CHERIoT

Robert Norton-Wright
Saar Amar, Tony Chen, David Chisnall,
Wes Filardo, Kunyan Liu, Hongyan Xia
Microsoft
IoT

The ‘S’ stands for security
Motivation – IoT and embedded

The IoT ecosystem:
- Includes diverse codebases
- Mostly unsafe C/C++
- Mitigations are rare

Rewriting in safe languages has challenges:
- Expensive
- Talent shortage
- Risk of introducing bugs
Much embedded code is intrinsically unsafe

- Memory allocators
- Schedulers
- Device interface code
Some things work to our advantage

- Full control of software
  - Break compatibility, drop hybrid mode, simplify ISA
- Very fast tightly coupled memory
  - Enables new temporal safety mechanism
CHERIoT shrinks metadata to 32 bits

### Bounds
- No guaranteed out-of-bounds range

### Sealing
- Only 3 bits of sealing type
- Separate code and data sealing spaces

### Permissions
- 12 permissions in 6 bits
And we add things

| Transitive permissions             | • Permit-load-mutable, deep immutability  
|                                  | • Permit-load-global, deep no-capture    |
| Interrupt control via sentries    | • Jumping to these enables / disables interrupts |
| Temporal safety via a hardware revocation bitmap | • 1 bit per 8 bytes in a separate SRAM bank |
Hardware load barrier adds temporal safety

- Load pointer computes the base address
- Looks up the corresponding revocation bit
- Invalidates the pointer if the memory is freed

```c
void *x = malloc(42);
// Print the allocated value:
Debug::log("Allocated: {}", x);
free(x);
// Print the dangling pointer
Debug::log("Use after free: {}", x);
```

Valid bit cleared, any attempt to use as a pointer will trap
Baseline security guarantees

No pointer injection
No bounds violations
No use after free

The system can assume these for building higher-level abstractions.
Compartments are code and data

- Code
- Program Counter (read/execute)

- Globals
- Global Pointer (read/write/global)
Compartments are code and data and exports.
Compartments are code and data and exports
From unforgeable pointers to compartments

Registers
- PC
- SP
- GP

Thread 1 Stack

Memory
- Compartment A code
- Compartment A globals
From unforgeable pointers to compartments

Registers
- PC
- SP
- GP
- A0

Memory
- Compartment A code
- Compartment B code
- Compartment A globals
- Compartment B globals

Thread 1 Stack
From unforgeable pointers to compartments
From unforgeable pointers to compartments

Registers
- PC
- SP
- GP
- A0

Memory
- Compartment A code
- Compartment B code

Thread 1 Stack
- Thread 1 Stack (B’s subset)

Compartment A globals

Compartment B globals
Security guarantees across compartments

No sharing except via explicit pointer passing

Pointers from the caller may prevent modification or capture
Trusted (privilege-separated) components

Loader
- Has full access to all memory
- Erases itself after boot
- Not needed if flash can store tags

Switcher
- Can see state from multiple threads and compartments
- Has access to a reserved register (and system registers)
- Around 300 instructions

Scheduler
- Trusted for availability
- No access to suspended thread state (registers or stack)

Memory allocator (optional)
- Sets bounds / revocation state on allocations
Add compartmentalization to C/C++

// Declaration adds an attribute to indicate
// the compartment containing the implementation
void __attribute__((cheri_compartment("kv_store_sdk"))) publish(char *key, uint8_t *buffer, size_t size);

// Call site looks like normal C.
// Compiled to a direct call in compartments build with
// -cheri-compartment=kv_store_sdk
// Compiled to a cross-domain call in all other cases.
uint8_t buffer[BUFFER_SIZE];
publish("key_id", buffer, sizeof(buffer));
Linker reports
What can we statically audit?
Everything in this talk is open source

https://aka.ms/cheriot-tech-report

<table>
<thead>
<tr>
<th>Item</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ISA specification:</td>
<td><a href="https://github.com/microsoft/cheriot-sail">https://github.com/microsoft/cheriot-sail</a></td>
</tr>
<tr>
<td>The reference core:</td>
<td><a href="https://github.com/microsoft/cheriot-ibex">https://github.com/microsoft/cheriot-ibex</a></td>
</tr>
<tr>
<td>The embedded OS:</td>
<td><a href="https://github.com/microsoft/cheriot-rtos">https://github.com/microsoft/cheriot-rtos</a></td>
</tr>
<tr>
<td>The compiler (cheriot branch):</td>
<td><a href="https://github.com/CTSRD-CHERI/llvm-project/">https://github.com/CTSRD-CHERI/llvm-project/</a></td>
</tr>
</tbody>
</table>
Thanks

• UKRI / DSbD / CHERITech
• All prior CHERI work we’ve built on / inspired us:
  • CHERI-RISCV Arch + LLVM
  • CompartOS (Almetary)
  • CheriOS(Esswood)
  • CHERI-RTOS (Xia)
  • Sail
  • Ibex / ETH Zurich / LowRISC
  • …
Summary

- Fine-grained spatial and temporal memory safety guarantees for C/C++
- Lightweight compartments
- Safe bounded cross-compartment sharing
- Strong attestation over compartment structure

Any more questions, please ask in the GitHub Microsoft/CHERIoT-RTOS Discussions! https://github.com/microsoft/cheriot-rtos/discussions/categories/q-a
Backup
Most codebases require very few changes

<table>
<thead>
<tr>
<th>Codebase</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microvium embedded JavaScript interpreter</td>
<td>• No changes</td>
</tr>
<tr>
<td>TPM reference stack</td>
<td>• No changes for memory safety</td>
</tr>
<tr>
<td></td>
<td>• Small changes (&lt;10 LoC) for RISC-V</td>
</tr>
<tr>
<td></td>
<td>• One line changed to run in a compartment</td>
</tr>
<tr>
<td>FreeRTOS network stack</td>
<td>• No changes for memory safety</td>
</tr>
<tr>
<td></td>
<td>• Annotations for cross-compartment calls</td>
</tr>
<tr>
<td></td>
<td>• Explicit sealing and unsealing</td>
</tr>
<tr>
<td></td>
<td>• Small changes (~100 LoC) to run without disabling interrupts for mutual exclusion</td>
</tr>
<tr>
<td>mBedTLS</td>
<td>• No changes for memory safety</td>
</tr>
<tr>
<td></td>
<td>• Small changes for compartmentalisation</td>
</tr>
</tbody>
</table>
## Capability format

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>22</th>
<th>18</th>
<th>17</th>
<th>9</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>p’6</td>
<td>otype’3</td>
<td>E’4</td>
<td>B’9</td>
<td>T’9</td>
<td>a’32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **R**: a reserved bit, which is zero in the root capabilities (and hence all tagged capabilities), but may be set if untagged data is loaded into a register. In this case its value must be preserved. This is very important because memory copies are performed with capability load a store instructions in order to preserve the tag on any capabilities present, meaning these instructions must also faithfully copy arbitrary untagged data.

- **p**: a 6-bit compressed permissions field (see Section 7.13.1)

- **otype**: a 3-bit ‘object type’ used for sealing capabilities (see Section 7.13.2)

- **E**: a 4-bit exponent used for the bounds encoding (see Section 7.13.3)

- **B**: a 9-bit base used for the bounds encoding (see Section 7.13.3)

- **T**: a 9-bit top used in the bounds encoding (see Section 7.13.3)

- **a**: the 32-bit address of the capability
Permission encoding

- **Memory cap-read-write**: GL 1 1 SL LM LG. Implicit: LD, MC, SD
- **Memory cap-read-only**: GL 1 0 1 LM LG. Implicit: LD, MC
- **Memory cap-write-only**: GL 1 0 0 0 0. Implicit: SD, MC
- **Memory data-only**: GL 1 0 0 LD SD. Implicit: None
- **Executable**: GL 0 1 SR LM LG. Implicit: EX, LD, MC
- **Sealing**: GL 0 0 U0 SE US. Implicit: None