CheriABI
A pure-capability OS process environment

Brooks Davis

University of Cambridge, SRI International

CHERI Microkernel Workshop – 23 April 2016
Software deployment models

Hybrid MMU-capability models: protection and compartmentalization within virtual address spaces

Single-address-space systems are possible but not yet our focus
Aim of CheriABI: pure-capability POSIX applications

• **Goals:**
  - Recompile to get memory safety and CFI
  - Explore pure-capability process environment

• **Principles:**
  - Change implementation, not declarations
  - Pointers to C objects are *data* capabilities
    - `malloc()` allocates data references, never code
  - Page table manipulation syscalls allow (controlled) escape from C memory model
CheriABI challenges

• Managing sources of capabilities
  • Kernel/run-time linker file mappings, argv, environ
  • Language level: malloc(), stack, globals, TLS
  • Page-table mappings
    • POSIX APIs allow any page to be replaced!
• Translating capabilities and capability containing objects into capability-oblivious kernel
  • Validating translations
• Composing defacto-POSIX with memory safety
Enforcing pointer provenance, bounds, permissions, and monotonicity

Valid userspace pointer set – pointers not generated using derivation rules are not part of the valid provenance tree and should not be dereferenceable
MIPS process layout

ELF binary

Process memory
SCO i386 ABI stack

_start(char **ap, ...) {
    ...
    argc = *(long *) ap;
    argv = ap + 1;
    env = ap + 2 + argc;
    ...

_init_tls() {
    ...
    sp = (Elf_Addr *) environ;
    while (*sp++ != 0)
        ;
    aux = (Elf_Auxinfo *) sp;

    ...
CheriABI process layout

Initial DDC, PCC

- Text
- Data
- BSS
- mmap() region
- Stack

- PCC (soon)
- DDC (soon)
- heap
- stack
- argv, environ, auxargs, etc
CheriABI stack

Stack capability

- Stack capability excludes strings, etc
- Remove ps_strings
- Capabilities for each object
- argc replaced with cheriabi_execdata
- Capability to execdata passed to _start as argument

```c
struct cheriabi_execdata {
    size_t ce_len;
    int ce_argc;
    char **ce_argv;
    char **ce_envp;
    struct cheriabi_auxarg* ce_auxargs;
    struct ps_strings *ce_ps_strings;
};
```
mmap() in CheriABI

- The mmap() syscall allocates address space
  - Is a direct and indirect source of capabilities
- mmap() also replaces mappings
  - Changes the contents of capabilities
    - Takes capability argument covering page(s)
    - Will require capability permission bit
  - May upgrade permissions on capabilities!
mmap() alignment

• Compressed capabilities require alignment constraints to ensure representability

• New alignment constraints
  • MAP_ALIGNED_CHERI
  • MAP_ALIGNED_CHERI_SEAL

• Flags constrain alignment and assert that the length is representable
  • Usable in CheriABI and native code

• MAP_CHERI_NOSETBOUNDS allows unrepresentable subregions to be mapped within reservation
mmap() continued

• Pattern of PROT_NONE reservation followed by mapping sub regions requires two changes:
  • Reservation must have desired permissions for capability followed by mprotect() to remove permissions
  • Use the MAP_CHERI_NOSETBOUNDS flag to avoid potentially unrepresentable sub-regions
• New APIs required
CheriABI syscall translation challenges

• Pointers and integers differ in registers
  • Different registers on MIPS
  • Similar issues with converged register files
• Pointers may point to structures containing pointers – similar to 32-bit/64-bit compat
  • Translation required
• Variadic system calls (fcntl, ioctl, open, etc)
• Multiplexed system calls (ioctl, _umtx)
System-call argument translation spectrum

<table>
<thead>
<tr>
<th></th>
<th>Pointer Translation</th>
<th>Capability validation</th>
<th>Capabilities in kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel acts via</strong></td>
<td>Virtual address</td>
<td>Virtual address</td>
<td>Capability</td>
</tr>
<tr>
<td><strong>Kernel implements user memory model</strong></td>
<td>-</td>
<td>tag, length*, perms</td>
<td>all</td>
</tr>
<tr>
<td><strong>Issues</strong></td>
<td>Capabilities bypassed</td>
<td>Annotations must be correct. Subject to confused deputy attacks.*</td>
<td>Need to translate arguments from non-capability ABIs.</td>
</tr>
</tbody>
</table>

* No length on strings or length not bound at time of check
Syscall argument translation vs validation

**Translation**

```c
tag = cheri_gettag(argcap);
if (tag)
    ptr = cheri_ctopr(argcap, kdc);
else
    ptr = cheri_getbase(argcap) + cheri_getoffset(argcap);
```

**Validation**

```c
if (!cheri_gettag(argcap))
    return (EPROT);
if (cheri_getsealed(argcap))
    return (EPROT);
if (cheri_getperms(argcap) & reqperms) != reqperms)
    return (EPROT);
if (cheri_getlen(argcap) – cheri_getoffset(argcap) < reqlen)
    return (EPROT);
ptr = cheri_ctoptr(argcap, kdc);
```
Syscall annotations drive validation

```c
int open(_In_z_ char *path, int flags, int mode);

int cheriabi_readv(int fd, _Inout_updates_(iovcnt)
    struct iovec_c *iovp, u_int iovcnt);

int minherit(_Pagerange_(len) void *addr,
    size_t len, int inherit);

int accept(int s,
    _Out_writes_bytes_(*anamelen)
    struct sockaddr *name,
    _Inout_opt_ socklen_t *anamelen);
```
CheriABI results

- CheriBSD tests run and largely pass
- Variety of applications work without modification
  - From echo to ssh
  - Spatial memory safety for “free”
- Progressing toward multi threaded, dynamically linked applications
  - Using modern, thread-aware malloc()
  - Starting work on runtime linker
CheriABI conclusions

• CheriABI runs (mostly) unmodified POSIX applications with memory safety and CFI

• We are exploring transition paths:
  • From a monolithic, capability-oblivious kernel to a capability-aware kernel.
  • From traditional C to memory-safe C programming environment
BACKUP SLIDES
## Kernel Changes

<table>
<thead>
<tr>
<th>Component</th>
<th>File</th>
<th>Lines +</th>
<th>Lines -</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headers</td>
<td>19</td>
<td>1424</td>
<td>11</td>
</tr>
<tr>
<td>CHERI initialization</td>
<td>2</td>
<td>49</td>
<td>4</td>
</tr>
<tr>
<td>Context management</td>
<td>2</td>
<td>392</td>
<td>10</td>
</tr>
<tr>
<td>Exception handling</td>
<td>3</td>
<td>574</td>
<td>90</td>
</tr>
<tr>
<td>Memory copying</td>
<td>2</td>
<td>122</td>
<td>0</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>5</td>
<td>398</td>
<td>27</td>
</tr>
<tr>
<td>Object capabilities</td>
<td>2</td>
<td>883</td>
<td>0</td>
</tr>
<tr>
<td>System calls</td>
<td>2</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>CheriABI</td>
<td>6</td>
<td>2855</td>
<td>0</td>
</tr>
<tr>
<td>Signal delivery</td>
<td>3</td>
<td>327</td>
<td>71</td>
</tr>
<tr>
<td>Process monitoring/debugging</td>
<td>3</td>
<td>298</td>
<td>0</td>
</tr>
<tr>
<td>Kernel debugger</td>
<td>2</td>
<td>264</td>
<td>0</td>
</tr>
</tbody>
</table>