Creating a memory-safe workstation with CheriBSD

Brooks Davis
SRI International

May 20, 2023
BSDCan 2023, Ottawa, Canada
[demo@morello ~]$ sysctl hw.machine_arch hw.model
hw.machine_arch: aarch64c
hw.model: Research Morello SoC r0p0
[demo@morello ~]$ file /usr/local/bin/konsole
/usr/local/bin/konsole: ELF 64-bit LSB pie executable, ARM aarch64, C64, CheriABI, version 1 (SYSV), dynamically linked, interpreter /libexec/ld-elf.so.1, for FreeBSD 14.0 (1400064), FreeBSD-style, with debug_info, not stripped
[demo@morello ~]$
```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#pragma weak write_buf

void write_buf(char *buf, size_t ix)
{
    buf[ix] = 'b';
}

int main(void)
{
    char upper[0x10];
    char lower[0x10];

    write_buf(0xfffffff7ff40, 0xfffffff7ff40-0xfffffff7ff50) "\t\021", ix=16
}
```

GDB Output

Continuing.
Program received signal SIGPROT, CHERI protection violation.
Capability bounds fault.
write_buf (buf=0xffffffff7ff40 [rwRw,0xffffffff7ff40-0xffffffff7ff50] "\t\021", ix=16)
at buffer-overflow-stack.c:13
buf[ix] = 'b';
What is CHERI anyway?

• CHERI is a processor architectural protection model
  • Composes a capability-system model with hardware and software
  • Adds new security primitives to Instruction-Set Architectures (ISAs)
  • Implemented by microarchitectural extensions to the CPU and SoC
  • Enables new security behavior in software

• CHERI mitigates vulnerabilities in C/C++ Trusted Computing Bases
  • Hypervisors, operating systems, language runtimes, browsers, ....
The CHERI Capability (128-bit)

- **Capabilities** extend integer memory addresses
- **Metadata** (bounds, permissions, ...) control how they may be used
- **Guarded manipulation** controls how capabilities may be manipulated; e.g., provenance validity and monotonicity
- **Tags** protect capability integrity/derivation in registers + memory
- All memory access is via a capability
Two key applications of the CHERI primitives

1. **Efficient, fine-grained memory protection for C/C++**
   - Strong source-level compatibility, but requires recompilation
   - Deterministic and secret-free referential, spatial, and temporal memory safety
   - Retrospective studies estimate ⅔ of memory-safety vulnerabilities mitigated
   - Generally modest overhead (0%-5%, some pointer-dense workloads higher)

2. **Scalable software compartmentalization**
   - Multiple software operational models from objects to processes
   - Increases exploit chain length: Attackers must find and exploit more vulnerabilities
   - Orders-of-magnitude performance improvement over MMU-based techniques
     (<90% reduction in IPC overhead in early FPGA-based benchmarks)
Memory-safe CHERI C/C++

- Capabilities used to implement all pointers
  - Implied – Control-flow pointers, stack pointers, GOTs, PLTs, ...
  - Explicit – All C/C++-level pointers and references
- Strong referential, spatial, and heap temporal safety
- Minor changes to C/C++ semantics; e.g.,
  - All pointers must have well defined single provenance
  - Increased pointer size and alignment
  - Care required with integer-pointer casts and types
  - Memory-copy implementations may need to preserve tags
CheriBSD: A pure-capability operating system

- Complete memory- and pointer-safe FreeBSD C/C++ kernel + userspace
  - **OS kernel**: Core OS kernel, filesystems, networking, device drivers, ...
  - **System libraries**: crt/cs, ld-elf.so, libc, zlib, libxml, libssl, ...
  - **System tools and daemons**: echo, sh, ls, openssl, ssh, sshd, ...
  - **Applications**: PostgreSQL, nginx, WebKit (C++), KDE
- **Valid provenance, minimized privilege** for **pointers, implied VAs**
  - Userspace capabilities originate in **kernel-provided roots**
  - Compiler, allocators, run-time linker, etc., **refine** bounds and perms
- **Trading off** **privilege minimization, monotonicity, API conformance**
  - Typically in memory management – realloc(), mmap() + mprotect()
CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment

Brooks Davis* brooks.davis@sri.com
Robert N. M. Watson† robert.watson@cl.cam.ac.uk
Alexander Richardson† alexander.richardson@cl.cam.ac.uk
Peter G. Neumann* peter.neumann@sri.com
Simon W. Moore† simon.moore@cl.cam.ac.uk
John Baldwin‡ john@araratriver.co
David Chisnall§
Jessica Clarke†
Nathaniel Wesley Filardo†

- Received best paper award at ASPLOS, April 2019
- Complete pure-capability UNIX OS userspace
- Basis for temporal memory safety, compartmentalization
- Demonstrates broad compatibility of CHERI C/C++
Provenance changes

1 char * oldbuf = buf;
2 buf = realloc(ptr, newsize);
3 if (buf != oldbuf)  
4     buflen += buf - oldbuf;

Still derived from oldbuf and now out of bounds

1 char * oldbuf = buf;
2 buf = realloc(ptr, newsize);
3 if (buf != oldbuf)  
4     buflen = buf + ( buflen - oldbuf );

Derived from buf
Non-monotonicity

• Sometimes we need to violate CHERI’s monotonicity guarantees

• Domain transition
  • Swap the contents of all registers
  • Thread switching
  • The basis of compartmentalization

• Calls to free()
  • Access to allocator metadata from bounded allocation
  • Easy in some allocators (jemalloc) and expensive in others (dlmalloc)
## CHERI C compatibility: CheriBSD Code Changes

<table>
<thead>
<tr>
<th>Area</th>
<th>Files total</th>
<th>Files modified</th>
<th>% files</th>
<th>LoC total</th>
<th>LoC changed</th>
<th>% LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>11,861</td>
<td>896</td>
<td>7.6</td>
<td>6,095k</td>
<td>6,961k</td>
<td>0.18</td>
</tr>
<tr>
<td>• Core</td>
<td>7,867</td>
<td>705</td>
<td>9.0</td>
<td>3,195k</td>
<td>5,787k</td>
<td>0.18</td>
</tr>
<tr>
<td>• Drivers</td>
<td>3,994</td>
<td>191</td>
<td>4.8</td>
<td>2,900k</td>
<td>1,174k</td>
<td>0.04</td>
</tr>
<tr>
<td>Userspace</td>
<td>16,968</td>
<td>649</td>
<td>3.8</td>
<td>5,393k</td>
<td>2,149k</td>
<td>0.04</td>
</tr>
<tr>
<td>• Runtimes (excl. libc++)</td>
<td>1,493</td>
<td>233</td>
<td>15.6</td>
<td>207k</td>
<td>989k</td>
<td>0.48</td>
</tr>
<tr>
<td>• libc++</td>
<td>227</td>
<td>17</td>
<td>7.5</td>
<td>114k</td>
<td>133k</td>
<td>0.12</td>
</tr>
<tr>
<td>• Programs and libraries</td>
<td>15,475</td>
<td>416</td>
<td>2.7</td>
<td>5,186k</td>
<td>1,160k</td>
<td>0.02</td>
</tr>
</tbody>
</table>

**Notes:**
- Numbers from cloc counting modified files and lines for identifiable C, C++, and assembly files
- Kernel includes changes to be a hybrid program and most changes to be a pure-capability program
  - Also includes most of support for CHERI-MIPS, CHERI-RISC-V, Morello
  - Count includes partial support for 32 and 64-bit FreeBSD and Linux binaries.
  - 67 files and 25k LoC added to core in addition to modifications
  - Most generated code excluded, some existing code could likely be generated
Pure-capability CheriBSD kernel

• Full UNIX operating-system kernel compiled with CHERI C
  • Roughly 2.4MLoC core kernel excluding device drivers
  • Referential safety for all explicit and implied pointers
  • Spatial safety for mappings, stack and heap allocations, globals; with sub-object bounds
  • Temporal memory safety is not yet supported, work is being planned.

• 1.4% LoC change, 7.7% files changed
  • Includes support for hybrid kernel with CheriABI userspace, which requires capability annotations for system-call arguments
  • We will have better data on a pure purecap kernel soon, stripping hybrid support, which should substantially reduce %LoC change
Pure-capability CheriBSD kernel: Vulnerabilities

- Security analysis based on retrospective vulnerability study over 22 years
- 56% of total vulnerabilities (113 of 200) are memory-safety; of these:
  - 54% mitigated through referential and spatial safety (implemented); of these, 8% of memory safety w/sub-object
  - 72% mitigated if including heap temporal memory safety (white-board design)
  - 26% unmitigated are uninitialized values; at least 5% of memory safety would likely be mitigated by LLVM stack initialization
  - Handful of unmitigated vulnerabilities: stack temporal safety, VM vulnerabilities, ...
- 1 FTE for ~2.5 years for MIPS, RISC-V, and Morello; most time on common code
- Be aware of selection bias in vulnerability discovery – e.g., KASAN finding use-after-free vulnerabilities with fuzzing, but not subobject bounds overflows
Pure-capability CheriBSD kernel: Sub-object bounds

```c
struct example {
    int ex_int;
    char ex_arr[16];
    int ex_secret;
};

// Example allocation
struct example *p;
p = malloc(sizeof(*p));

// Narrow bounds on ex_arr
char *arrp = p->ex_arr;

// Overflowing copy triggers
// bounds violation
memcpy(arrp, src, 20);
```

- Sub-object bounds are an optional compilation mode for CHERI C
- Additional protection at slightly greater friction due to `containerof()`
2021 desktop pilot study results

Developed:

- 6 million lines of C/C++ code compiled for memory safety; modest dynamic testing
- Three compartmentalization whiteboard case studies in Qt/KDE

Evaluation results:

- 0.026% LoC modification rate across full corpus for memory safety
- 73.8% mitigation rate across full corpus, using memory safety and compartmentalization

Useful observation to be made about memory safety: Not enough to address the de facto threat model of quite a few libraries ...
cheribuild: Swiss army knife of cross building

• Builds, creates disk images for, and runs CheriBSD
  • And much much more... (compilers, gdb, qemu, etc)
  • ~300 unique targets → >2300 variants
• For aarch64, amd64, mips64, and riscv64
  • Plus CHERI purecapability and hybrid variants (except amd64)
FreeBSD build system improvements

• External cross toolchain instead of integrated compiler
  • Upstreamed XCC, etc in 2013 – brooks@
  • CROSS_TOOLCHAIN wrapper in 2014 – bapt@

• NO_ROOT & DB_FROM_SRC
  • Allow installation without privilege or dependency on local user database
  • Upstreamed in 2013 – brooks@

• Cross build from Linux & MacOS
  • Use tools/build/make.py in place of make
  • Supports both internal (bootstrapped) and external toolchain
  • Upstreamed in 2020 – arichardson@

• Non-root releases from non-FreeBSD systems
  • Committed 2022 – jrtc27@
CheriBSD packages

- We provide both pure-capability (default ABI) and hybrid packages
- Separate LOCALBASE – Breaks some ports
- Separate pkg commands (and databases)
- Custom ABI strings – includes __CheriBSD_version
- Fork of the ports tree with local patches

<table>
<thead>
<tr>
<th>ABI</th>
<th># built</th>
<th>pkg command</th>
<th>LOCALBASE</th>
<th>ABI string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure-capability</td>
<td>9k</td>
<td>pkg64c</td>
<td>/usr/local</td>
<td>CheriBSD:20220828:aarch64c</td>
</tr>
<tr>
<td>Hybrid</td>
<td>24k</td>
<td>pkg64</td>
<td>/usr/local64</td>
<td>CheriBSD:20220828:aarch64</td>
</tr>
</tbody>
</table>
Port building

• Need to support multiple ways to build:
  • Native manual - testing
  • Native poudriere - testing
  • Qemu user mode in poudriere – production builds

• Cross dependency installs
  • E.g., need hybrid python for sphinx, etc
  • Supported in ports and poudriere
  • Required to move KDE builds out of cheribuild
Qemu user mode

• Emulates program execution, but makes native system calls
• Useful where adequately large and fast hardware doesn’t exist
• Stacey Son adapted rotting bsd user mode port upstream for mips64
  • Other ports followed (armv7, riscv64, aarch64)
• Konrad Witaszczyk added basic CHERI support
  • Real tag support coming soon

• Call for help: Warner Losh looking for help getting more upstream
Releases

• Aiming for twice a year
• Use the FreeBSD release build
• Modified BSDinstaller
  • Reduced dialogs (still too many non-default choices)
  • Added option to install desktop
• Multi-console installer
  • Prompt to start the installer on all consoles
• Can’t yet build release with packages from Linux
• No supported upgrade process
Memory Safety Grand challenge: Google Chromium

• “The real thing”:
  • Foundation for Google Chrome, Microsoft Edge, Microsoft Teams, Electron, ...
  • Over 35MLoC, >190 library dependencies
  • V8, an intimidatingly real language runtime
  • Code from numerous diverse origins and in countless forms of idiomatic C and C++
  • Vast wealth of past vulnerabilities to use in evaluation
  • Performance critical components
  • Memory-safety and compartmentalization objectives

• 9 staff months, most effort went into V8 adaptation
  • Hope to having it running memory safe in May 2023

• Pilot project supported by UKRI, and Google

How could we compartmentalize software at this scale?
2022.12 Morello memory-safe desktop software stack

- Roughly 30MLoC on an Arm Morello board today, with memory-safe:
  - CheriBSD kernel with DRM + Panfrost drivers
  - CheriBSD userspace with libraries, OpenSSH, ...
  - OpenGL, Wayland display server
  - Plasma, KDE base applications including Dolphin, Okular, Konsole.
  - Aarch64 CHERI/Morello-aware GDB debugger
  - 9K CheriABI packages, 24K aarch64 (“legacy”) packages; notable exclusions for language runtimes
Signs of change: Secure-by-Design and -Default

- **Memory safe programming languages (SSDF PW.6.1):** Prioritize the use of memory safe languages wherever possible. The authoring agencies acknowledge that other memory specific mitigations, such as address space layout randomization (ASLR), control-flow integrity (CFI), and fuzzing are helpful for legacy codebases, but insufficient to be viewed as secure-by-design as they do not adequately prevent exploitation. Some examples of modern memory safe languages include C#, Rust, Ruby, Java, Go, and Swift. Read NSA’s memory safety information sheet for more.

- **Secure Hardware Foundation:** Incorporate architectural features that enable fine-grained memory protection, such as those described by Capability Hardware Enhanced RISC Instructions (CHERI) that can extend conventional hardware Instruction-Set Architectures (ISAs). For more information visit, University of Cambridge’s [CHERI webpage](https://cheri.llvm.org/).

- US CISA and 10 other agencies from 7 nations endorsed memory safe languages and *Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and -Default*

- Just advisory. For now...
Questions?

Brooks Davis <brooks.davis@sri.com
SRI International