CAPSTONE: A Capability-based Foundation for Trustless Secure Memory Access

32nd USENIX Security Symposium

Jason Zhijingcheng Yu, Conrad Watt*, Aditya Badole, Trevor E. Carlson, Prateek Saxena

National University of Singapore
University of Cambridge*
## World of Security Extensions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer Integrity</td>
<td>ARMv8</td>
</tr>
<tr>
<td>Spatial Memory Safety</td>
<td>Intel MPK, x86/64 DEP/NX, Intel MPX, RISC-V/ARM CHERI</td>
</tr>
<tr>
<td>Temporal Memory Safety</td>
<td>[None]</td>
</tr>
<tr>
<td>Concurrent Thread Safety</td>
<td>Intel TSX – Transactional Synchronization Extensions</td>
</tr>
<tr>
<td>Intra-process Sandboxing</td>
<td>Intel SGX, x86 Segmentation</td>
</tr>
<tr>
<td>Process Sandboxing</td>
<td>x86/64 Privilege Rings</td>
</tr>
<tr>
<td>Virtualization</td>
<td>AMD SEV, Intel VT-x, Intel TDX, ARM CCA</td>
</tr>
<tr>
<td>Red-Green Secure Worlds</td>
<td>ARM TZ, Intel TXT</td>
</tr>
<tr>
<td>Nested / App Virtualization</td>
<td>Intel VT-x, Intel SGX</td>
</tr>
</tbody>
</table>
Problems with Security Extensions

1. Unreliable availability of security features

2. Poor interoperability for multiple security goals

The processor has deprecated the following technologies and they are no longer supported:

- Intel® Memory Protection Extensions (Intel® MPX)
- Branch Monitoring Counters
- Hardware Lock Elision (HLE), part of Intel® TSX-NI
- Intel® Software Guard Extensions (Intel® SGX)
- Intel® TSX-NI
- Power Aware Interrupt Routing (PAIR)


Problems with Security Extensions

I. Unreliable availability of security features

- 1. Unreliable availability of security features
- 2. Poor interoperability for multiple security goals

Is there a **unified foundation** for multiple security goals?


Traditional Architectures Rely on Access Control

![Diagram showing traditional access control]

- Trust
- App
- App
- App
- OS kernel
- Hypervisor
- MMU/MPU
- Physical memory
- Allow/disallow
Traditional Architectures Rely on Access Control

Relies on explicit security policies
Assumes a central trusted authority → limiting in expressiveness

Can we make memory access trustless?
Contributions

**Unified Foundation for Trustless Memory Access**

- Minimal set of properties
- **P1: Exclusive Access**
- **P2: Revocable Delegation**
- **P3: Extensible Hierarchy**
- **P4: Secure Domain Switching**

**CAPSTONE**

- Pointer Integrity
- Spatial Memory Safety
- Temporal Memory Safety
- Concurrent Thread Safety
- Intra-process Sandboxing
- Process Sandboxing
- Virtualization
- Red-Green Secure Worlds
- Nested / App Virtualization
Threat Model: Benign Scenario

Domain A

Physical memory

Domain B

$t_0$

A invokes B

Domain A

$X$

Domain B

$t_1$

B returns

Domain A

$X$

Domain B

$t_2$

Time
Threat Model: Malicious Scenario

A invokes B

Physical memory

Domain A

Domain B

$t_0$

$t_1$

Secret leakage
Broken integrity
TOCTTOU

Time
Threat Model: Malicious Scenario

1. At time $t_0$, Domain A invokes Domain B.

2. At time $t_1$, Domain B returns.

3. At time $t_2$, Domain B is marked as compromised due to Denial-of-service attack.
Minimal set of properties for a unified foundation
Property 1: Exclusive Access

Property 2: Revocable Delegation

Property 3: Extensible Hierarchy

I can also delegate access to other domains!

Property 4: Secure Domain Switching

Properties for a Trustless Unified Foundation

P1: Exclusive Access
P2: Revocable Delegation
P3: Extensible Hierarchy
P4: Secure Domain Switching

How to enforce those properties through a unified interface?
Architectural Capabilities: A Baseline

Physical Memory

:= (cursor, base, end, perms, ...)

Capability

LD/ST addr, ...
LD/ST ..., ...

Unforgeability

Minting

op

Monotonicity

R. N. M. Watson et al., ‘Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)’. 
Enforcing Property 1: Exclusive Access

Physical memory

$\rightarrow$

A delegates memory to B

$t_0$

Domain A

Domain B

Capacity

Domain A

Domain B

Domain A

Domain B

$t_1$

$t_2$

A delegates same memory to C

Time

Domain C
Enforcing Property 1: Exclusive Access

We need something more to enforce exclusive access!
Exclusive Access: Linear Capabilities

Linear Capability Operations

- **Move**
  - **Loc A**
  - **Loc B**
  - $t_0$ to $t_1$
  - move

- **Delinearize**
  - $t_0$ to $t_1$
  - delinearize
Memory Delegation with Linear Capabilities

At $t_0$, Domain A splits capability from Physical memory to Domain B.

At $t_1$, Domain A delegates capability to B.

At $t_2$, I want B to return the capability!
Enforcing Property 2: Revocable Delegation

1. **Domain A** mints a revocation capability at time $t_1$.
2. Domain A delegates capability to B at time $t_2$.
3. Domain A performs revocation at time $t_3$.
4. Domain B performs revocation at time $t_4$.

Key types:
- **Non-linear capability**
- **Linear capability**
- **Revocation capability**
Problem: Secret Leakage Can Happen

Physical memory

Domain A

Domain B

Domain A

Domain B

Domain A

Domain B

\( t_3 \) \n
\( t_4 \) \n
\( t_5 \)

Non-linear capability

Revocation capability

Linear capability
Problem: Secret Leakage Can Happen

How to prevent secret leakage while allowing revocation?
Solution: Uninitialized Capabilities

Physical memory

Domain A

Domain B

$t_3$

B writes secrets

Domain A

Domain B

$t_4$

A performs revocation

Domain A

Domain B

$t_5$

write-only

Non-linear capability

Revocation capability

Linear capability

Uninitialized capability

Time
Properties for a Trustless Unified Foundation

$P1$: Exclusive Access

$P2$: Revocable Delegation

$P3$: Extensible Hierarchy

$P4$: Secure Domain Switching

Please see paper!
CAPSTONE: Putting It Together

ISA with capability types and instructions

Sealed-return
Sealed

Revocation
Linear
Non-linear

Uninitialized

Sealed
Seal

retseal
mint
rev

revoke
revoke

delinearize
initialize

call
Implementation and Evaluation
Functional Prototype

Case studies
- Full memory safety (Rust-like semantics)
- Untrusted memory allocator
- Untrusted scheduler
- Nestable enclaves

C-like code → CapstoneCC → CAPSTONE instructions → CapstoneEmu → Output

Machine configurations
Case Study: Memory Safety (Rust-like Semantics)

- Spatial Memory Safety
- Temporal Memory Safety
- Concurrent Thread Safety

Architectural capabilities

Linear capabilities + revocation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Rust semantics</th>
<th>CAPSTONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td><code>let a = b;</code></td>
<td><code>mov ra, rb;</code></td>
</tr>
<tr>
<td>Immutable borrow</td>
<td><code>let a = &amp;b;</code></td>
<td><code>mrev rr, rb; delin rb; li r0, 0; tighten rb, r0; mov ra, rb; (use ra) revoke rr; mov rb, rr</code></td>
</tr>
<tr>
<td>Mutable borrow</td>
<td><code>let a = &amp;mut b;</code></td>
<td><code>mrev rr, rb; mov ra, rb; (use ra) revoke rr; mov rb, rr</code></td>
</tr>
</tbody>
</table>
Case Study: Trustless Memory Allocator

- Allocatable memory
- Allocated memory
- Allocator code
- Allocator data

- Non-linear capability
- Linear capability
- Revocation capability
- Uninitialized capability
- Sealed capability
Case Study: Trustless Scheduler

- Thread A's context
- Scheduler code
- Thread B's context
- Scheduler data
- Thread C's context

Non-linear capability
Linear capability
Revocation capability
Uninitialized capability
Sealed capability
Case Study: Nestable Enclaves

Physical memory

Domain A

Split, mint rev, and delinearize

Domain A

A passes capabilities to B

Domain B

Domain A

Non-linear capability

Revocation capability

Sealed capability

Linear capability

Uninitialized capability

$t_0$

$t_1$

$t_2$

Time
Case Studies

- Pointer Integrity
- Spatial Memory Safety
- Temporal Memory Safety
- Concurrent Thread Safety
- Intra-process Sandboxing
- Process Sandboxing
- Virtualization
- Red-Green Secure Worlds
- Nested / App Virtualization

- Rust-like semantics
- Trustless memory allocator
- Trustless scheduler
- Nestable enclaves

Takeaway: CAPSTONE is highly expressive
Preliminary Performance Evaluation

Results: within ~50% run time overhead
Conclusion

- **Goal:** unified foundation for trustless memory access
- **Required properties**
  - Exclusive access
  - Revocable delegation
  - Extensible hierarchy
  - Secure domain switching
- **CAPSTONE**
  - Capability-based architecture
  - Core ideas: linear capabilities, revocation, uninitialized capabilities
  - Prototype implementations with emulator, compiler, and library
  - Case studies: CAPSTONE is highly expressive

[https://capstone.kisp-lab.org/](https://capstone.kisp-lab.org/)

Thanks for listening!