# BLACKOUT: Data-Oblivious Computation with Blinded Capabilities

Hossam ElAtali*
*University of Waterloo*
Waterloo, Canada
hossam.elatali@uwaterloo.ca

Merve Gülmez*
*Ericsson Security Research*
Kista, Sweden
merve.gulmez@ericsson.com

Thomas Nyman
*Ericsson Product Security*
Kista, Sweden
thomas.nyman@ericsson.com

N. Asokan
*University of Waterloo*
Waterloo, Canada
asokan@acm.org

## Abstract

Lack of memory-safety and exposure to side channels are two prominent, persistent challenges for the secure implementation of software. Memory-safe programming languages promise to significantly reduce the prevalence of memory-safety bugs, but make it more difficult to implement side-channel-resistant code. We aim to address both memory-safety and side-channel resistance by augmenting memory-safe hardware with the ability for data-oblivious programming. We describe an extension to the CHERI capability architecture to provide *blinded capabilities* that allow data-oblivious computation to be carried out by userspace tasks. We also present BLACKOUT, our realization of blinded capabilities on a FPGA softcore based on the speculative out-of-order CHERI-Toooba processor and extend the CHERI-enabled Clang/LLVM compiler and the CheriBSD operating system with support for blinded capabilities. BLACKOUT makes writing side-channel-resistant code easier by making non-data-oblivious operations via blinded capabilities explicitly fault. Through rigorous evaluation we show that BLACKOUT ensures memory operated on through blinded capabilities is securely allocated, used, and reclaimed and demonstrate that, in benchmarks comparable to those used by previous work, BLACKOUT imposes only a small performance degradation (1.5% geometric mean) compared to the baseline CHERI-Toooba processor.

## Keywords

Capabilities, constant-time code, data-oblivious code, hardware implementation, side-channel analysis, transient-execution attacks

## 1 Introduction

Weak memory-safety and side-channel leakage are two significant security challenges for security-critical software, such as cryptographic libraries. Lack of memory-safety in programming languages such as C and C++, is one of the oldest, most persistent problems in computer security. The urgency of addressing the impact of memory-unsafe code at scale has grown under recent regulatory scrutiny [73] leading cybersecurity authorities like US CISA [20] and UK NCSC [70] to advocate for memory-safe languages like Rust [54], and memory-safe hardware, such as Capability Hardware Enhanced RISC Instructions (CHERI) [90] as potential ways to eliminate entire classes of memory-safety vulnerabilities. Rust is poised to benefit from increased uptake thanks to the advocacy from regulators, but has not yet been proven in many industry sectors, and a complete rewrite of the vast body of existing C/C++ code in use today at a grand scale is economically impossible.

*Constant-time code* provides security against timing side-channels by preventing attackers from inferring secret data by observing the timing of operations and instructions, cache effects, etc. Writing constant-time code by hand is hard, evident from the many flaws discovered in production side-channel-resistant code [12]. Consequently, industry recommendations urge all but the most specialized developers to refrain from developing constant-time code [47] because even slight mistakes can lead to exploitable side channels. For example, central processing units (CPUs) today do not treat control flow as a secret, thus allowing secret-dependent control-flow to leak sensitive information through the timing of operations or their effects on caches in a complex, hard-to-control, but observable way. Optimizing compilers can transform code intended to be constant-time into *functionally equivalent* (in terms of inputs and output) machine instructions but, for instance, branch on a secret value [42, 85]. Similarly, hardware optimizations like speculative execution that are transparent to software can be exploited to leak secret data through existing side channels [56, 60]. Unlike memory-safety defects, which manifest as crashes or other faulty program behavior once a bug is triggered, flaws in constant-time programming *fail silently* and might be detected only after successful exploitation, if at all.

This problem is exacerbated in languages such as Rust which employ higher-level abstractions than C while still being compiled down to highly-optimized machine code. Several unsuccessful attempts have been made to introduce constant-time-programming abstractions to the core Rust language [8, 53, 58], leaving developers with few viable approaches for constant-time Rust programming. Experts implementing Rust libraries ("crates") providing primitives with constant-time properties acknowledge that such efforts are fundamentally limited because side-channel resistance is not a property of the software alone, but that of a system comprising both software and hardware [1]. A complete solution would require substantial changes to the compilers to allow severely restricting optimization on variables containing secret information, incurring significant cost to performance. We discuss the alternatives employed in real-world cryptographic libraries in §9.

Memory-safe hardware, such as CHERI, provides an alternative way to ensure memory-safety properties, particularly in systems where C and C++ will remain the dominant languages for the foreseeable future. CHERI effectively addresses memory safety, but does not inherently protect against side-channel attacks. Previous work to harden CHERI against transient execution attacks [36] does not protect against non-transient side-channel attacks. On the other hand, inherent resistance against side channels would also be effective against transient execution attacks that employ such side channels for inferring their effects on microarchitectural properties like cache state. In principle, CHERI-capable hardware could be complemented with existing data-oblivious computation solutions like OISA [102] or BliMe [30] that protect against side channels. However, these approaches introduce considerable performance

*Both authors contributed equally to this research.

overhead because they require the addition of hardware-managed tag bits to memory. The adoption of hardware-accelerated memory tagging is hampered by significant meta-data storage and memory traffic overheads, since tags must be managed for every piece of data in memory.

**This paper and contributions.** In this work, we propose Blinded Architectural Capabilities and Kernel for Oblivious Userspace Tasks (BLACKOUT), an extension to the CHERI memory-safety architecture. It provides novel *blinded capabilities* to augment CHERI with hardware-enforced taint tracking to guarantee confidentiality of secret data against conventional and speculative side channels. Unlike prior approaches, blinded capabilities avoid the need for additional tag bits in memory (beyond those employed by the baseline CHERI design). Inside the CPU, registers are extended with a *blindedness* bit, which is set when the register is loaded with secret data. Arithmetic logic unit (ALU) operations incorporate taint tracking, propagating blindedness bits from operands to destination registers; therefore, any data derived from secret data is also marked as secret. Instructions operating on such data in registers ensure that secret data is only written to memory through blinded capabilities which have exclusive-access to "blinded" areas of memory. Crucially, any attempt to misuse secret data—such as affecting control flow or as an address in memory operations—results in a fault, effectively preventing side-channel leakage.

We also present a *data-oblivious programming model* for CHERI C which, aided by our blinded-capability-aware Clang/LLVM compiler helps programmers in writing constant-time code for BLACKOUT. Our changes to the LLVM compiler infrastructure are *non-invasive*, consisting of compiler passes that do not interfere with existing compiler optimizations, but instruct BLACKOUT hardware about blinded variables. BLACKOUT allows developers to write constant-time code with minimal additional annotations, benefit from compile-time diagnostics, and turn previously silent constant-time bugs into explicit errors reported through CHERI's exception mechanism.

To demonstrate the practicality of our approach, we realize blinded capabilities on the CHERI-RISC-V architecture on a field-programmable gate array (FPGA) softcore based on the speculative out-of-order CHERI-Toooba processor and integrate blinded capabilities into the CheriBSD operating system (OS) and software stack. We evaluate the area and performance overheads using data-oblivious benchmarks from seminal works in data-oblivious computing for comparability. BLACKOUT offers **the first unified solution to memory safety and side-channel confidentiality with minimal overhead**, addressing key limitations of existing methods. In summary, our contributions are:

- A hardware-software co-design for *blinded capabilities* that extends CHERI with the ability to carry out data-oblivious computation in userspace tasks (§5).
- BLACKOUT: a realization of blinded capabilities on a CHERI-RISC-V FPGA softcore based on the speculative out-of-order CHERI-Toooba processor IP (§6.1).
- A programming model and software stack for blinded capabilities and support for BLACKOUT in the CHERI-enabled Clang/LLVM compiler and CheriBSD OS, thus enabling a broad class of applications to benefit from BLACKOUT (§6.2).
- Evaluation of BLACKOUT using the CoreMark industry-standard performance benchmark and data-oblivious benchmarks from prior work, showing minimal performance impact on data-oblivious code compared to the baseline CHERI-RISC-V processor (1.5%), and moderate impact compared to a processor with neither CHERI's memory-safety nor BLACKOUT enforcement (23.5%) (§7).

Source code artifacts for the BLACKOUT hardware and software stack are available at https://github.com/blindedcapabilities.

## 2 Background

### 2.1 The CHERI capability architecture

CHERI is an instruction-set architecture (ISA) extension that integrates a capability-based hardware-software co-design for memory protection. It extends conventional ISAs with hardware-supported capabilities to verify memory accesses via code or data pointers. The CHERI ISA specification [91] defines how capabilities are represented in registers and memory and offers capability-aware instructions to manipulate them. Current implementations of CHERI support the RISC-V and Arm8-A instruction sets with CHERI-enabled processors developed by Arm [43], Microsoft [5] and companies and universities in the RISC-V ecosystem.

Figure 1 illustrates the in-memory representation of a CHERI capability. Each capability is double the width of the native pointer type: 128 bits on 64-bit platforms and 64 bits on 32-bit platforms. A single-bit *validity tag* ①, stored separately, ensures integrity by invalidating capabilities manipulated by non-capability-aware instructions. Capability-aware instructions preserve tags as long as the operation on a capability is valid but prevent unauthorized manipulation and injection of arbitrary capabilities. These include the store capability via capability (csc) and load capability via capability (clc) instructions which are used to store, respectively load, capabilities to and from locations in memory identified by a capability operand. In a CHERI-enhanced architecture, address and general-purpose CPU registers are extended to store the full capability representation. For example, in CHERI-RISC-V, all general-purpose registers are 128-bit *capability registers*; the program counter (PC) and stack pointer (SP) are represented by program counter capability (pcc) and stack pointer capability (csp) registers, respectively. The capabilities themselves include several fields:

- **Permissions** (②) define allowed operations.
- **Object type** (③) enables temporary "sealing", rendering a capability unusable until it is "unsealed" by a special instruction. This enables opaque pointer types and fine-grained in-process isolation.
- **Bounds** (④) specify the accessible memory range relative to the baseline architecture address (⑤). They are stored in a compressed format [99] to reduce memory footprint, but require stricter alignment on larger object allocations.

New capabilities are always derived from an existing capability, with their lineage traceable to initial boot-time capabilities. CHERI enforces *monotonicity* ensuring newly created capabilities cannot exceed the permissions or bounds of their parent. The candperm instruction allows the permissions of a capability to be dropped according to a given mask, but not gained. Controlled exceptions, such as sealed capabilities for exception handling and compartmentalization, allow limited non-monotonicity.

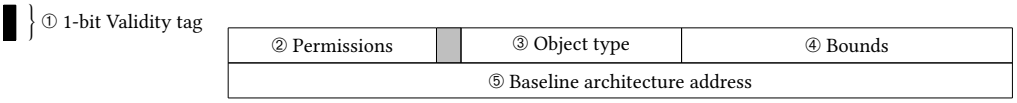



**Figure 1: In-memory representation of CHERI capabilities adapted from Watson et al. [90]**

**Spatial- and temporal-safety enforcement.** The bounds, stored with the virtual address, underpin CHERI's spatial memory safety. Each memory allocation is associated with a capability describing its valid address range and access permissions. This enables inherent spatial memory-safety properties. To provide temporary safety for heap-based allocations, CHERI requires a capability revocation mechanism, such as Cornucopia [32]. Cornucopia scans for capabilities pointing to freed memory, allowing such stale capabilities to be revoked. Extensions to the CHERI software and hardware have also explored sandboxing [19], and initialization safety [41, 45].

**CHERI-enabled software stack.** CheriBSD [25] is a modified version of the open-source FreeBSD operating system, designed to support CHERI-RISC-V both in emulation and on hardware. It is a fully functional OS prototype, demonstrating how CHERI support can be integrated into a conventional OS design. The CheriBSD kernel and userspace can be built in "*pure-capability*" CHERI C/C++ which means all conventional memory pointers are replaced by corresponding capabilities by the CHERI-LLVM compiler. In pure-capability mode, compiler-generated code derives bounded capabilities for automatic (stack-allocated) variables from the `csp`. For global variables, the run-time linker populates into a capability-aware global offset table (GOT), also referred to as the *captable*. Capabilities for heap-allocated data is handled by Cornucopia which is integrated into CheriBSD via a *shim layer*, known as the `malloc` revocation shim (MRS), which sits on top of the underlying BSD libc's memory allocator. Consequently, CheriBSD enforces both spatial and temporal memory safety for CHERI-enabled software.

## 2.2 Side-Channel Leakage

Side channels are unintended outputs of a system that can leak information about its behavior. CPU side channels are caused by implementation details, often the result of performance optimizations, such as caching mechanisms and speculative execution. Attackers can extract sensitive information by observing variations in execution time, power consumption, or electromagnetic emissions that are inadvertently output by these microarchitectural features. In this paper, we focus on timing side channels as they are the most commonly used in remote attacks.

A prominent example of timing side channels is cache timing, which leaks information by measuring cache access latency. By comparing the latency to a predetermined threshold, software can determine whether an address is already cached by the CPU. Prime+Probe [74], FLUSH+RELOAD [101] and Flush+Flush [44] are example side-channel attacks using different techniques to make this timing difference dependent on secret data and extract it from the system. Transient attacks like Spectre [56] and Meltdown [60], expose new ways to access (architecturally inaccessible) secrets and leak them using side channels like cache timing. Speculative taint-tracking techniques [103], such as SpectreGuard [37], ConTExT [86], and ProSpeCT [24], mitigate Spectre-style attacks by delaying instructions dependent on speculatively loaded secrets, but mitigating side-channels in general falls on the developer.

Preventing side-channel leakage is difficult for software developers. Software is often written in high-level languages that abstract away the individual CPU instructions operating on secret data, leaving decisions to the underlying compilers and software stack. Even lower-level languages like C do not provide developers with sufficient controls to prevent side-channel leakage, as the compiler can introduce side channels in the generated assembly. Furthermore, even with seemingly side-channel-free assembly, hardware optimizations transparent to software (and often undocumented), such as speculation and out-of-order execution, can inadvertently leak secrets, as described above with the Spectre vulnerabilities. The challenge of preventing side-channel leakage is therefore present at every level of the software and hardware stack, and must be solved through hardware-software co-design.

## 2.3 Data-Oblivious Instruction Set Architectures

Data-oblivious computation refers to algorithms and software that process data in a manner where their control-flow and memory access patterns are not affected by their input data. Constant-time code, which hardens software against side-channels, relies on data-oblivious algorithms and that the CPU cycles consumed by individual hardware instructions are independent of their operands' values[1]. Historically, developers of security-critical code sensitive to side-channels, e.g., cryptographic implementations, have relied on obscure information on instruction timing differences obtained through extensive experiments on different ISAs [77]. In recent years, major processor manufacturers have began to document instructions with data-operand independent timing (DIT) [48] and incorporate DIT modes into their designs [49, 63].

The challenge of ensuring the correctness of data-oblivious software implementations, however, remains. Verifying whether a program written in a high-level language is data-oblivious has a number of challenges (§9). Chief among them is that data-obliviousness can only be defined at a machine-code level. This observation has motivated the creation of *data-oblivious ISAs* that, similar to data operand independent timing modes, provide modes of operation for CPUs to ensure that instructions operating on sensitive data do so in a data-oblivious manner. Previous works use hardware-enforced taint tracking using "*blindedness tags*" [29] and dedicated memory partitions [102]. These approaches result in best-case performance overheads from 8% to 35% (worst case involving an order of magnitude slowdown). Memory tagging exhibits poor performance as the number of tag bits increase. Schemes such as the Arm architecture's Memory Tagging Extension (MTE), used for memory-safety sanitation, limit the number of tag bits to 4. MTE further mitigates performance impact by allowing tag checks to occur asynchronously. But asynchronous tag checks are not practical

for maintaining data-obliviousness as data-oblivious ISAs must be able to prevent data accesses before confidentiality is violated.

## 3 System and Adversary Model

We assume the same system model as in CHERI. We trust the OS to prevent memory containing sensitive information from being exposed to other processes after a process exits, either normally or as a result of CHERI exceptions. We assume memory-safety properties provided by CHERI, which prevent adversaries from tampering with the program's control flow or directly inferring memory contents beyond that which are exposed during normal program operation. We also assume a DIT [63] mode is available.

We assume the same adversary model as in CHERI. In addition, we assume the adversary has the ability to observe side channels during a program's execution, possibly from another process running on the system simultaneously. In this work, we consider side-channel threats that can be mitigated by data-oblivious software, including timing side channels, such as cache timing [18, 44, 74, 101], instruction timing [78] (with DIT) and port-contention timing [4]. We also consider transient execution attacks that leak information *loaded* by mis-speculated instructions within the same address space, such as Spectre [46, 56, 57, 64], in scope, even if mis-training can be done across address spaces [14]. Attacks with transient instructions that can load data *across* address spaces, such as Meltdown and microarchitectural data sampling (including load-store-buffers) [15], and transient capability forgery, such as Meltdown-CF [36], are out of scope, but can be prevented by using Capability Speculation Contracts (CSCs) [36]. Attacks that require physical access to the system, e.g., to measure power consumption, or those through direct memory access (DMA) peripherals are out of scope.

## 4 Goals and Challenges

In this section, we outline our overarching goals and the primary challenges associated with them.

### 4.1 Goals and Primary Challenges

**Adapting hardware-enforced taint tracking to capability-based architectures.** Our primary goal is integrating data-oblivious computation with the CHERI protection model. While it may seem straightforward to implement a data-oblivious ISA on top of an CHERI-enabled architecture, there are two key challenges to overcome. First, the CHERI architecture is intended to be applied onto conventional ISAs such as RISC-V and Arm. Bolting an existing data-oblivious ISA (§ 2.3) on top of CHERI will require invasive changes to CHERI and/or the underlying architecture making real-world adoption less realistic. Second, significant changes to the architecture will negatively impact performance.

To address these challenges, we propose enhancing the existing CHERI capability model with *blinded capabilities* which ensure data accessed through them is operated on exclusively in a data-oblivious manner. Memory managed by blinded capabilities is referred to as *blinded memory*. This avoids the need to propagate blindedness tags to memory thus overcoming the drawbacks of simply integrating an existing data-oblivious ISA with CHERI [30, 102].

**Practical programming model for blinded capabilities.** Our second goal is introducing a practical programming model for blinded capabilities. CHERI-enabled languages (CHERI C/C++ [92]) enforce memory-safety properties (§ 2.1) at run-time whereas data-oblivious ISAs enforce oblivious access to secret data (§5). Thus, our model must adhere to both CHERI and data-oblivious properties.

A particular challenge arises because the CHERI compiler occasionally allows memory accesses without explicit capabilities (e.g., certain stack accesses). To ensure such accesses do not target blinded memory (which would raise a hardware fault), our blinded capability-enhanced compiler must explicitly enforce that all accesses to blinded stack variables occurs through blinded capabilities. We describe this compiler enhancement further in § 5 and § 6.2.

### 4.2 Additional Challenges

Integrating blinded memory management with the CHERI protection model introduces several additional challenges.

**Maintaining capability monotonicity.** Introducing blinded capabilities adds new privileges that must align with CHERI's existing monotonicity properties. Our design considers "non-blinded" a distinct permission, which, when removed, permanently classifies a capability as a blinded capability. This upholds monotonicity—blinded capabilities cannot be promoted to regular, non-blinded capabilities.

**Exclusive access to blinded memory.** As blinded capabilities are based on CHERI capabilities, they inherently face the *revocation problem*. Deallocated memory regions may still be accessible through residual capabilities, risking use-after-free vulnerabilities and inadvertent disclosure of blinded data. Our design addresses this by ensuring new blinded capabilities obtain exclusive access to newly allocated blinded memory inside the process. Thus, blinded memory cannot be subject to use-after-free conditions from residual, non-blinded capabilities to it.

**Securely reclaiming blinded memory.** Blinded memory must be securely reclaimed, as residual sensitive data may remain upon deallocation. Since CHERI does not inherently guarantee memory initialization safety, we need to ensure secure deallocation through automatic memory clearing with blinded capability-enhanced compiler and a blinded capability-enhanced allocator (§6.2).

## 5 Blinded Capability Design

Blinded capabilities introduce a new programming model that developers must follow to protect their sensitive data. While providing a completely unrestricted programming model can seem appealing, it 1) cannot guarantee exclusive access to blinded data (§4.2) and 2) does not guide the developer towards avoiding hardware faults caused by security violations. The end result is that the developer must manually analyze and transform their code to prevent it from faulting. The goal of our programming model is therefore to provide strong security guarantees for memory safety and side-channel protection by guiding the developer using compiler warnings/errors, and, where possible, automatically applying code transformations.

### 5.1 Software Architecture

Figure 2 shows a high-level overview of the blinded capability software stack. Developers can indicate to the compiler that a variable is

[1]"Constant-time code" is a misnomer, as even side-channel-resistant code may still exhibit variable latency if the latency differences are not due to the (secret) data values. Common sources of such latencies include instruction-level latencies due to frequency scaling, or unpredictable contention of data buses for loads and stores.

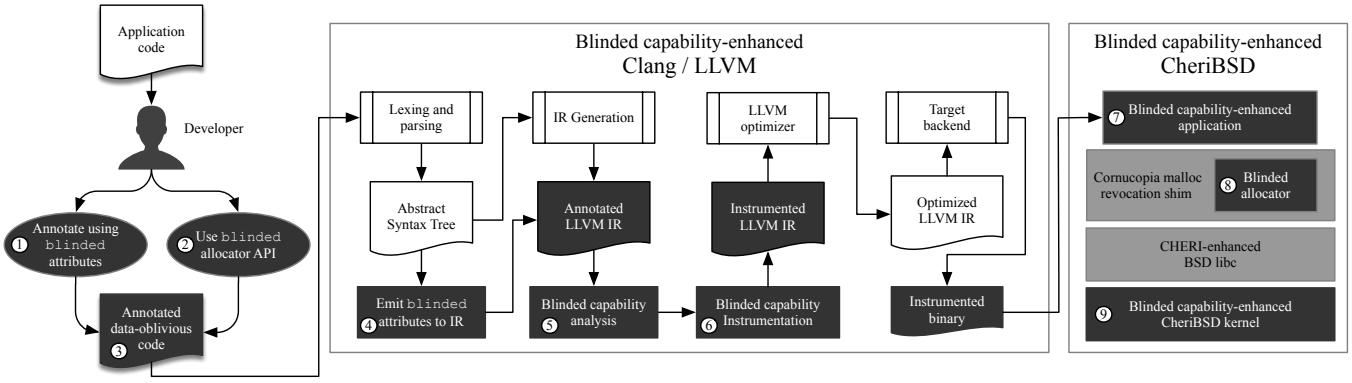



**Figure 2: High-level overview of the components of the blinded capability software stack. Dark gray denotes additions or modifications needed to support blinded capabilities, while light grey indicates components inherited from the CHERI architecture. White denotes components without significant changes.**

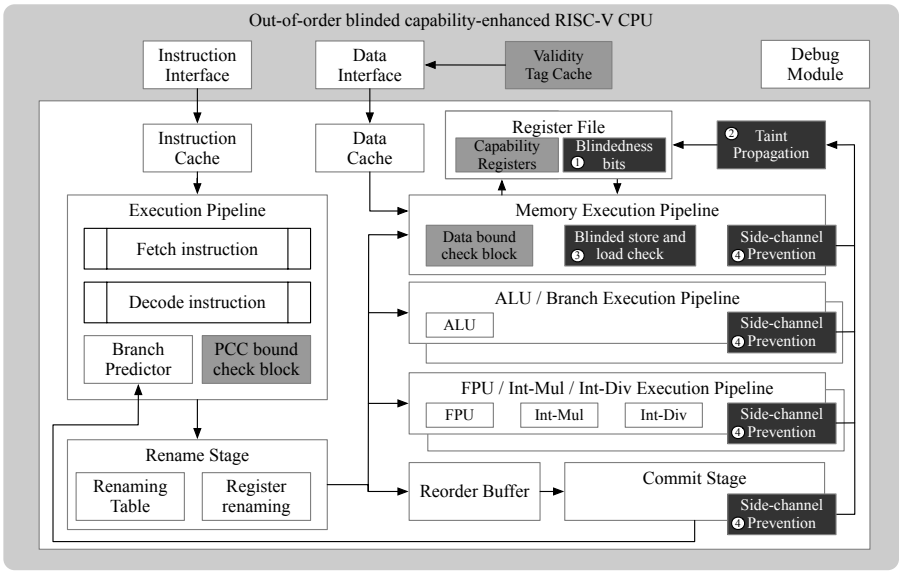



**Figure 3: High-level overview of the blinded capability-enhanced CHERI-RISC-V CPU hardware components. Dark grey denotes additions or modifications needed to support taint tracking, while light grey indicates additions common to CHERI-enabled CPUs. White denotes components without significant changes.**

sensitive by annotating its declaration with the `blinded` attribute ①. We refer to such variables as *blinded variables*. The compiler will then ensure that any accesses to blinded variables are permitted only through blinded capabilities, guaranteeing that the values of these variables will always be blinded when loaded into registers. We refer to registers holding blinded data as *blinded registers*. Blinded memory can be dynamically allocated using the `blinded` allocator application programming interface (API) ② which returns a blinded capability. Similarly, the compiler enforces that references to blinded memory cannot be assigned to non-blinded variables. Inside the compiler, we modify the Clang front-end to emit `blinded` attributes to the LLVM intermediate representation (IR) of the compiled program ④. These are used by the blinded capability analysis ⑤ and instrumentation passes ⑥ to produce a blinded capability-instrumented application binary. Global variables annotated with `blinded` attribute are reserved from a separate `blinded data` section by the static linker.

At run-time, the blinded-capability-enhanced application ⑦ requires platform support in the form of the blinded allocator ⑧ and an OS with a CHERI-capable kernel that has been altered to ensure its internal handling of capabilities does not trigger blinded capability violations. C startup code added by the compiler to ⑦ is responsible for initializing capabilities pointing to data in the blinded data section as blinded capabilities. System software not making use of blinded capabilities does not require modification (apart from their adaptation to CHERI). We discuss the compiler enhancements and operating system changes in detail in §6.2.

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

❶ __attribute__ ((blinded))
int data_oblivious_select(bool cond, int x, int y) {

❷ bool __blinded c; // c declared blinded and
                    // accessed via blinded capability

  int res;          // res not declared blinded
                    // but blindedness is inferred
  → ❸ Compiler infers res blinded from ❺

  c = cond;    // Uses store via blinded capability
               // (argument already in register)
  → ❹ Compiler knows c is already blinded based on declaration

  res = (x * c) + (y * (!c)); // HW propagates
                              // blindedness to res
  → ❺ Compiler infers res is blinded from this assignment

  return res;
}
```

**Listing 1: Data oblivious conditional select function which returns one of the arguments x or y based on the value of cond demonstrating the inference capabilities of the blinded capability-enhanced compiler.**

```
void bad_func(bool cond, int x, int *out) {

① int __blinded a = x;   // a declared blinded
  int b;

  if(cond)
      b = a;    // HW propagates blindedness to b
  → ② Compiler undecided on whether b becomes blinded

  *out = a;     // HW fault if out non-blinded (I1)
  → ③ Allowed by compiler as out might be blinded

  if(a != 0)    // Violates I4
      b = a;
  else
      return
  → ④ Rejected by compiler as blinded a used in control-flow decision

  if (b != 0)       // HW fault if b blinded (I4)
      *out = a;     // HW fault if out non-blinded (I1)
}→ ⑤ Control-flow allowed by compiler as b might be non-blinded
```

**Listing 2: Non-data-oblivious function which attempts to branch on a blinded condition and write blinded data to a possibly non-blinded output parameter. This code is rejected by the blinded capability-enhanced compiler.**

**Table 1: Blindedness bit propagation and side-channel prevention rules enforced by BLACKOUT hardware. x represents a "don't care" condition, which indicates the actual signal value or values have no impact on the decision outcome.**

| Instruction | Blinded capability | Blindedness | | Decision | Result blindedness |
|---|---|---|---|---|---|
| Arithmetic / Logic | - | Op1 | Op2 | | |
| | | $a$ | $b$ | Propagate | $a \vee b$ |
| | in Addr Reg | Addr Reg | Condition Ops | | |
| Branching | no | 0 | 0 | Allow | - |
| | no | 1 | x | Fault | - |
| | no | x | 1 | Fault | - |
| | yes | x | x | Fault | - |
| | in Addr Reg | Addr Reg | | | |
| Load | no | 0 | | Propagate | 0 |
| | yes | 0 | | Propagate | 1 |
| | x | 1 | | Fault | - |
| | in Addr Reg | Addr Reg | Data Reg | | |
| Store | no | 0 | 0 | Allow | - |
| | no | 0 | 1 | Fault | - |
| | yes | 0 | x | Allow | - |
| | x | 1 | x | Fault | - |

## 5.2 Hardware architecture

Figure 3 shows the high-level overview of the hardware changes needed to support blinded capabilities (shown in dark gray) applied to an out-of-order RISC-V core, such as MIT's RISCY-OO [104] or Bluespec's Toooba [71] processor intellectual property (IP). Apart from the changes introduced by CHERI (shown in light gray) the majority of the changes necessary in the CPU are to facilitate in-CPU taint tracking. In the register file, the general-purpose capability registers are extended to hold an additional *blindedness bit* ① which tracks whether the register is blinded. The taint-propagation hardware logic ② will in turn ensure that any data derived from these blinded registers, e.g., through arithmetic operations, will also "be blinded", i.e., it sets the blindedness bit of the destination register.

All blinded data is bound by a set of invariants (I1 to I5), stemming from the need to enforce exclusive access to blinded memory (§4.2), and side-channel prevention.

**Exclusive access invariants:**

- I1 Blinded data cannot be stored into memory using non-blinded capabilities.
- I2 No capability of any kind can be stored into blinded memory.
- I3 Bounds of valid blinded and non-blinded capabilities must not simultaneously overlap. However, once a region is released (i.e., is no longer accessible through a valid capability, e.g., due to heap deallocations or stack frame pops), it can be assigned to future blinded or non-blinded capabilities.

**Side-channel protection invariants:**

- I4 Control-flow instructions cannot use blinded operands as conditions or target addresses.
- I5 Load and store instructions cannot use blinded operands as addresses.

I1 is enforced by hardware through additional *blinded store and load checks* ③ in the memory execution pipeline which is responsible for the execution of any load and store instructions, whereas I4 and I5 are enforced by *side-channel prevention* logic integrated into any pipeline which is responsible for instructions with blinded operands ④. Any attempt by software to violate I1, I4 and I5 will result in a hardware fault. Table 1 shows the full blindedness bit propagation and side-channel prevention rules enforced by BLACKOUT.

I3 cannot be efficiently enforced by hardware alone. As mentioned in §4.2, blinded memory reclaimed when blinded capabilities are destroyed must be erased to avoid leaking blinded data; this is handled by the compiler and memory allocator for stack and heap variables, respectively (Figure 2, §6.2). The combination of hardware, compiler and software stack guarantees data confidentiality.

Data-oblivious computation ensures that control flow and memory accesses don't depend on secret data. Outputs, however, may be non-secret (e.g., decrypted data meant for users). In practice,

such output data cannot be left blinded indefinitely as some result of data-oblivious computation must eventually be extracted. In BLACKOUT, results can be marked non-secret by issuing non-blinded capabilities for result buffers. This reveals the final, non-secret result of blinded computation by relaxing invariant I3 in a controlled way to unblind the result. For example, since I3 is (in part) enforced by the blinded allocator, the allocator can expose a separate API for obtaining blinded results that, in contrast to, regular blinded memory returns two capabilities for a dedicated area for blinded results: one which is blinded, and used to write to the area, and the other non-blinded, but only usable for reading the result. However, the developer must use the blinded capability to the result area carefully to prevent accidental leakage of secret data. We discuss additional options to secure extraction further in §8.

### 5.3 Motivating Examples

The concrete examples in Listings 1 and 2 demonstrate how the hardware and the compiler work together to prevent blinded memory leakage. The hardware ensures that taint tracking is both accurate and precise (no under- or over-tainting occurs in the hardware), and that attempted leaks cause a fault. The compiler ensures that capabilities respect exclusive blinded memory access, and that reclaimed blinded stack memory is zeroed out.

The compile-time analysis is dependent on the developer providing *(*initial) information about which variables are expected to contain sensitive information through `blinded` attributes (`__blinded`, ❷ in Listing 1). Optionally, the developer can also declare functions blinded (`__attribute__((blinded))`, ❶ in Listing 1). This clearly identifies functions expected to return blinded data, improving developer ergonomics by enhancing the readability of data-oblivious code. The compile-time analysis is not limited to the information provided by the developer. For example, in Listing 1 ❸, the compiler can infer the variable `res` must be blinded due to the assignment from `c` tainting it at ❺.

Listing 2 shows a non-data-oblivious function which would fault at run-time due to violating I4 (and possibly I1). This example demonstrates that the compiler can reject some non-data-oblivious programs outright. However, as discussed in §2.3, compile-time data-obliviousness analysis cannot be sound as data-obliviousness is ultimately a property at machine-code level. For example, at ②, the compiler cannot know whether `cond==1`. Thus, it *cannot prove* that `b` is always blinded and must allow the control flow at ⑤. However, if `cond==1` at run time, the hardware will propagate the blindedness bit to `b` at ②; the subsequent branch at ⑤ will fault.

For ③, the compiler does not know whether `out` is a blinded capability and must also allow compilation. If `out` at run time is referenced by an unblinded capability, the hardware will detect the violating store and raise a fault.

The hardware would fault at ④ due to violation of I4. However, since it is detectable by the compiler, we force a compilation error to inform the developer early in the development cycle. This is especially useful for code paths that are rarely exercised with real workloads and require fuzzing techniques to uncover. More annotations allow the compiler to make more informed decisions and detect violating operations that it cannot infer on its own.

## 6 BLACKOUT Blinded Capability Implementation

We now describe BLACKOUT, our realization of blinded capabilities for CHERI-RISCV, CHERI-LLVM, and the CheriBSD OS.

### 6.1 BLACKOUT CHERI-RISC-V CPU

We implement blinded capabilities in register-transfer level (RTL) on CHERI-Toooba [36], a CHERI-enabled RISC-V processor based on Bluespec's speculative out-of-order Toooba IP [71]. Unlike previous data-oblivious ISAs [30, 102], BLACKOUT *does not add any additional instructions to the underlying ISA*; all architectural changes are achieved by adjusting the CHERI permission model.

**Non-oblivious access permission.** We introduce a new "non-oblivious access" permission bit in the previously unused section of the CHERI capability representation. The CHERI-Toooba IP allocates 12 bits for hardware permissions and 4 bits for user permissions, leaving 3 bits unused. By default, the blinded capability bit in newly created capabilities is set to 1. In this initial configuration, the enforcement of blinded capabilities is turned off, allowing the capability to be used freely within its defined bounds. When the existing CHERI `candperm` instruction (§2.1) is invoked with the blinded capability permission as an operand, it changes the non-oblivious access bit to 0, thereby enabling the enforcement of blinded capabilities. Defining the non-oblivious access bit this way means that a capability with this permission is *allowed* to handle data in a non-oblivious manner.

**Registers & taint-tracking.** CPU registers are extended with an additional blindedness bit to signify whether the data stored in the register is blinded. Any load instructions executed with a blinded capability set the blindedness bit of the target register to 1. Store instructions with a blinded capability only store the data in memory, but not the blindedness bit. Data confidentiality is maintained by the exclusive access invariants (I1 to I3, §5), which ensure that blinded data stored in memory is only accessible through blinded capabilities. Register-to-register instructions, such as arithmetic and logical operations, propagate the blindedness bit: if the output depends on a blinded input, the output becomes blinded. Overwriting a register containing blinded data with non-blinded data will result in that register's blindedness bit to be unset.

**Limiting impact on capability-modifying instructions.** We prohibit valid capabilities from being blinded (I2, §5). The execution of any capability-modifying instruction with operands that would result in a blinded capability causes a fault. Enforcing this invariant allows us to avoid unnecessarily making capability-modifying instructions data-oblivious. Note that this enforcement does not break the CHERI programming model; "*blinded blinded capabilities*" are themselves redundant as they can never be dereferenced to access memory in order to maintain confidentiality (I5, §5).

**Spilling registers containing blinded data.** Blinded data can reside in any general-purpose capability register, including those designated as caller- or callee-saved by the RISC-V application binary interface (ABI). The CHERI-RISC-V compiler can generate code that spills blinded registers using `csc` instructions, and restore them with `clc` instructions. Normally, spilling blinded registers onto the stack via the non-blinded `csp` violates invariant I1 (§5) as spills target non-blinded memory. However, since stack memory allocated for registers spills is inaccessible by other capabilities than

the csp, CHERI's memory-safety guarantees ensure that spilled register cannot be accessed improperly. Thus, we permit BLACKOUT's csc and clc instructions to spill and restore blinded registers csp.

However, another challenge arises: the non-blinded csp generates non-blinded data when storing spilled values. This causes ambiguity when restoring spilled registers, as the CPU cannot distinguish between blinded and non-blinded register spills. To address this, BLACKOUT's csc emits special *blinded register records (BRRs)* as a result of spilling blinded registers. A BRR is a 128-bit structure containing the spilled value and 64-bit marker. This marker differentiates BRRs from conventional CHERI capabilities and maintains their validity tags. When a spilled capability register is restored, the validity tag signals that the value restored from memory is either a capability or a BRR. The marker indicates whether the value is a BRR, ensuring the destination register is correctly blinded.

**Transient execution.** As mentioned in §2.2, hardware optimizations, such as speculation, can inadvertently introduce side channels when operating on secrets. To protect blinded data against such leakage, we ensure that any blinded data flowing to *decision-making* parts of the hardware (such as branch predictors) are zeroed out. This guarantees that execution is truly oblivious to blinded data, both architecturally and transiently. This is similar to approaches used in prior work [30, 102].

## 6.2 BLACKOUT Software Stack

We leverage Clang's existing annotate_type attribute designed for static analysis tools [13] for the __blinded attribute. Normally, annotate_type is not propagated to the LLVM IR. Thus, we extend Clang to emit __blinded attributes into the IR (④, Figure 2).

We introduce an additional analysis pass for blinded variables (⑤, Figure 2) in the LLVM-backend. This pass performs recursive dependency analysis to trace all uses of __blinded variables in data flows involving the store, load, and GetElementPtr (GEP) IR instructions. By recursively analyzing flows from blinded source instructions, e.g., loads from variables declared as __blinded, the pass automatically propagates the blindedness property to variables acting as sinks, blinding them at their respective point of allocation.

We additionally introduce an instrumentation pass (⑥, Figure 2) that transforms stack allocations (alloca IR instructions) for variables annotated with __blinded. The instrumentation adds the llvm.cheri.cap.perms.and intrinsic to each annotated alloca to unset the "non-blinded" permission control bit, thus turning the associated capabilitity into a blinded capability. Consequently, all stack memory associated with __blinded variables are only accessible through blinded capabilities.

**Blinded capability-enhanced allocator.** We extended the Cornucopia revocation mechanism to provide a blinded_malloc API that returns blinded capabilities to blinded heap allocations. To meet I3 (§5), such blinded heap allocations must not overlap with other, non-blinded allocations. This ensures the blinded capability returned blinded_malloc has exclusive access to the newly created allocation. The blinded_malloc API relies on Cornucopia to ensure the blinded capability does not overlap with any concurrently existing capabilities. BLACKOUT's blinded_malloc is integrated into CheriBSD via the MRS (similar to the integration of Cornucopia, discussed in §2.1). This makes blinded capabilities allocator-agnostic, allowing userspace processes to employ different underlying allocators as long as allocations occur via the shim.

**Securely reclaiming blinded memory.** Recall from §4.2 that blinded memory must be securely reclaimed to prevent information leaking from previously blinded memory regions. BLACKOUT implements explicit zeroing policies for blinded heap and stack allocations. For blinded heap allocation, the free API, implemented in the shim, inspects whether memory to be deallocated is blinded and erases the contents of blinded allocations before freeing it. For stack-allocated blinded variables, the compiler automatically inserts a memset IR instruction to zero out their memory immediately after the variable's lifetime ends. This ensures sensitive data from blinded variables is securely erased, preventing unintended reuse or leakage when stack frames are reallocated. Blinded global data persists until the process terminates and consequently does not need to be reclaimed. The contents of (physical) pages are zeroed out by the OS before they are recycled for other processes.

**Integration to CheriBSD.** As discussed in §6.1, any newly created capability must initially be configured with its "non-blinded" permission bit set. To support blinded capabilities in CheriBSD, we perform a thorough scan of all CheriBSD code to ensure that any derived permissions (e.g., for drivers or userspace) also unset the "non-blinded" permissions. We also extend the CheriBSD CPU exception and signal handlers to handle blinded capability exceptions and the new signal introduced to the BLACKOUT CHERI-Toooba Core. We integrated blinded capabilities into CheriBSD 24.05, resulting in a total of 100 lines of code changes over 14 distinct files.

# 7 Evaluation

We evaluate the overheads (area and performance) and security of BLACKOUT. To evaluate overheads, we extended the CHERI-RISC-V Toooba FPGA softcore (RV64ACDFIMSUxCHERI), which is based on the open-source Bluespec RISC-V 64-bit Toooba core.

We use the BESSPIN-GFE security evaluation platform [76], which has out-of-the-box support for the CHERI-Toooba softcore, replacing the standard core with our Blinded CHERI-Toooba. We synthesize the system-on-chip (SoC) at 25MHz (default for BESSPIN-GFE) targeting the Xilinx Virtex UltraScale+ VCU118 FPGA.

## 7.1 Power & resource usage

Table 2 shows the power and resource usage obtained from Xilinx Vivado 2019.1. The overheads are minimal ($\approx$ 1% area and $\approx$ 5% power) compared to the unmodified CHERI-Toooba. This is expected since our hardware additions require no additional storage in memory or caches and only a single additional bit for registers.

## 7.2 Performance

For performance measurements, we run several benchmarks on the VCU118 FPGA. First, we evaluate the impact of BLACKOUT CHERI-Toooba on unblinded workloads by running the EEMBC CoreMark benchmark [38] bare-metal on the GFE for $5 \times 10^3$ iterations over three separate test runs. Table 3 shows the mean result demonstrating that there is only a negligible effect on performance for unblinded workloads between the unmodified CHERI-Toooba core and our BLACKOUT CHERI-Toooba variant. We also obtain timing reports from Xilinx Vivado to show the effect of our hardware changes on the maximum clock frequency attainable. Both BLACKOUT and

**Table 2: Area and power costs on VCU118 @ 25MHz expressed in number of LUTs and registers, and Watts respectively.**

| | 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 |

**Table 3: Performance cost on VCU118 @ 25MHz expressed as CoreMark test results for $5x10^3$ iterations. The CoreMark score for a processor is reported as CoreMark-iterations-per-second-per-core-MHz. The Δ is relative to CHERI-Toooba Core results.**

| | Total ticks | Δ | | Total time (sec) | Δ | | Iterations/sec | Δ | | Score |
|---|---|---|---|---|---|---|---|---|---|---|
| **CHERI-Toooba** | | | | | | | | | | |
| baseline (nocap) | 927674227 | – | | 37 | – | | 135 | – | | 5.4 |
| purecap | 951983228 | 24309000 | 2.62% | 38 | 1 | 3% | 131 | 4 | 2.3% | 5.24 |
| **Blinded CHERI-Toooba** | | | | | | | | | | |
| baseline (nocap) | 927729879 | 55652 | 0.01% | 37 | 0 | 0% | 135 | 0 | 0% | 5.4 |
| purecap | 952083879 | 24409652 | 2.63% | 38 | 1 | 3% | 131 | 4 | 2.3% | 5.24 |

the baseline are synthesizable at 25MHz, and the worst negative slack (WNS) for BLACKOUT is 0.08ns compared to 0.06ns for the baseline, demonstrating no significant effect on clock frequency.

**OISA benchmarks.** Second, for blinded workloads, we evaluate the performance impact of BLACKOUT using five OISA benchmarks adapted from Yu et al. [102]. To adapt them to BLACKOUT, we simply blind secret inputs with the `__blinded` attribute, change dynamic allocations to use our blinded allocator API, and compile the benchmarks using our blinded capability-enhanced compiler. Porting the OISA benchmarks to BLACKOUT took 1-5 LoC changes (<1%) per benchmark. We measured the performance on CheriBSD with BLACKOUT support in pure-capability mode with the Cornucopia revocation mechanism enabled. Our setup follows the CheriBSD benchmark guide [26], with the exception of running the BLACKOUT benchmarks in pure-capability mode (§2.1) with blinded capabilities, and enabling the experimental Cornucopia revocation mechanism, which is essential to guarantee exclusive access for blinded capabilities. We compiled all benchmarks with `-O3` and measured the combined user and system time using the *time* command. This measures the entire lifetime of blinded capabilities—including capability creation, algorithm execution, and memory reclamation (i.e., zeroing out blinded memory). For the `findmax`, `binary_search`, and `integer_sort` benchmarks, we used blinded input arrays containing $2^N$ integers. For the matrix multiplication benchmark, we multiplied two square matrices of size $2^{\frac{N}{2}} \times 2^{\frac{N}{2}}$. In the DNN example, the blinded input size was $2^{\frac{N}{2}}$, and the network consisted of two layers, each of size $2^{\frac{N}{2}} \times 2^{\frac{N}{2}}$, with a fixed output size of $2^6$. We evaluated all benchmarks with $N \in \{12, 14, 16, 18, 20\}$.

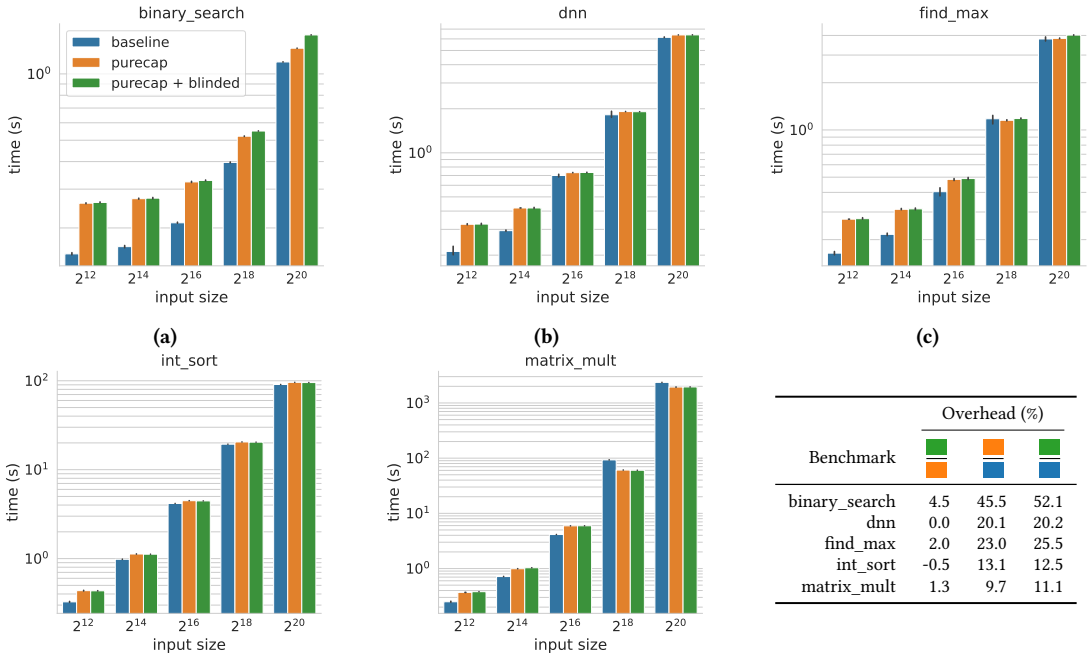
The results in Figure 4 show the OISA benchmark run-time in nanoseconds for different input sizes and three configurations: *baseline*, with capability-enforcement disabled, *purecap*, which enforces CHERI memory-safety only, and *purecap + blinded*, which enforces all BLACKOUT invariants including CHERI memory-safety and data-obliviousness. As our hardware modifications do not add additional cycles to any instructions, all configurations can run with the same FPGA bitstream, requiring only different compilation flags. The baseline configuration uses CHERI's "hybrid" mode which allows CHERI-capable processors to run legacy, non-capability code.

The geometric mean overhead per benchmark is shown in Figure 4f. The overall result shows a minimal geometric mean overhead of 1.5% for BLACKOUT compared to the *purecap* configuration, which is several times lower than overheads for prior work enforcing data-oblivious computation [30, 102]. The overhead in blinded workloads is caused by several factors:

(1) Clearing blinded memory at the end of a blinded capability's lifecycle. This is done by the compiler for blinded memory on the stack on function returns, and by the `blinded_malloc` for blinded memory on the heap on freeing. For instance, when N = 20, binary search operates on a 4MiB array. In this setting, memory reclamation (zeroing) accounts for most of the overhead.
(2) Additional stores and loads to set the blindedness bit in registers. When initializing blinded data at the start of the benchmarks, any data in registers must be first stored into blinded memory and then loaded back to set the register blindedness bit. This only occurs for the initial blinded data and is not required to propagate blindedness during execution. The effect on performance is therefore minimal. Nevertheless, we discuss a potential optimization to avoid this in §8.
(3) Changes to instruction cache performance caused by the slight increase in code size due to compiler instrumentation.

We also evaluate the *combined* performance impact of CHERI's memory-safety and BLACKOUT's data-oblivious enforcement. We measure a moderate geometric mean overhead of 23.5% compared to the unprotected baseline configuration with capability-enforcement completely disabled. We observe that the overhead for both CHERI's purecap and BLACKOUT's purecap + blinded modes relative to the unprotected baseline is significantly larger in the `binary_search` benchmark, particularly for smaller input sizes, compared to the other OISA benchmarks. We investigate further the composition of the overhead in that benchmark. In consultation with the University of Cambridge Computer Laboratory Security Group we identified three possible reasons for the poor performance:

(1) Inefficiencies in CHERI-RISC-V's relocation representation and processing converting function pointers into capabilities.
(2) Lack of support for lazy binding of functions in CHERI-RISC-V which manifests as high initial load times at startup.




| Benchmark | Overhead (%) | | |
|---|---|---|---|
| | | | |
| 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 |

**Figure 4: Run-time in seconds of the OISA `binary_search` (a), `dnn` (b),`find_max` (c), `int_sort` (d), and `matrix_mult` (e) benchmarks built for the *baseline* with capability-enforcement disabled, *purecap* mode with CHERI memory-safety enforced but data-obliviousness not enforced, and *purecap+blinded* mode with both CHERI memory-safety and data-obliviousness enforced. Figure 4f shows the geometric mean overheads for each benchmark aggregated over all input sizes.**

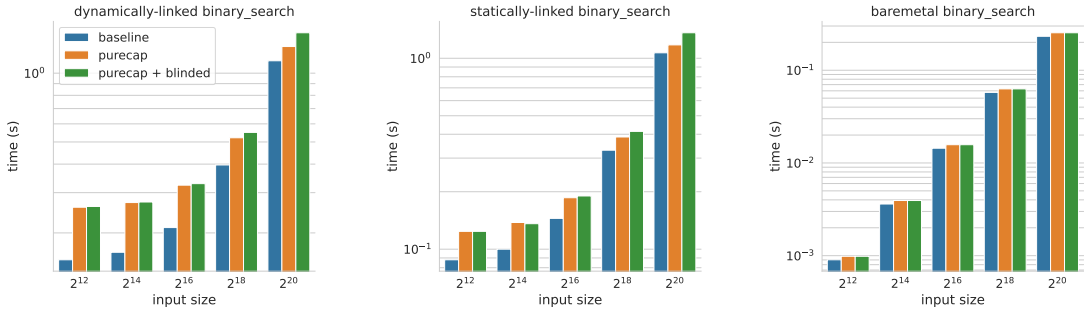



**Figure 5: Run-time in seconds of the OISA `binary_search` benchmark in dynamically-linked, statically-linked, and bare-metal configurations for the *baseline*, *purecap*, and *purecap+blinded* modes. The geometric mean overhead of *purecap+blinded* compared to *baseline* aggregated over all input sizes falls from 52.1% for the dynamically-linked configuration (a) to 32.0% for the statically-linked one (b). The corresponding geometric mean is 9.1% for the baremetal configuration (c).**

(3) The current version of CHERI-LLVM currently disables most loop optimizations, making it slower for the type of code the `binary_search` benchmark relies on compared to compilation to a non-CHERI RISC-V target.

To isolate the effect of these potential root causes for the overhead that are independent of the `BLACKOUT`-related changes, we repeat the `binary_search` benchmark in two additional configurations: a) a statically linked benchmark on CheriBSD designed to mitigate the inefficiencies in relocation (1) and lazy binding (2), and b) a bare-metal version of the benchmark that runs without CheriBSD. Crucially, the bare-metal benchmark measurements include only the execution of the data-oblivious algorithm, eliminating the initial startup cost, reclaiming of memory, and system calls, thus isolating the computational cost of blinded capabilities without their impact on memory management.

Figure 5 shows the results of these additional `binary_search` configurations and allows us to make the following observations: First, in the statically-linked benchmarks, the overhead for smaller input sizes improve, but the result for larger input sizes is similar to the dynamically-linked benchmarks, suggesting that the one-off cost of initial load times is amortized over the longer benchmark runs in both cases. Second, the overheads for both purecap and

purecap + blinded modes compared to the baseline are significantly lower when run on baremetal. This supports our hypothesis that relocation (1) and lack of lazy binding (2) are the main sources of overhead for `binary_search` in Figure 4.

We also note some cases in the `matrix_mult` benchmark in Figure 4e where the baseline performs slightly worse than the other two. We examined the generated assembly for these benchmarks and discovered that the CHERI-enabled compiler does a better job at optimizing parts of the code whose effect on performance grows with input size, leading to slightly lower run-times in the purecap and purecap + blinded configurations.

**SpectreGuard benchmarks.** Third, we run the SpectreGuard [37] synthetic benchmark using the version adapted by Daniel et al. [24] to evaluate ProSpeCT (see §9). This version uses data-oblivious implementations of the `chacha20`, `sha2`, and `curve25519` cryptographic algorithms from HACL*[107], a formally verified cryptographic library written in F* [88]. The benchmarks represent a workload consisting of public-data computations, which gain substantial performance from speculative execution, alongside encryption routines, which are less impacted by speculation. Each benchmark varies the proportion of speculative versus cryptographic work (S/C), as shown in Table 4. The ProSpect benchmark version already annotates all secret values in those test cases. We adapt those annotations to `blinded` attributes and run all the test cases on `BLACKOUT` CHERI-Toooba in both *purecap* and *purecap + blinded* versions. Table 4 shows the results (average of 20 runs) for the `chacha20` and `sha2` test cases. Unlike for the OISA benchmark results, where `BLACKOUT`'s overhead could be attributed to zeroing out stack and heap variables, the ProSpeCT version of the SpectreGuard benchmark places all secret variables in static variables allocated from the program's data section, similar to global variables. Consequently, `BLACKOUT` does not introduce any discernible overhead for encryption time or workload time. The slight speed improvement of the purecap + blinded version compared to the purecap baseline in chacha20 25S/7C and sha2 90S/10C falls within standard deviation (maximum $\sigma = 0.08\%$). For completeness, we also include results for nocap, which performs worse than purecap in most cases. We attribute this difference to improvements in the RISC-V backend for CHERI-RISC-V targets compared to plain RISC-V targets in CHERI-LLVM, resulting in more compact and efficient assembly. We confirmed this through manual investigation.

For the Curve25519 test case, we discovered a constant-time violation caused by a secret-dependent branch instruction in the compiled binary. Although this violation is not present in the formally verified source code, it is introduced by the compiler, which in our case is CHERI's fork of Clang. This violation is not reported in ProSpeCT's evaluation because it uses the GCC compiler, which likely correctly avoided generating such a secret-dependent branch. This difference in generated assembly code highlights the usefulness of `BLACKOUT`: different compilers and/or configurations can silently introduce constant-time violations which `BLACKOUT` can detect and stop even if the source does not contain such a violation.

## 7.3 Security

**Empirical Spectre mitigation evaluation.** CHERI-Toooba is vulnerable to Spectre branch target buffer (BTB) [56], return stack buffer (RSB) [57, 64] and store to load (STL) [46] (cf. test suite in [35]). We blinded the secret value in all test cases by inserting a single `candperm` instruction. As seen in Table 5, `BLACKOUT` prevents all Spectre attacks from [35] because `BLACKOUT` catches side-channel violations even during speculation, but suppresses faults until speculation is confirmed (and ignores them otherwise). Additionally, we have replicated ProSpeCT's [24] Spectre tests. However, as they involve dereferencing a secret data value as a pointer, they are inherently prevented by CHERI's architectural security properties, which preclude dereferencing non-capability data. Note that none of the Spectre tests in [24, 35] use transient capability forgery, and therefore do not require CSC to prevent.

**Empirical non-interference evaluation.** Data-oblivious software inherently provides the non-interference property [97]. We verify this empirically using the methodology from [11, 97, 98]: relevant signal traces are extracted from the value change dump (VCD) waveforms across different program runs to ensure that runs with different secret data values produce identical signal traces. As in prior work [97], we extract traces for signals that could leak secret data through cache-timing (addresses of cache accesses), speculative execution (branch predictor states), and port contention and instruction latencies (reorder buffer scheduling). As `BLACKOUT` enforces data-oblivious processing of blinded data, we use the data-oblivious `binary_search` program from [102] as a case study. We vary the blinded data values between two runs of the program, and verify that the extracted traces for the relevant signals are identical.

**Theoretical security evaluation.** Our security argument is composed of three invariants:

(1) **Standard CHERI-provided memory safety**. This includes attacks based on memory safety violations, such as out-of-bounds access and use-after-free. We do not loosen any of the restrictions imposed by standard CHERI (such as proper capability bounds and lifetime). We are therefore compliant with the standard CHERI model, and as such inherit its safety guarantees against spatial and temporal memory violations. This extends to the guarantees provided by formal models [28, 39, 75].
(2) **Side-channel protection for blinded data**. This is provided by the enforcement of data-oblivious computation on blinded data (I4 and I5). Any violation of this results in a fault, as explained in §5. We inherit formal guarantees for this from the OISA and BliMe models [30, 102]. We further show its efficacy empirically by running the OISA benchmarks [102] using `BLACKOUT` and verify that introducing non-data-oblivious changes to them is either detected by our compiler or causes a run-time fault.
(3) **Confidentiality of blinded data with respect to direct access**. While BliMe guarantees the confidentiality of blinded data in memory using in-memory tags, we achieve the same goal by ensuring that 1) blinded capabilities have exclusive access to blinded data throughout their lifetime (I3), 2) memory containing blinded data is cleared at the end of the corresponding blinded capability's lifetime (also I3), and 3) blinded data cannot be stored to memory using unblinded capabilities (I1). The resulting combination ensures that, as in OISA and BliMe, any data stored as blinded into memory, is loaded as blinded into registers. This holds until the data is explicitly declassified (§5).

# 8 Discussion & Limitations

**Blinding existing variables and oblivious-data-race-safety.** As discussed in §5, `BLACKOUT` ensures that memory designated as

**Table 4: SpectreGuard benchmark performance average measured over 20 runs of each test case (maximum $\sigma = 0.08\%$).**

| Blinded CHERI-Toooba | 25S/75C | | 50S/50C | | 75S/25C | | 90S/10C | |
|---|---|---|---|---|---|---|---|---|
| | Total Ticks | Δ (to row above) | Total Ticks | Δ (to row above) | Total Ticks | Δ (to row above) | Total Ticks | Δ (to row above) |
| **chacha20** | | | | | | | | |
| nocap | 25860874 | – | 23132682 | – | 25417083 | – | 20661078 | – |
| purecap | 22546871 | -3314003 (−12.81%) | 20378637 | -2754045 (−11.91%) | 25430015 | 12932 (0.05%) | 20256607 | -404472 (−1.96%) |
| purecap + blinded | 22541024 | -5847 (−0.03%) | 20378935 | 298 (0.00%) | 25434929 | 4914 (0.02%) | 20256061 | -546 (0.00%) |
| **sha2** | | | | | | | | |
| nocap | 24819790 | – | 21595015 | – | 25417599 | – | 22163550 | – |
| purecap | 24771185 | -48604 (−0.20%) | 21591164 | -3851 (−0.02%) | 26342237 | 924638 (3.64%) | 22378912 | 215362 (0.97%) |
| purecap + blinded | 24771730 | 544 (0.00%) | 21593499 | 2335 (0.01%) | 26342910 | 672 (0.00%) | 22371464 | -7448 (−0.03%) |

**Table 5: Transient-execution attacks successfully prevented on CHERI-Toooba and Blinded CHERI-Toooba. ✓ indicates the attack is successfully defended against, while ✗ indicates the attack succeeds.**

| | Spectre variant | | | |
|---|---|---|---|---|
| | PHT | BTB | RSB | STL |
| **CHERI-Toooba** | ✓ | ✗ | ✗ | ✗ |
| **Blinded CHERI-Toooba** | ✓ | ✓ | ✓ | ✓ |

blinded—either annotated by the developer, or inferred as such by the compiler—is blinded at allocation time. This ensures exclusive access through corresponding blinded capabilities, satisfying invariants I1 to I3 (§5). However, some correct data-oblivious programs may initially allocate memory as non-blinded, only requiring it to be blinded after interacting with other blinded data. While BLACKOUT restricts this pattern, a less-restrictive variation could allow dynamically blinding previously allocated memory. This would require scanning program memory to revoke existing, non-blinded capabilities referencing to-be-blinded memory, similar to revocation strategies used by Cornucopia [94] and Cornucopia Reloaded [32]. The trade-off is increased performance overhead. Future work might explore dynamic or delayed revocation strategies to reduce this cost while ensuring eventual exclusive access.

Eventually, data-oblivious software must *unblind* memory after data-oblivious computation concludes and secret intermediates are cleared (§5). In concurrent scenarios, shared blinded memory may require revoking overlapping blinded capabilities to maintain temporal memory safety and prevent unauthorized reuse after unblinding. Currently, BLACKOUT does not address this *oblivious-data-race-safety*, leaving it as an open problem for future work.

**Unblinding results through capability escrow.** A solution for further securing the unblinding of blinded memory containing non-sensitive results is to leverage CHERI's facilities for *capability sealing* (§2.1). Sealed capabilities cannot be de-referenced and protect against tampering by fixing properties like permissions and bounds; any attempt to tamper with the capability will result in invalidating it. In this context, sealed capabilities enable a form of "*capability escrow*", where a non-blinded version of a blinded capability stored for safe-keeping until the data-oblivious operations complete. Software can use a blinded capability to seal its non-blinded counterpart, which cannot be de-referenced while sealed and thus poses no threat to confidentiality. After data-oblivious computation has completed, secret intermediate values are cleared, and the blinded capability is used to unseal the non-blinded counterpart, allowing access to the result. Designing software APIs for capability escrow is left for future work.

**Different techniques for developer annotations** In BLACKOUT, developers use `__blinded` annotations to indicate blinded variables (§6.2). Compiler attributes are a common mechanism for conveying semantic information without changing the underlying language. BLACKOUT leverages Clang's existing `annotate_type` mechanism, available since LLVM 14, avoiding intrusive compiler changes.

However, this approach has limitations—annotating positional parameters in functions can be cumbersome within the confines of existing types of annotations. An alternative would be to introduce a "blinded C" dialect that integrates blinding directly into the type system, similar to the approaches discussed for Rust in § 1. For example, a `blinded` keyword akin to `const` could be used to mark variables as a blinded counterpart of their basic type. While this would improve developer ergonomics and simplify function parameter annotations, it would require intrusive changes to the language and compiler. This could hinder maintainability and extensibility, especially for evolving platforms like CHERI.

**Limitations of BRR structures for blinded data storage.** While blinded register record (BRR) structures effectively manage blinded register spills by clearly identifying and restoring blinded values, it is impractical to store all blinded data exclusively using BRR-like structures. The principal reason is the substantial memory overhead—BRR structures effectively double memory consumption. Such overhead would significantly degrade system performance and scalability, particularly for applications requiring large quantities of blinded data. Therefore, BRR structures should remain reserved for specific contexts, such as register spilling, while more efficient blinded capabilities handle general blinded memory storage.

**ISA extensions.** As we note in § 6.1, BLACKOUT, unlike previous data-oblivious ISAs [30, 102], does not add any additional instructions to the underlying ISA but leverages existing CHERI functionality with only small adjustments to its permission model to facilitate necessary architectural changes to support blinded capabilities. Future work can extend the ISA by adding dedicated instructions to support blinded memory. A potential extension is an instruction to directly set the blindedness bit in a register without a load via a blinded capability. This would allow the blinded capability-enhanced compiler to optimize code where a certain variable can be kept completely in a register throughout its lifetime. Currently, BLACKOUT requires such variables to be allocated on the stack in order to load them via the corresponding blinded capability.

**Relevance for CHERI standarization.** BLACKOUT demonstrates a practical method for enhancing CHERI's security model without invasive architectural changes by integrating data-oblivious computation capabilities into the existing CHERI protection model. This integration provides insights into how capability-based architectures can evolve to support advanced security properties, such as data-oblivious computation, while maintaining compatibility and performance. Consequently, BLACKOUT's principles and mechanisms can inform ongoing CHERI standardization efforts [17], potentially guiding the evolution of CHERI-enabled architectures towards broader security guarantees. Above, we discussed adding custom instructions. But maintaining the CHERI ISA as we currently do in BLACKOUT provides backward-compatibility for non-BLACKOUT hardware and can ease integration into the standard.

**Relationship to non-interference.** Since BLACKOUT builds on BliMe, it also ensures (as shown in Section VI of [30]) the values of blinded data have no effect on the rest of the system by 1) preventing blinded values from directly flowing to insecure instructions, and 2) by ensuring no secret-dependent branches are allowed when operating on blinded data, with an exception for controlled declassification as described above and in §5.2. Given its scope, we conjecture that this guarantee satisfies the non-interference property even though it was not explicitly discussed in [30]. Providing a formal model for BLACKOUT itself and explicitly establishing non-interference are left for future work. Nevertheless, we empirically investigate non-interference for BLACKOUT in §7.3.

## 9 Related Work

**Developing side-channel resistant software.** Software that handles sensitive data—such as cryptographic libraries or operating-system kernels—must carefully avoid introducing side-channel leakage. Most code is not naturally resistant to such leaks as writing constant-time code is often counterintuitive for programmers used to optimizing for performance or resource efficiency [47]. Development toolchains are not well suited for constant-time programming for two main reasons: (1) optimizing compilers can break data-obliviousness properties of high-level code [85], and (2) constant-time libraries [1, 7, 55, 67] must balance security with the cost of disabling optimizations, e.g., by implementing constant-time functionality in architecture-specific inline assembly. A complete solution would likely require invasive changes to the compiler infrastructure, such as extending LLVM's type system to annotate values needing timing protection and limiting optimizations on them—at significant performance cost [67]. Some constant-time Rust libraries avoid compiler optimization interference by implementing all sensitive operations in inline assembly [55, 66], which is opaque to LLVM's optimization passes. However, this also prevents the Rust compiler from verifying the code's memory-safety correctness and inhibits even simple optimizations such as constant folding and algebraic simplification.

**Verifying constant-time code.** Researchers have explored various methods for writing and verifying data-oblivious and constant-time code [2, 6, 9, 10, 12, 21, 22, 29, 31, 33, 68, 69, 72, 80, 84, 87, 106], resulting in numerous tools offering informal [27, 59, 81, 93, 95, 96] and formal [16, 23] guarantees. An actively maintained list of "constant-timeness" verification tools (CT-tools) currently includes 55 such tools [65]. However, both static analysis or formal methods face signficant challenges. The principal shortcoming of static analysis approaches is that data-obliviousness can only be defined at machine-code level, rather than for high-level language constructs. Capturing microarchitectural subtleties of real-world hardware in formal models (and keeping the models up-to-date as hardware evolves) is difficult. Static analyses may not be sound and can lead to over- or under-tainting. Lastly, testing whether a program is data-oblivious remains challenging as tools built on static and formal analysis are typically not integrated into modern toolchains, have significant technical limitations including high overheads to compilation time, many false-positives, and are difficult to use [34, 40, 50].

In practice, constant-time coding practices alone are unreliable [79]. Compiler optimizations continue to evolve, and new compilers emerge in unexpected contexts (e.g., in-silicon just-in-time compilers in CPU hardware). Documentation gaps across the software and hardware stack further hinder efforts, especially when target platforms are not narrowly confined [78]. Thus, robust enforcement of data-obliviousness requires hardware/software codesign [29, 30, 61, 102]. Such designs allow programmers to annotate sensitive data, enabling hardware to enforce protection against side-channel leakage. Data-oblivious ISAs [30, 102], discussed in §2.3, use hardware-based taint tracking to transform *silent* timing side-channel leaks into *explicit* hardware faults. Yet, programming for data-oblivious ISAs remains hard due to the lack of integrated toolchain support for expressing such software/hardware contracts.

Architectural-Mimicry (AMi) [97] introduces new ISA primitives for more efficient control-flow-linearization, addressing how to make programs data-oblivious. AMi is complementary to data-oblivious ISAs, which enforce data-obliviousness through ISA contracts. However, it requires greater developer involvement as it requires developer write assembly manually (with correct usage of new primitives); incorrect usage can *silently* leak secrets.

BLACKOUT complements static-analysis and formal verification approaches by bridging the gap between high-level verified data-oblivious software and the low-level machine code. BLACKOUT *enforces* data-oblivious implementation through a hardware/software co-design that integrates with mainstream toolchains. Unlike static analysis tools, it provides hardware-enforced taint tracking, reliably converting implicit timing leaks into explicit faults. By extending the CHERI capability model with blinded capabilities, BLACKOUT ensures both memory-safety and data-obliviousness without invasive architectural changes. It simplifies secure programming by modifying the CHERI Clang/LLVM compiler to infer when data-oblivious invariants should be enforced. Thus, BLACKOUT simultaneously addresses usability, security, and performance challenges that previous approaches tackled in isolation.

**Securing speculation.** Many hardware defenses adapt CPU microarchitectures to defend against Spectre and other speculative execution side-channel attacks by isolating the microarchitectural elements, such as cache hierarchies [3, 51, 52, 62, 82, 83, 89, 100, 105], that can either influence speculation or leak data across security domains. Speculative taint-tracking techniques[24, 37, 86, 103] delay instructions dependent on speculatively loaded secret data. Of recent work, ProSpeCT [24] formalizes the constant-time policy with respect to control flow and memory accesses for a broad class of speculation mechanisms. Its scope is guaranteeing that hardware does not leak secrets of constant-time programs during speculation.

Unlike data-oblivious ISAs, neither ProSpeCT nor other transient execution defenses enforce their guarantees on non-speculative execution. As such, they do not help developers write constant-time code. BLACKOUT prevents secret leakage even when programs are not constant-time and helps developers refine annotations. Future work can integrate formal approaches with BLACKOUT so that once developers transform code, it can be guaranteed data-oblivious.

## 10 Conclusion

We introduced BLACKOUT, a novel approach that integrates data-oblivious computation into the CHERI capability-based security model. Future research will focus on optimizing performance of reclaiming blinded memory, exploring dynamic revocation techniques, and bringing these into CHERI standardization efforts.

## Acknowledgments

We thank our colleagues at University of Waterloo and Ericsson: Adam Caulfield, Christoph Baumann, Håkan Englund, Santeri Paavolainen, and Sini Ruohomaa for their comments on drafts of this paper. We also thank the University of Cambridge Computer Laboratory members, particularly Jessica Clarke, Jonathan Woodruff, Peter Rugg, Robert Watson, and former member Alex Richardson.

This work is supported in part by Natural Sciences and Engineering Research Council of Canada (grant number RGPIN-2020-04744), and the Government of Ontario (RE011-038). Views expressed in the paper are those of the authors and do not necessarily reflect the position of the funders.

## References

[1] isis agora lovecruft, Henry de Valence, and Tony Arcieri. 2025. Dalek-Cryptography/Subtle. https://github.com/dalek-cryptography/subtle

[2] Adil Ahmad, and others. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *NDSS '18*. Internet Society. doi:10.14722/ndss.2018.23284

[3] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-like Attacks by Capturing Speculative State. In *ISCA '20*. IEEE, 132–144. doi:10.1109/ISCA45697.2020.00022

[4] Alejandro Cabrera Aldaya, and others. 2019. Port Contention for Fun and Profit. In *S&P '19*. IEEE, 870–887. doi:10.1109/SP.2019.00066

[5] Saar Amar, and others. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *MICRO '23*. ACM, 641–653. doi:10.1145/3613424.3614266

[6] Marc Andrysco, and others. 2015. On Subnormal Floating Point and Abnormal Timing. In *S&P '15*. IEEE, 623–639. doi:10.1109/SP.2015.44

[7] Go Authors. 2025. Crypto/Subtle. https://pkg.go.dev/crypto/subtle

[8] Ariel Ben-Yehuda. 2018. Keeping Secrets in Rust · Issue #2533 · Rust-Lang/Rfcs. GitHub. https://github.com/rust-lang/rfcs/issues/2533 (accessed 2025-03-21).

[9] Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *FSE '05*. Springer-Verlag, 32–49. doi:10.1007/11502760_3

[10] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC '06*. Springer-Verlag, 207–228. doi:10.1007/11745853_14

[11] Marton Bognar, and others. 2023. MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling. In *EuroS&P '23*. 651–670. doi:10.1109/EuroSP57164.2023.00045

[12] Pietro Borrello, and others. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS '21*. ACM, 715–733. doi:10.1145/3460120.3484583

[13] Martin Brænne and Dmitri Gribenko. 2022. [RFC] New Attribute 'annotate_type' (Iteration 2). LLVM Discussion Forums. https://discourse.llvm.org/t/61378 (accessed 2025-03-30).

[14] Claudio Canella, and others. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security '19*. USENIX, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[15] Claudio Canella, and others. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS '19*. ACM, 769–784. doi:10.1145/3319535.3363219

[16] Sunjay Cauligi, and others. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *SecDev '17*. IEEE, 69–76. doi:10.1109/SecDev.2017.24

[17] CheriAlliance. 2025. CHERI Alliance – Industry-led Security Technology. https://cheri-alliance.org/ (accessed 2025-04-06).

[18] Li-Chung Chiang and Shih-Wei Li. 2025. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In *ASPLOS '25*. ACM, 1014–1027. doi:10.1145/3676641.3716017

[19] David Chisnall, and others. 2017. CHERI JNI: Sinking the Java Security Model into the C. In *ASPLOS '17*. ACM, 569–583. doi:10.1145/3037697.3037725

[20] CISA. 2023. *Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software*. Whitepaper. Cybersecurity and Infrastructure Security Agency. 36 pages. https://www.cisa.gov/sites/default/files/2023-10/SecureByDesign_1025_508c.pdf

[21] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. 2012. Compiler Mitigations for Time Attacks on Modern X86 Processors. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012), 1–20. doi:10.1145/2086696.2086702

[22] Bart Coppens, and others. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors. In *S&P '09*. IEEE, 45–60. doi:10.1109/SP.2009.19

[23] Lesly-Ann Daniel, Sebastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *S&P '20*. IEEE, 1021–1038. doi:10.1109/SP40000.2020.00074

[24] Lesly-Ann Daniel, and others. 2023. ProSPeCT: Provably Secure Speculation for the Constant-Time Policy. In *USENIX Security '23*. USENIX, 7161–7178. https://www.usenix.org/conference/usenixsecurity23/presentation/daniel

[25] Brooks Davis, and others. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *ASPLOS '19*. ACM, 379–393. doi:10.1145/3297858.3304042

[26] Digital Security by Design. 2024. Benchmarking Guidance - Getting Started with CheriBSD 23.11. https://www.cheribsd.org/getting-started/23.11/benchmarking/ (accessed 2025-04-13).

[27] Craig Disselkoen. 2024. Haybale-Pitchfork. UCSD PLSysSec. https://github.com/PLSysSec/haybale-pitchfork

[28] Anna Lena Duque Antón, and others. 2025. VeriCHERI: Exhaustive Formal Security Verification of CHERI at the RTL. In *ICCAD '24*. ACM, 1–9. https://doi.org/10.1145/3676536.3676841

[29] Hossam ElAtali, and others. 2024. BliMe Linter. In *SecDev '20*. IEEE, 46–53. doi:10.1109/SecDev61143.2024.00011

[30] Hossam ElAtali, and others. 2024. BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking. In *NDSS '24*. Internet Society. doi:10.14722/ndss.2024.24105

[31] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 169–183. doi:10.14778/3364324.3364331

[32] Nathaniel Wesley Filardo, and others. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *ASPLOS '24*. ACM, 251–268. doi:10.1145/3620665.3640416

[33] Ben Fisch, and others. 2017. IRON: Functional Encryption Using Intel SGX. In *CCS '17*. ACM, 765–782. doi:10.1145/3133956.3134106

[34] Marcel Fourné, and others. 2024. "These Results Must Be False": A Usability Evaluation of Constant-Time Analysis Tools. In *USENIX Security '24*. USENIX. https://www.usenix.org/conference/usenixsecurity24/presentation/fourne

[35] Franz A Fuchs, and others. 2021. Developing a Test Suite for Transient-Execution Attacks on RISC-V and CHERI-RISC-V. In *CARRV '21*. 7. https://carrv.github.io/2021/papers/CARRV2021_paper_95_Fuchs.pdf

[36] Franz A. Fuchs, and others. 2024. Safe Speculation for CHERI. In *ICCD '24*. IEEE, 364–372. doi:10.1109/ICCD63220.2024.00063

[37] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *DAC '19*. ACM, 1–6. doi:10.1145/3316781.3317914

[38] Shay Gal-On and Markus Levy. 2012. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* 6, 23 (2012), 87.

[39] Dapeng Gao and Tom Melham. 2021. End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers. In *FMCAD '21*. TU Wien. doi:10.34727/2021/ISBN.978-3-85448-046-4

[40] Antoine Geimer, and others. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *CCS '23*. ACM, 1690–1704. doi:10.1145/3576915.3623112

[41] Aïna Linn Georges, and others. 2021. Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities. In *POPL '21*. doi:10.1145/3434287

[42] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. 2025. Do Compilers Break Constant-time Guarantees?. In *FC '25*. Springer. https://fc25.ifca.ai/preproceedings/13.pdf

[43] Richard Grisenthwaite. 2022. Arm Morello Evaluation Platform -Validating CHERI-based Security in a High-performance System. In *HCS '22*. IEEE, 1–22. doi:10.1109/HCS55958.2022.9895591

[44] Daniel Gruss, and others. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA '16*. Springer-Verlag, 279–299. doi:10.1007/978-3-319-40667-1_14

[45] Merve Gülmez, and others. 2025. Mon CHÉRI: Mitigating Uninitialized Memory Access with Conditional Capabilities. In *S&P '15*. IEEE, 829–847. doi:10.1109/

SP61157.2025.00133
[46] Jann Horn. 2018. Speculative Execution, Variant 4: Speculative Store Bypass. Project Zero Issue Tracker. https://project-zero.issues.chromium.org/issues/42450580 (accessed 2025-07-26).
[47] Intel. 2019. Security Best Practices for Side Channel Resistance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/security-best-practices-side-channel-resistance.html (accessed 2025-04-01).
[48] Intel. 2023. Data Operand Independent Timing Instructions. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html (accessed 2025-04-04).
[49] Intel. 2023. Data Operand Independent Timing ISA Guidance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html (accessed 2025-04-04).
[50] Jan Jancar, and others. 2022. "They're Not That Hard to Mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *S&P '22*. IEEE, 632–649. doi:10.1109/SP46214.2022.9833713
[51] Sungkeun Kim, and others. 2020. ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage. In *SecDev '20*. IEEE, 96–107. doi:10.1109/SecDev45635.2020.00029
[52] Vladimir Kiriansky, and others. 2018. DAWG: A Defense against Cache Timing Attacks in Speculative Execution Processors. In *MICRO '18*. IEEE, 974–987. doi:10.1109/MICRO.2018.00083
[53] Steve Klabnik. 2015. Annotate Blocks That Must Run in Constant Time Regardless of Inputs · Issue #847 · Rust-Lang/Rfcs. GitHub. https://github.com/rust-lang/rfcs/issues/847 (accessed 2025-03-21).
[54] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press.
[55] klutzy. 2015. Nadeko. https://github.com/klutzy/nadeko
[56] Paul Kocher, and others. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P '19*. 1–19. doi:10.1109/SP.2019.00002
[57] Esmaeil Mohammadian Koruyeh, and others. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *WOOT '18*. USENIX. https://www.usenix.org/conference/woot18/presentation/koruyeh
[58] Cody Laeder. 2016. Include Constant Time Integer Operation inside Core · Issue #1814 · Rust-Lang/Rfcs. GitHub. https://github.com/rust-lang/rfcs/issues/1814 (accessed 2025-03-21).
[59] Adam Langley. 2025. Ctgrind. https://github.com/agl/ctgrind
[60] Moritz Lipp, and others. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security '18*. USENIX, 19. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
[61] Chang Liu, and others. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS '15*. ACM, 87–101. doi:10.1145/2694344.2694385
[62] Kevin Loughlin, and others. 2021. Dolma: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security '21*. USENIX, 1397–1414.
[63] lowRISC. 2025. Ibex Security Features. Ibex Documentation. https://ibex-core.readthedocs.io/en/latest/03_reference/security.html (accessed 2025-07-26).
[64] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS '18*. ACM. doi:10.1145/3243734.3243761
[65] Masaryk University Centre for Research on Cryptography and Security. 2025. CT-Tools. https://crocs-muni.github.io/ct-tools/ (accessed 2025-04-07).
[66] Tim McLean. 2023. Rust-Timing-Shield. https://github.com/timmclean/rust-timing-shield
[67] Tim McLean. 2023. Rust-Timing-Shield - Security. https://www.chosenplaintext.ca/open-source/rust-timing-shield/security/ (accessed 2025-04-01).
[68] Pratyush Mishra, and others. 2018. Oblix: An Efficient Oblivious Search Index. In *S&P 18*. IEEE, 279–296. doi:10.1109/SP.2018.00045
[69] David Molnar, and others. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC '05*. Springer-Verlag, 156–168. doi:10.1007/11734727_14
[70] NCSC. 2024. Raising the Cyber Resilience of Software 'at Scale'. UK National Cyber Security Centre blog. https://www.ncsc.gov.uk/blog-post/raising-cyber-resilience-software-at-scale (accessed 2025-03-30).
[71] Rishiyur S. Nikhil. 2025. Riscy-OOO. CSAIL CSG. https://github.com/csail-csg/riscy-OOO
[72] Olga Ohrimenko, and others. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security '16*. USENIX, 619–636.
[73] ONCD. 2024. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Technical Report. US White House Office of the National Cyber Director. https://web.archive.org/web/20250118014817/https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
[74] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA '06*. Springer-Verlag, 1–20. doi:10.1007/11605805_1
[75] Louis-Emile Ploix, and others. 2025. Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor. arXiv:2502.04738 [cs] doi:10.48550/arXiv.2502.04738
[76] Michal Podhradsky, Ramy Tadros, and Austin Roach. 2022. BESSPIN-GFE. Galois, Inc. https://github.com/GaloisInc/BESSPIN-GFE
[77] Thomas Pornin. 2018. BearSSL - Constant-Time Crypto. https://www.bearssl.org/constanttime.html (accessed 2025-04-04).
[78] Thomas Pornin. 2018. BearSSL - Constant-Time Mul. https://bearssl.org/ctmul.html (accessed 2025-04-04).
[79] Thomas Pornin. 2025. Constant-Time Code: The Pessimist Case. https://eprint.iacr.org/2025/435
[80] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security '15*. USENIX, 431–446.
[81] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, Is My Code Constant Time?. In *DATE '17*. European Design and Automation Association, 1701–1706.
[82] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *MICRO '19*. ACM, 73–86. doi:10.1145/3352460.3358314
[83] Christos Sakalis, and others. 2019. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *ISCA '19*. ACM, 723–735. doi:10.1145/3307650.3322216
[84] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS '18*. Internet Society. doi:10.14722/ndss.2018.23239
[85] Moritz Schneider, and others. 2024. Breaking Bad: How Compilers Break Constant-Time~Implementations. arXiv:2410.13489 [cs] doi:10.48550/arXiv.2410.13489
[86] Michael Schwarz, and others. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS '20*. Internet Society. doi:10.14722/ndss.2020.24271
[87] Fahad Shaon, and others. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *CCS '17*. ACM, 1211–1228. doi:10.1145/3133956.3134095
[88] Nikhil Swamy, and others. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL '16*. ACM, 256–270. doi:10.1145/2837614.2837655
[89] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ASPLOS '19*. ACM, 395–410. doi:10.1145/3297858.3304060
[90] Robert N M Watson, and others. 2019. *An Introduction to CHERI*. Technical Report UCAM-CL-TR-941. Computer Laboratory, University of Cambridge. 43 pages pages. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf
[91] Robert N. M. Watson, and others. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. Computer Laboratory, University of Cambridge. 523 pages pages. doi:10.48456/TR-987
[92] Robert N M Watson, and others. 2020. *CHERI C/C++ Programming Guide*. Technical Report UCAM-CL-TR-947. Computer Laboratory, University of Cambridge. 33 pages. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf
[93] Samuel Weiser, and others. 2018. DATA–differential Address Trace Analysis: Finding Address-Based Side-Channels in Binaries. In *USENIX Security '18*. USENIX, 603–620.
[94] Nathaniel Wesley Filardo, and others. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *S&P '20*. IEEE, 608–625. doi:10.1109/SP40000.2020.00098
[95] Jan Wichelmann, and others. 2018. MicroWalk: A Framework for Finding Side Channels in Binaries. In *ACSAC '18*. ACM, 161–173. doi:10.1145/3274694.3274741
[96] Jan Wichelmann, and others. 2022. Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications. In *CCS '22*. ACM, 2915–2929. doi:10.1145/3548606.3560654
[97] Hans Winderix, and others. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *CCS '24*. ACM, 19–33. doi:10.1145/3658644.3690319
[98] Hans Winderix, and others. 2024. Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs. In *S&P '24*. IEEE, 3697–3715. doi:10.1109/SP54263.2024.00047
[99] Jonathan Woodruff, and others. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (Oct. 2019), 1455–1469. doi:10.1109/TC.2019.2914037
[100] Mengjia Yan, and others. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO '18*. IEEE, 428–441. doi:10.1109/MICRO.2018.00042
[101] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security '24*. USENIX, 719–732.
[102] Jiyong Yu, and others. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS '19*. Internet Society. doi:10.14722/ndss.2019.23061

[103] Jiyong Yu, and others. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO '52*. ACM, 954–968. doi:10.1145/3352460.3358274
[104] Sizhuo Zhang. 2025. Toooba. Bluespec, Inc. https://github.com/bluespec/Toooba
[105] Zirui Neil Zhao, and others. 2020. Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis. In *MICRO '20*. IEEE, 1138–1152. doi:10.1109/MICRO50266.2020.00094
[106] Wenting Zheng, and others. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI'17*. USENIX, 283–298.
[107] Jean-Karim Zinzindohoué, and others. 2017. HACL*: A Verified Modern Cryptographic Library. In *CCS '17*. ACM, 1789–1806. doi:10.1145/3133956.3134043