ERICSSON

UNIVERSITY OF WATERLOO

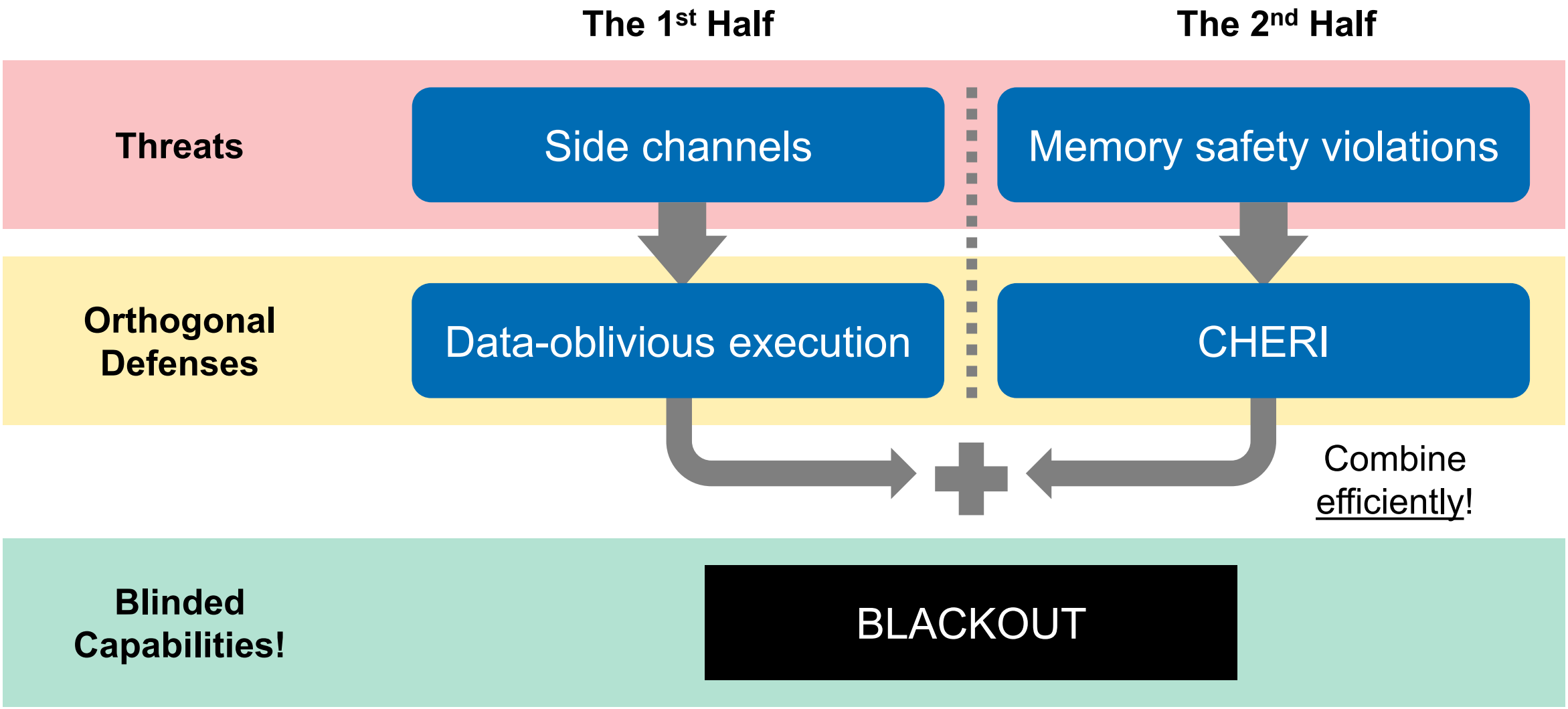
BLACKOUT
Data-Oblivious Computation with Blinded Capabilities

Hossam ElAtali[*][1], Merve Gülmez*[†][1], Thomas Nyman*[‡], *N. Asokan*[*]

** University of Waterloo* *† Ericsson Security Research* *‡ Ericsson Product Security*

[1] Joint first authors

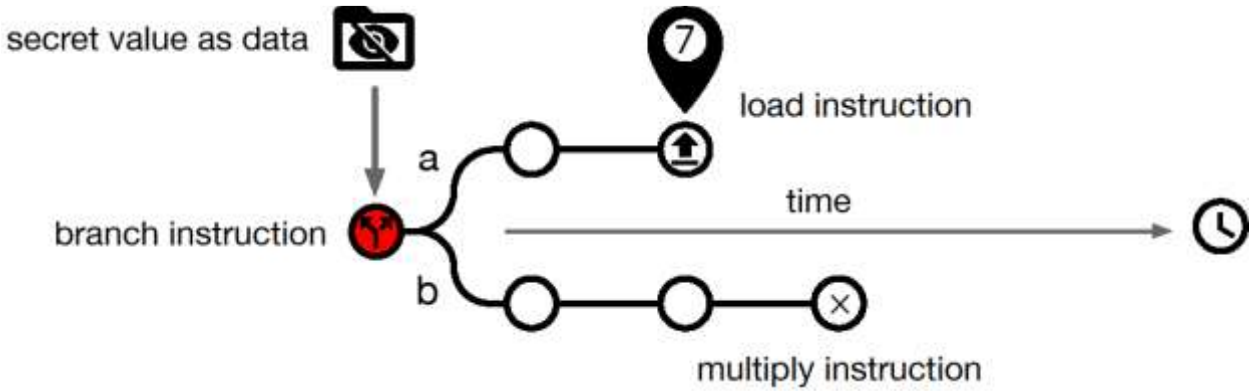
Talk in a nutshell



The 1st Half – Timing side channels

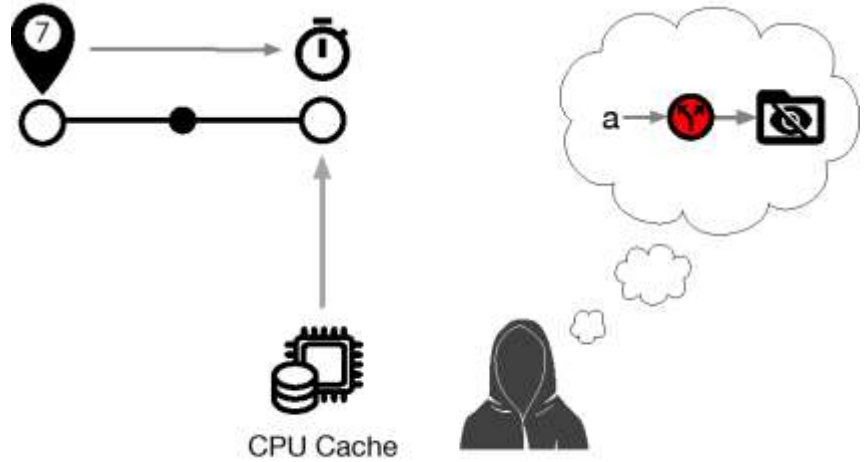
Unintended outputs of a system

- can be used to leak secrets



Timing side channels

- Secret-dependent branching and memory access can cause observable changes to control flow and cache state



Side-channel protection

Isolation techniques, e.g., cache partitioning

- Prevent observation of changes, e.g., from different process
- Do not prevent changes themselves.
- Attackers keep discovering new observation methods

Alternative solution: Data-oblivious code

- Prevents changes based on secret data

Data-oblivious code

Seemingly data-oblivious source code can still lead to side channels

- Compilers can introduce side channels into assembly code
- HW optimizations can cause side channels even with "correct" assembly
- Leakage occurs silently!

Defenses:

- Constantine[BDQG+21] – compiler transformations
  - Uses best-effort approach → confidentiality not guaranteed
- BliMe[EGLA+24]/OISA[YHEF+19] – HW taint-tracking and enforcement
  - High memory tagging overheads + difficult to program correctly

[BDQG+21] "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization", ACM CCS (2021)
[EGLA+24] "BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking", NDSS (2024)
[YHEF+19] "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing", NDSS (2019)

The 2nd Half – Memory safety violations

Memory safety violations still one of the top causes of exploits

Can lead to attacker gaining arbitrary read, write or execute privileges

Examples: out-of-bounds access, use-after-free

Defenses:

- Memory-safe language, e.g., Rust
- CHERI

Side-channel protection vs. memory safety

Current defenses for side channels and memory safety are orthogonal

Defense	Side-channel protection	Memory safety
HW taint-tracking approaches, e.g., BliMe, OISA	✓	
Cache partitioning	✓	
Constantine	✓	
Memory-safe languages, e.g., Rust		✓
CHERI capabilities		✓

Prior attempts to combine defenses have been unsuccessful

- e.g., adding side-channel protection to Rust

Side-channel protection vs memory safety

Current defenses for side channels and memory safety are orthogonal

Defense	Side-channel protection	Memory safety
HW taint-tracking approaches, e.g., BliMe, OISA	✓	
Cache partitioning	✓	
Constantine	✓	
Memory-safe languages, e.g., Rust		✓
CHERI capabilities		✓
BLACKOUT	✓	✓

Goal: CHERI memory safety + BliMe-like side-channel protection

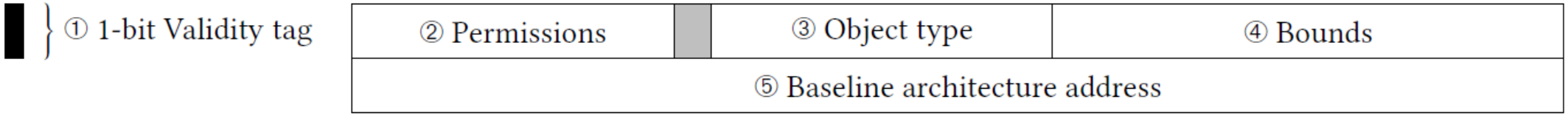
Naïve BliMe+CHERI:

- doubles memory tagging → high overheads, even for non-secret workloads

Background – CHERI

Pointers to code and data replaced by “capabilities”

- Contains metadata storing bounds and permissions
- Memory tagging used to store validity bit



Provenance and monotonicity prevent capability forgery

- Provenance: valid capabilities can only be derived from other valid ones
- Monotonicity: capabilities cannot “gain” permissions

BLACKOUT overview

HW propagates blindedness and prevents "leaky" operations, e.g.:

- Blindedness is taint that denotes secret data

Introduces blinded capabilities (BCs)

- Have exclusive access to blinded data in memory
- Data loaded with BCs is marked in registers with blindedness bit

Compiler & SW support guides developer towards data-oblivious code

- Compiler generates BCs for stack and analyzes code for leakage detection
- CheriBSD and blinded malloc clear blinded data on revocation

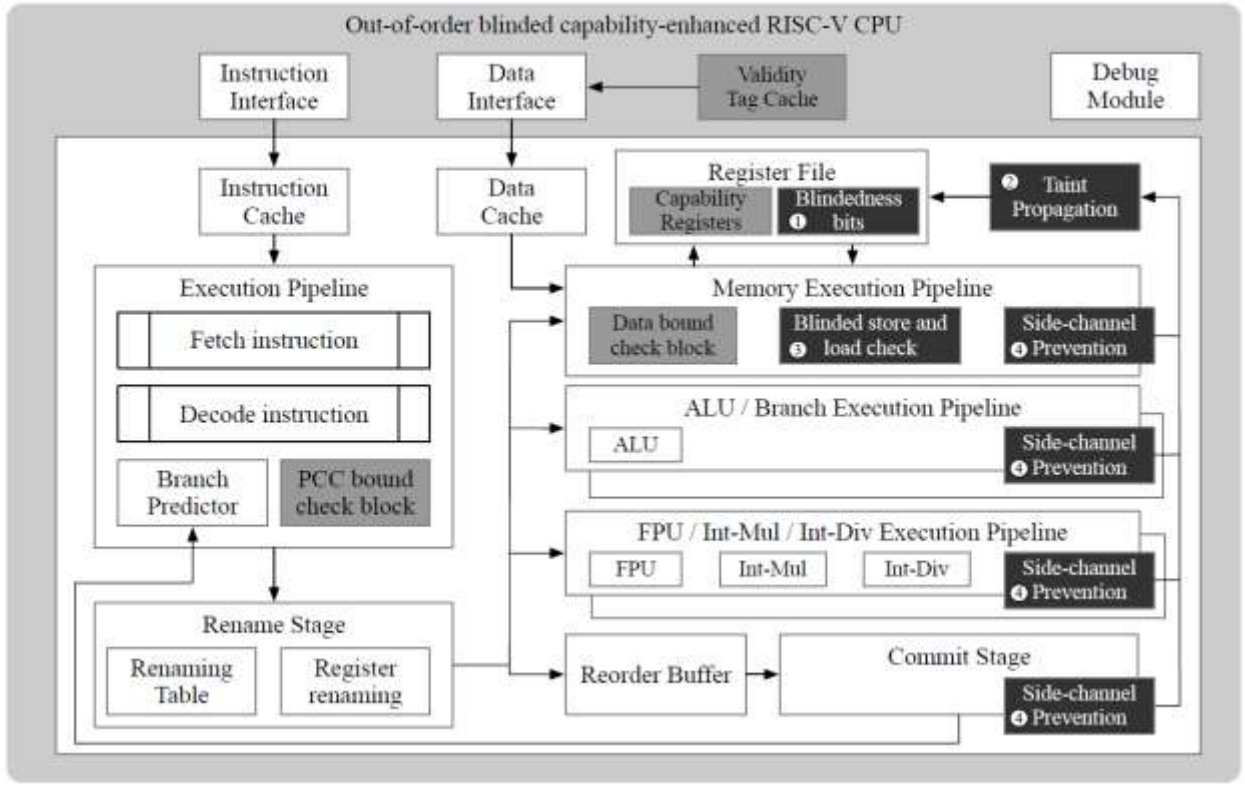
BLACKOUT hardware

Extend registers with BliMe-like blindedness bit

HW propagates blindedness

HW enforces data-oblivious operation on blinded operands

- Control-flow
- Loads & stores



Adds support for *blinded capabilities*

Blinded capabilities (BCs)

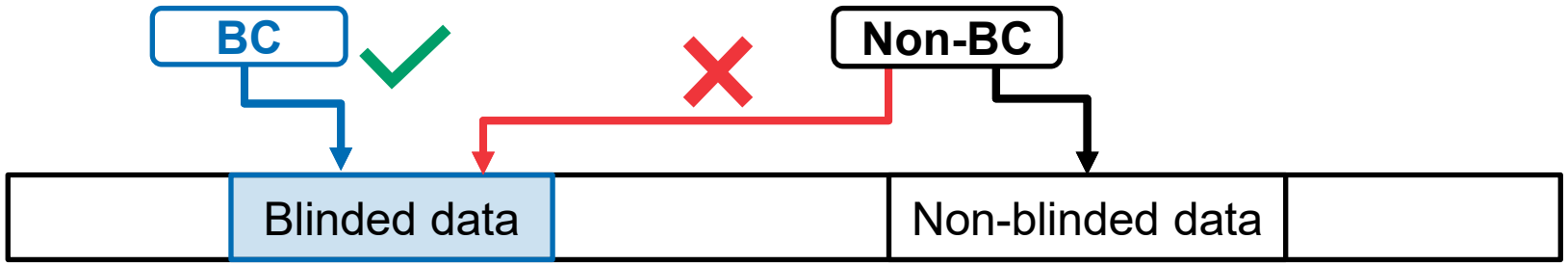
Capabilities with new ‘non-oblivious access’ permission unset

- Unsetting permission means operations on data must be data-oblivious

Data loaded using BCs is marked in registers with blindedness bit

- Avoids the need to track secret data in memory
- Memory tagging not required!

BCs guaranteed to have exclusive access to blinded data in memory



Exclusive access

Exclusive access invariants:

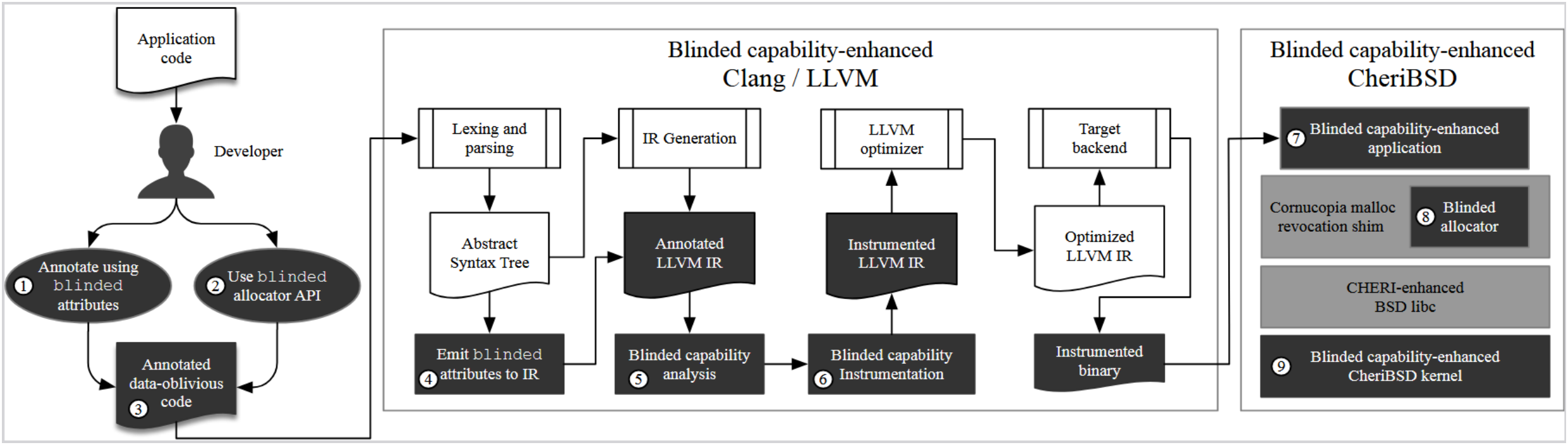
1. Blinded data cannot be stored using non-BCs ➡ enforced by HW
2. Capabilities cannot be blinded ➡ enforced by HW
3. Bounds of valid BCs and non-BCs must not overlap ➡ enforced by SW

Compiler and heap allocator ensure proper BC and non-BC bounds

- Blinded data is cleared before memory region is reused
- Compiler handles blinded local variables on stack frame pop
- Heap allocator uses Cornucopia[F+20]-style reclamation for blinded data

[F+20] "Cornucopia: Temporal Safety for CHERI Heaps", IEEE Symposium on Security and Privacy (2020)

BLACKOUT software stack



BC-enhanced compiler

Creates BCs to load/store variables annotated with ‘blinded’ attribute

```
# define __blinded [[ clang :: annotate_type (" blinded ")]]
int __blinded a;
```

Clears blinded data on stack frame pop

Analysis pass:

- tracks blinded data flows
- issues error when confident of exclusive access or side-channel violation
- enables early detection of bugs that would otherwise cause HW faults

Example 1

```
1  #define __blinded [[clang::annotate_type("blinded")]]
2
3  ❶ __attribute__ ((blinded))
4  int data_oblivious_select(bool cond, int x, int y) {
5
6  ❷  bool __blinded c; // c declared blinded and
7                       // accessed via blinded capability
8
9     int res;          // res not declared blinded
10                      // but blindedness is inferred
11    ❸ Compiler infers res blinded from ❺
12
13    c = cond;    // Uses store via blinded capability
14                 // (argument already in register)
15    ❹ Compiler knows c is already blinded based on declaration
16
17    res = (x * c) + (y * (!c)); // HW propagates
18                                // blindedness to res
19    ❺ Compiler infers res is blinded from this assignment
20
21    return res;
22 }
```

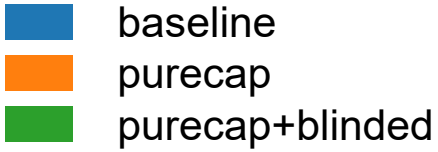
Performance Evaluation

Implemented on RISC-V CHERI Toooba core

No overhead for unblinded workloads

1.5% geomean vs. CHERI for blinded workloads

	Overhead (%)		
Benchmark			
binary_search	4.5	45.5	52.1
dnn	0.0	20.1	20.2
find_max	2.0	23.0	25.5
int_sort	-0.5	13.1	12.5
matrix_mult	1.3	9.7	11.1



Vanilla CHERI overheads due to missing LLVM support:

- High initial startup times
- Missing loop optimizations
- Baremetal experiments show much lower overheads (9.1% vs. 52.1%)

Security Evaluation

Security guarantees inherited from CHERI and BliMe

Spectre

- Vanilla CHERI-Toooba vulnerable to Spectre-BTB, -RSB and -STL[F+21]
- BLACKOUT successfully stops all Spectre attacks

Non-interference

- Data-oblivious execution inherently provides non-interference
- Verified empirically through prior work methodology – Libra[F+24]

[F+21] "*Developing a test suite for transient-execution attacks on RISC-V and CHERI-RISC-V*." Computer Architecture with RISC-V workshop (CARRV). 2021.
[F+24] "*Libra: Architectural Support for Principled, Secure and Efficient Balanced Execution on High-End Processors*," in ACM CCS (2024).

Conclusion

Combines CHERI memory safety with BliMe-like side-channel protection

Introduces blinded capabilities to access blinded data in memory

- No additional memory tagging required

HW propagates blindedness and stops "leaky" operations on blinded data

Incurs minimal performance, area and power overheads vs. baseline CHERI

Inherits security guarantees from CHERI and BliMe

- Evaluation demonstrates non-interference & protection against Spectre

blindedcapabilities.github.io

[EGNA25] "BLACKOUT: Data-Oblivious Computation with Blinded Capabilities", ACM CCS (2025) (also on arXiv)

Example

```
1  void bad_func(bool cond, int x, int *out) {
2
3  ①  {  int __blinded a = x;    // a declared blinded
4     {  int b;
```

Area & power overheads

	logic	Δ(%)	memory	Δ(%)	registers	Δ(%)	power	Δ(%)
CHERI-Toooba Core	697508	–	20852	–	412493	–	6.205	–
Blinded CHERI-Toooba Core	705863	1.2	20855	0.0	412913	0.1	6.536	5.3

Low overheads result of minimal changes to CHERI Toooba

Overheads caused by:

- Taint-tracking logic
- Violation-checking logic
- Extension of registers with blindedness bit