 3rd-party software works:
  PostgreSQL database, Webkit
Further Reading

http://cheri-cpu.org/


This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.