

CapOpt: Capability-Aware Superoptimization for Secure and Provably Faster Code

XIAOYANG SUN, University of Leeds, United Kingdom
DEJICE JACOB, University of Glasgow, United Kingdom
HUANTING WANG, University of Leeds, United Kingdom
JEREMY SINGER, University of Glasgow, United Kingdom
ZHENG WANG, University of Leeds, United Kingdom

Superoptimization is a powerful code optimization technique that generates optimized instruction sequences by exploring the space of instruction-level transformations. However, existing superoptimizers assume that pointers and integers are interchangeable, an assumption that no longer holds in memory-security-enhanced architectures like CHERI, where pointers are represented as metadata-rich capabilities with enforced bounds, permissions, and provenance. This semantic change breaks many traditional optimizations and forces CHERI compilers to adopt conservative strategies that sacrifice performance for safety. We present CapOpt, the first superoptimization framework that explicitly incorporates capability semantics into both its search space and correctness model. CapOpt introduces Provenance-Guided Stratified Synthesis (PGSS), a synthesis strategy that structures the search space around capability-aware abstractions and uses provenance-based reasoning to eliminate unsafe transformations. We also define a capability-aware equivalence model that extends conventional functional correctness to include metadata integrity. We evaluated CapOpt on an ARM-based CHERI hardware platform and the CHERI-RISC-V simulator. Experimental results show that CapOpt improves performance by up to 4.1% over the existing CHERI-LLVM toolchain, while strengthening security by tightening pointer bounds and permissions.

CCS Concepts: • **Security and privacy** → **Security in hardware**; • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: Superoptimization, CHERI, Compiler, Code Generation, Memory Safety

ACM Reference Format:
Xiaoyang Sun, Dejice Jacob, Huanting Wang, Jeremy Singer, and Zheng Wang. 2026. CapOpt: Capability-Aware Superoptimization for Secure and Provably Faster Code. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 349 (October 2026), 28 pages. https://doi.org/10.1145/3839481

1 Introduction

Superoptimization [30] is a powerful technique for finding high-performance instruction sequences by searching for equivalent program transformations [5, 20, 41, 49, 50]. Traditional superoptimizers such as STOKE [46, 47] and Souper [45] are built around conventional machine and compiler semantics, where pointer values can often be represented or manipulated as integer-like values. This allows optimizers to reason about addresses as plain 32-bit or 64-bit values and, where permitted, transform code using pointer-integer conversions [23].

Authors' Contact Information: Xiaoyang Sun, University of Leeds, Leeds, United Kingdom, X.Sun4@leeds.ac.uk; Dejice Jacob, University of Glasgow, Glasgow, United Kingdom, Dejice.Jacob@glasgow.ac.uk; Huanting Wang, University of Leeds, Leeds, United Kingdom, H.Wang7@leeds.ac.uk; Jeremy Singer, University of Glasgow, Glasgow, United Kingdom, Jeremy.Singer@glasgow.ac.uk; Zheng Wang, University of Leeds, Leeds, United Kingdom, Z.Wang5@leeds.ac.uk.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART349
https://doi.org/10.1145/3839481

Prior work has introduced pointer *provenance* into C memory models to record the origin and permitted use of pointers [4, 24, 31]. Modern compiler analyses can use such information to reason about pointer relationships and restrict transformations involving unsafe pointer–integer conversions. However, conventional machine-level representations do not retain the rich metadata associated with pointer provenance. At the binary level of conventional ISAs, where superoptimizers such as STOKE operate, a pointer is ultimately represented as an ordinary value held in a register.

Emerging capability-enabled architectures such as CHERI[1] fundamentally change this model. CHERI extends conventional architectures with fine-grained memory protection and secure compartmentalization [35]. Instead of unstructured integer addresses, CHERI represents pointers as *capabilities*: protected values containing an address together with metadata such as bounds, permissions, and a validity tag maintained by hardware. The tag protects capability integrity and provenance: it is preserved only through valid capability operations and is cleared when a capability is modified in an unsafe manner. As a result, capabilities can only be used within their authorized bounds and permissions.

This richer pointer representation creates a mismatch with many traditional compiler transformations. Converting a capability to an integer and later reconstructing it as a capability does not, in general, preserve its provenance. Even if the resulting numerical address remains unchanged, the reconstructed capability may lose its validity tag, causing a hardware fault. Transformations that are correct on conventional ISAs can therefore be unsafe on CHERI, either producing invalid capabilities [60] or directly causing capability invalidation [12].

For example, the LLVM compiler provides the `CreateBitOrPointerCast()` utility function for generating conversions between pointer and integer types. Such pointer-to-integer-to-pointer transformations cannot, in general, be applied unchanged to CHERI capabilities because they may violate capability provenance. CHERI-LLVM therefore adopts conservative strategies for transformations that could violate capability semantics [57, 60]. While necessary for correctness, these restrictions can limit optimization opportunities. Existing studies of CHERI systems report performance overheads partly associated with additional capability operations and less optimized code sequences [53, 58].

We introduce CapOpt[2], the first superoptimization framework designed specifically for capability architectures such as CHERI. CapOpt is an *offline* superoptimizer intended for ahead-of-time optimization. It operates at both LLVM intermediate representation (IR) and assembly levels and can be inserted as additional stages in an existing compilation pipeline (§4). The verified rewrites discovered by CapOpt can be applied directly to programs or used to inform the design of future compiler optimizations.

Our key insight is that some of the performance overhead in current CHERI software is not an inherent cost of hardware-enforced security, but arises because conventional compiler optimizations do not fully exploit capability-aware semantics. By extending program equivalence to account for capability metadata and provenance, CapOpt can search for more efficient transformations while preserving CHERI's security guarantees.

At the core of CapOpt is a two-level capability-aware superoptimizer. The IR-level stage captures high-level algebraic and data-flow simplifications, while the assembly-level stage exploits ISA-specific capability instructions and addressing modes. To make the resulting search space tractable, we introduce *Provenance-Guided Stratified Synthesis (PGSS)*. PGSS organizes candidate transformations into semantic strata and explores simpler capability-preserving operations before considering

[1]CHERI refers to Capability Hardware Enhanced RISC Instructions [57, 59].
[2]Code and data are available at https://github.com/oopslacapopt.

more complex ones. It further uses provenance-guided pruning, including capability-aware alias reasoning, to eliminate unsafe or unproductive candidates before formal verification.

We also introduce a capability-aware correctness model that extends conventional functional equivalence with constraints on capability metadata and provenance. A valid rewrite must preserve functional behavior, must not widen capability bounds or grant additional permissions, and must preserve capability validity and provenance. Each candidate rewrite is checked using an SMT solver before it is accepted. This provides a formal basis for safely applying superoptimization to CHERI and, more generally, to capability-based architectures.

We evaluated CapOpt on the CHERI-enabled Arm Morello platform [7, 16] and a CHERI-RISC-V simulator [44], using 12 pointer-intensive benchmarks from SPEC CPU2017 [8], bit-manipulation kernels from Hacker's Delight [17, 56], and the `LLaMA.cpp` inference engine [26]. On Morello, CapOpt improves the performance of CHERI-LLVM-generated code by up to 4.1%, with a geometric-mean improvement of approximately 2.5% for the evaluated SPEC CPU2017 benchmarks. Relative to the non-CHERI `-O3` baseline, CapOpt thereby recovers part of the CHERI capability-enforcement overhead (§6). Our ablation study further shows that combining IR- and assembly-level optimization outperforms either level alone on the pointer-intensive workloads.

This work makes the following contributions:

- The first superoptimization framework designed for CHERI capability systems, combining LLVM IR- and assembly-level optimization;
- Provenance-Guided Stratified Synthesis (PGSS), which structures and prunes the capability-aware transformation space using provenance information;
- A capability-aware equivalence model that formally constrains functional behavior, capability authority, and provenance during optimization.

2 Background and Motivation

2.1 CHERI Capabilities and Constraints

CHERI changes the conventional representation of pointers by introducing a hardware-enforced type, the *capability* [22, 59]. A capability is a *fat pointer* that combines an address with metadata enforced by hardware. This metadata includes *bounds*, which define the memory region that the capability may access; *permissions*, which specify allowed operations such as load, store, and execute; and a hidden *tag* bit indicating whether the capability is valid. Hardware checks these properties when the capability is used and prevents accesses that exceed its bounds or permissions. The tag protects capability integrity: it can be set only through valid capability derivation and is cleared when a capability is modified in a way that violates the architecture's rules.

At the machine level of conventional ISAs, pointers are typically represented as numerical addresses without hardware-enforced bounds, permissions, or provenance metadata. As a result, many traditional compiler optimizations manipulate addresses using integer arithmetic or transform code across pointer-integer representations.

CHERI invalidates this assumption because capability metadata must remain consistent across program transformations. Three CHERI properties are particularly important in the context of pointer-integer conversion. First, *unforgeability* prevents valid capabilities from being constructed from arbitrary data: only capability-aware operations preserve the validity tag. Second, *monotonicity* ensures that capability authority cannot be increased through derivation. For example, a derived capability may narrow its bounds or remove permissions, but it cannot gain access beyond that of its parent. Third, *provenance* records the valid derivation of a capability from an existing capability authority. Arbitrary capability-to-integer-to-capability transformations, therefore, cannot, in general, reconstruct the original capability authority and may result in an invalid capability.

Violating these properties breaks CHERI's hardware-enforced safety guarantees. Therefore, a CHERI-aware optimizer must account for capability metadata and provenance when transforming code, rather than reasoning only about numerical addresses.

2.2 Superoptimization

Superoptimization treats code optimization as a search or synthesis problem. Given a code fragment, often a straight-line basic block, a superoptimizer searches for an equivalent sequence with lower cost according to a specified metric, such as latency, instruction count, energy consumption, or code size. Early systems used exhaustive *generate-and-test* search for very short instruction sequences [30]. Later approaches include brute-force generation of peephole rules [5, 6], stochastic search as used by STOKE [46, 47], and SMT-based synthesis as used by Souper [45]. Solver-based tools are also widely used to check compiler transformations. For example, Alive2 [29] performs bounded translation validation for LLVM IR, demonstrating that SMT-based reasoning can support realistic compiler semantics.

At the machine level, superoptimization typically requires a candidate sequence to preserve the relevant observable register, memory, and side-effect behavior of the original sequence. At the compiler IR level, correctness is usually expressed as a refinement relation that also accounts for undefined behavior.

CHERI adds an additional dimension to this correctness requirement. Two instruction sequences may compute the same numerical addresses while differing in the bounds, permissions, validity, or provenance of the resulting capabilities. A capability can be dereferenced only when its tag is valid, and capability metadata can be safely modified only through operations that satisfy CHERI's provenance and monotonicity rules [12, 61]. Consequently, equivalence based only on ordinary register and memory values is insufficient for CHERI [22, 59, 60].

CHERI capability semantics affect optimization in two ways. First, transformations that are valid under conventional pointer representations may become unsound if they do not account for capability metadata, provenance, or authority. Second, transformations that are profitable on conventional ISAs may become less effective on CHERI because preserving capability semantics can require additional instructions. For example, representing pointer induction through integer addresses and subsequently reconstructing capability values may introduce repeated capability-manipulation operations. A CHERI-aware optimizer should instead preserve capability-native representations whenever possible.

CHERI thus requires superoptimization to reason explicitly about capability semantics. The search space must include capability-native transformations, such as bounds and permission refinement, capability-based induction variables, and capability-aware addressing operations. At the same time, candidate transformations that invalidate tags, violate provenance, or increase capability authority must be rejected.

2.3 Motivating Example

We use a simple array-copy example to illustrate the optimization opportunities introduced by CHERI and why capability-aware optimization is needed. Consider the `elementwise_copy` function in Listing 1. The function initializes one fixed-size array, copies its elements to a second array, and finally reads one element to keep the computation observable.

Standard RISC-V (without CHERI). The copy loop `for (int i = 0; i < 64; ++i) b[i] = a[i];` is a common optimization target. When dependence and alias analyses establish the required preconditions, modern compilers such as LLVM can recognize such loops as memory-copy idioms or optimize them using transformations such as vectorization, unrolling, and induction-variable

```c
void elementwise_copy(int argc, char **argv) {
    int a[64], b[64];
    for (int i = 0; i < 64; ++i) a[i] = i;
    // ... (other operations on a[]) ...
    for (int i = 0; i < 64; ++i) b[i] = a[i];
    // ... (other operations on b[]) ...
    volatile int sink = b[0];
    // ... (other operations) ...
    return;
}
```

Listing 1. The `elementwise_copy` function in C/C++.

simplification [27, 28]. In our baseline RISC-V configuration, LLVM produces the compact scalar load/store loop shown in Listing 2. [3]

```
# Standard RISC-V (without CHERI)
# 11 core instructions

...

    beqz    a2, .LBB0_3
.LBB0_1:
    lw      a3, 0(a1)
    sw      a3, 0(a0)
    addi    a2, a2, -1
    addi    a0, a0, 4
    addi    a1, a1, 4
    bnez    a2, .LBB0_1
.LBB0_2:
    addi    sp, sp, -16
    sw      zero, 12(sp)
    lw      a0, 12(sp)
    addi    sp, sp, 16
    ret
.LBB0_3:
    ret
```

Listing 2. RISC-V

```
# Purecap CHERI-RISC-V
# 143 core instructions
.LBB0_1:
    cincoffset  csp, csp, -528
    li      a0, 0
    sw      zero, 272(csp)
    li      a1, 1
    sw      a1, 276(csp)
    ... (hidden 122 lines) ...  # unfold
    li      a1, 63
    sw      a1, 524(csp)
    cincoffset  ca1, csp, 272
    csetbounds  ca1, ca1, 256
    cincoffset  ca2, csp, 16
    csetbounds  ca2, ca2, 256
    li      a3, 64
.LBB0_2:
    slli    a4, a0, 2
    cincoffset  ca5, ca1, a4
    lw      a5, 0(ca5)
    cincoffset  ca4, ca2, a4
    addi    a0, a0, 1
    sw      a5, 0(ca4)
    bne     a0, a3, .LBB0_1
    lw      a0, 16(csp)
    sw      a0, 12(csp)
    lw      a0, 12(csp)
    cincoffset  csp, csp, 528
    ret
```

Listing 3. CHERI-RISC-V

Additional operations under CHERI. Listings 2 and 3 compare the generated code. Conventional RISC-V uses a compact pointer-walking loop. In contrast, CHERI-LLVM (pure-capability, `-O3`) emits additional instructions to establish capability bounds and repeatedly derives element capabilities from base capabilities and integer offsets. Although this code preserves CHERI semantics, it leaves opportunities for capability-specific optimization. Rather than recomputing the capability corresponding to each array element, the loop can maintain capability induction variables for the source and destination pointers and advance them directly using capability-aware offset operations, described as follows.

A capability-aware optimization. Listing 4 illustrates a hand-tuned capability-walk implementation. At the source level, the transformation corresponds to the pattern `while (src != end) *dst++ = *src++;`. The end capability is computed once, while `src` and `dst` are advanced using

[3] The exact set of compiler passes and the emitted instruction sequence is target-dependent.

```
# Optimized Purecap CHERI-RISC-V
# 14 core instructions
    beqz    a2, .LBB0_3
    slli    a2, a2, 2
    cincoffset  ca2, ca1, a2
.LBB0_1:
    lw      a3, 0(ca1)
    cincoffset  ca4, ca1, 4
    cincoffset  ca1, ca0, 4
    sw      a3, 0(ca0)
    cmove   ca0, ca1
    cmove   ca1, ca4
    bne     a4, a2, .LBB0_1
.LBB0_2:
    cincoffset  csp, csp, -16
    sw      zero, 12(csp)
    lw      a0, 12(csp)
    cincoffset  csp, csp, 16
    ret
.LBB0_3:
    ret
```

Listing 4. Optimized CHERI-RISC-V for Listing 1.

Algorithm 1 CapOpt's two-stage optimization pipeline

Require: LLVM IR basic block B_{IR}, semantic strata $\mathcal{S} = \{\text{S1}, \text{S2}, \text{S3}, \text{S4}\}$
Ensure: Optimized assembly block $B^{\star}$ satisfying capability-aware equivalence
1: $B'_{\text{IR}} \leftarrow \text{PGSS}(B_{\text{IR}}, \mathcal{S}, \text{IR})$ ▷ *IR-level synthesis*
2: $B_{\text{asm}} \leftarrow \text{LowerToAssembly}(B'_{\text{IR}})$
3: $B^{\star} \leftarrow \text{PGSS}(B_{\text{asm}}, \mathcal{S}, \text{asm})$ ▷ *ISA-specific synthesis*
4: **return** $B^{\star}$

provenance-preserving capability-offset operations. This avoids repeatedly deriving an element capability from a base capability and an integer index.

For this example, the capability-walk form reduces the reported number of core instructions from 143 to 14, using approximately one-tenth as many core instructions as the CHERI-LLVM output. More importantly, it illustrates an optimization opportunity that is specific to capability-based execution: the numerical address computation can be simplified while retaining capability provenance and authority.

This example shows that CHERI-LLVM leaves opportunities for capability-specific optimization. CapOpt is designed to systematically discover such transformations while ensuring that they preserve CHERI semantics. Doing so requires both a capability-aware transformation space and an equivalence model that accounts for capability metadata and provenance.

3 CapOpt: A Capability-Aware Superoptimizer

CapOpt adopts a two-stage superoptimization pipeline that operates at the LLVM intermediate representation (IR) level and the assembly level, as shown in Figure 1. In the first stage, CapOpt applies high-level transformations to individual basic blocks in LLVM IR. In the second stage, the optimized IR is lowered to assembly, where CapOpt applies ISA-specific transformations that exploit CHERI instructions and addressing modes. This division allows CapOpt to capture both high-level algebraic and data-flow simplifications and low-level improvements that depend on CHERI-specific architectural features.

LLVM IR-level optimization. CapOpt's IR-level optimizer operates on LLVM IR in Static Single Assignment (SSA) form [10, 11], which makes data flow and value dependencies explicit while

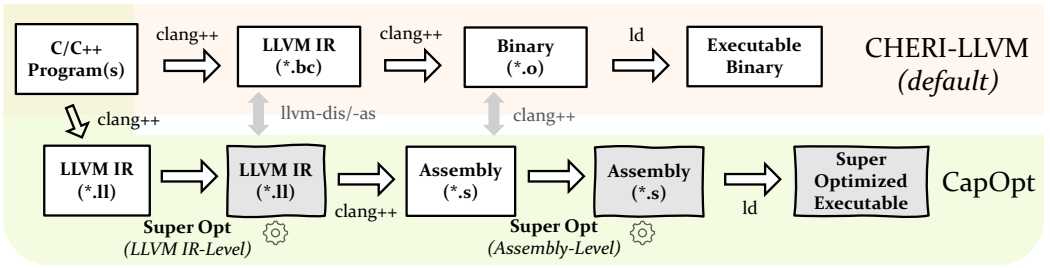


Fig. 1. Compilation workflows for CHERI-LLVM and CapOpt.

abstracting away physical registers and instruction encodings. Operating at this level provides a structured search space in which capability derivations can be tracked directly and high-level redundancies can be identified before lowering to machine instructions.

For example, the CHERI-LLVM compiler may represent capability address manipulation as a sequence that conceptually obtains the capability's address, performs arithmetic on the resulting integer, and then updates the capability address. At the IR level, these operations are represented through the corresponding CHERI intrinsics and data-flow operations. CapOpt can follow their dependencies and provenance and determine when such a sequence can be replaced by a direct capability-aware offset operation. The resulting transformation avoids unnecessary conversion through integer address values while preserving the capability's validity and authority.

Assembly-level optimization. After IR-level optimization, the improved IR is lowered to target assembly and processed by CapOpt's second stage. At this level, the optimizer can exploit details that are not visible in LLVM IR, including ISA-specific capability instructions, addressing modes, and instruction combinations.

For example, the CHERI-RISC-V architecture implementation provides capability-aware forms of operations that are represented only indirectly at the IR level. Similarly, assembly-level optimization can exploit instructions such as capability loads, stores, offset updates, and control-transfer operations when these produce a lower-cost sequence than the code generated by the compiler. The IR stage, therefore, removes high-level redundancies, while the assembly stage further refines the generated code using target-specific opportunities.

Both stages operate on one basic block at a time. This single-block scope is a deliberate trade-off. Many CHERI-related inefficiencies, including repeated address derivation, redundant bounds or permission operations, and capability-walk patterns, occur within straight-line code. Restricting synthesis to basic blocks captures these opportunities while keeping both candidate generation and verification tractable. We discuss extending CapOpt to multiple basic blocks in §8.

3.1 Formalizing Capability-Aware Equivalence

A key requirement for capability-aware superoptimization is a correctness relation that captures both program behavior and CHERI capability semantics. In our model, a candidate rewrite R is a correct replacement for an original instruction sequence S if, for every valid initial machine state σ, three conditions hold: (i) functional equivalence or refinement, (ii) non-expansion of capability authority, and (iii) preservation of capability validity and provenance.

Let Σ denote the set of possible machine states, and let $Exec(P, \sigma)$ denote the state obtained after executing a program fragment P from $\sigma \in \Sigma$. Each machine state contains ordinary registers, memory, and a set of capabilities C. We represent a capability $c \in C$ as

$$c = \langle addr(c), base(c), bounds(c), perms(c), otype(c), tag(c) \rangle, \tag{1}$$

where $addr(c)$ is the current address or cursor, $base(c)$ and $bounds(c)$ describe the accessible memory region, $perms(c)$ records the permission bits, $otype(c)$ represents the sealing type, and $tag(c)$ indicates whether the capability is valid.

A rewrite R is defined to be *capability-equivalent* to S when the following conditions are satisfied.

(i) Functional equivalence. Let $\text{proj}_{\text{noncap}}$ extract the observable non-capability state, including relevant ordinary register values, memory effects, and other architecturally visible behavior. At the assembly level, the source and rewrite must agree on these observable outputs:

$$\forall \sigma \in \Sigma, \quad \text{proj}_{\text{noncap}}\big(Exec(S, \sigma)\big) = \text{proj}_{\text{noncap}}\big(Exec(R, \sigma)\big). \tag{2}$$

CapOpt also treats architecturally visible memory operations as observable effects: a rewrite must preserve the required loads and stores, including their order where observable. This ensures that source-level constraints such as `volatile`, which are reflected in the emitted memory operations, remain valid after optimization.

At the LLVM IR level, exact equality is replaced by an Alive2-style refinement relation because LLVM values may contain `poison` or `undef`. The rewritten program may therefore be more defined than the original, but it must not introduce behavior that is absent from the source program.

(ii) Capability authority restriction. For each observable capability c_S produced by S and its corresponding capability c_R in R, the rewrite must not introduce additional authority. In particular,

$$\begin{aligned} addr(c_S) &= addr(c_R), \\ base(c_S) &= base(c_R), \\ bounds(c_R) &\subseteq bounds(c_S), \\ perms(c_R) &\subseteq perms(c_S), \\ otype(c_S) &= otype(c_R). \end{aligned} \tag{3}$$

These constraints allow a rewrite to preserve or reduce capability authority, but never to widen the accessible region or add permissions. Any authority removed from a live-out capability must be irrelevant to the observable behavior covered by the rewrite's refinement obligation; otherwise, the candidate is rejected. The SMT verifier, therefore, prevents CapOpt from accepting transformations that remove authority required by subsequent observable behavior.

These conditions are checked before a rewrite is applied. CapOpt does not rely on CHERI hardware to detect an incorrect transformation at runtime.

(iii) Provenance validity preservation. Corresponding capabilities must also preserve validity. For each observable corresponding pair,

$$tag(c_S) = tag(c_R). \tag{4}$$

A capability that is valid in the source must remain valid in the rewrite, while an invalid capability must not be made valid by the transformation. Together with the derivation constraints used by the verifier, this prevents a rewrite from introducing a capability whose provenance is not supported by the source computation.

When all three conditions hold, we write $R \equiv_{\text{CHERI}} S$.

Capability correspondence. To compare capability values produced by S and R, CapOpt defines a correspondence relation $\mathcal{M} \subseteq C_S \times C_R$ between capability inputs and observable outputs. At the LLVM IR level, this relation is derived from SSA data flow and capability-derivation chains. At the

Table 1. Semantic strata in PGSS with representative CHERI instructions.

Stratum	Semantic Principle	Description	Examples
S_0	Integer & Metadata Inspection	Ordinary computation and capability-metadata inspection that do not modify capability authority. These operations are always available as the reasoning baseline.	`add`, `sub`, `CGetTag`
S_1	Address & Bounds Update	Updates a capability address or narrows its bounds while preserving valid derivation.	`CIncOffset`, `CSetBounds`, `CSetAddr`
S_2	Permission & Flag Update	Restricts permissions or updates capability flags without adding authority.	`CAndPerm`, `CSetFlags`
S_3	Sealing & Unsealing	Operations that create, inspect, or remove sealing relationships.	`CSeal`, `CUnseal`, `CSealEntry`
S_4	Memory & Control Transfer	Capability-based memory accesses and control-flow operations.	`CLoad`, `CStore`, `CInvoke`

assembly level, correspondence is established from symbolic register definitions and capability dependencies. This allows a rewrite to replace several source instructions with a different sequence while still identifying capability values that have the same semantic role.

Verification. We use the Z3 SMT solver [13] to verify these conditions. CapOpt encodes CHERI operations as symbolic transformations over the machine-state representation above. Each instruction updates ordinary state and the relevant capability fields according to its semantics.

The CHERI instruction semantics are cross-validated against the official CHERI Sail specification [15], while the capability-free LLVM IR semantics are differentially tested against Alive2 (§4). During verification, S and R are symbolically executed from the same arbitrary initial state σ, producing formulas ϕ_S and ϕ_R. The verifier asks whether there exists an initial state for which the required functional or refinement condition, capability-authority constraints, or validity constraints are violated. If Z3 proves that no such counterexample exists, R is accepted.

3.2 Provenance-Guided Stratified Synthesis (PGSS)

CHERI substantially enlarges the space of possible instruction transformations. CHERI-enabled ISAs such as CHERI-RISC-V and Arm Morello introduce operations for inspecting and updating addresses, bounds, permissions, sealing state, and other capability metadata. Even short instruction sequences can therefore have a large number of possible alternatives, many of which are invalid or unnecessary.

To make this search tractable, CapOpt uses *Provenance-Guided Stratified Synthesis (PGSS).* PGSS organizes operations according to their semantic complexity and progressively expands the set of transformations considered during synthesis. It also uses provenance information to eliminate candidates that introduce invalid capability derivations before invoking the SMT verifier.

3.2.1 Stratified search by semantic complexity. CapOpt organizes the search space into the semantic strata S_0–S_4 shown in Table 1. S_0 contains ordinary arithmetic and metadata-inspection operations that form the baseline vocabulary available throughout synthesis. PGSS then progressively enables transformations from S_1 through S_4.

The lower strata contain common capability transformations such as address adjustment, bounds restriction, and permission masking. Higher strata introduce semantically more complex operations involving sealing, memory access, and capability-based control transfer. If a lower-stratum transformation is sufficient to reduce the cost of a region, PGSS does not need to introduce operations from higher strata.

This ordering reduces the branching factor by delaying semantically complex operations until they are needed. In our workloads, most profitable capability-specific rewrites occur in S_1 and S_2, so this organization substantially limits the number of candidates that reach the later stages of synthesis and verification.

3.2.2 Layered synthesis by semantic strata. Algorithm 2 describes the PGSS process. Given a code block B, PGSS searches for a lower-cost semantics-preserving rewrite by progressively exploring $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$. The parameter `level` specifies whether the optimizer is operating on LLVM IR or assembly and selects the corresponding instruction vocabulary, rewrite library, and semantic model. Operations from S_0 remain available at every level and do not form a separate expansion step. The stratification also supports the reuse of verified transformations. A higher-stratum candidate can use lower-stratum operations and previously verified rewrite patterns as building blocks. For example, transformations involving sealing or memory access may depend on simpler address or permission manipulations that have already been verified. This reduces repeated synthesis and verification work without changing the final correctness requirement: every accepted rewrite must still satisfy the capability-aware equivalence check.

Candidate enumeration and greedy application. `EnumerateCandidates` in Algorithm 2 constructs candidate rewrites by filling typed instruction holes. Opcodes are selected from the operation set enabled at the current stratum and from the verified pattern library $\mathcal{L}$, while operands are selected from values available within the current block. Candidate length is bounded by rewrite templates, avoiding unrestricted enumeration of arbitrary instruction sequences.

Across strata, the search is greedy. Whenever a verified candidate lowers the cost, CapOpt applies the rewrite and restarts from the first active stratum (Algorithm 2, lines 13 and 17).

CapOpt does not claim global optimality. Unlike bounded-exhaustive superoptimizers, which may prove that no cheaper sequence exists within a fixed search bound, CapOpt guarantees that each accepted rewrite satisfies the capability-aware correctness condition and improves the optimization objective used during search. This is a deliberate trade-off: stratified greedy search keeps per-block synthesis practical while still finding useful transformations on the programs evaluated in §6.

3.3 Provenance-Guided Pruning via Alias Analysis

PGSS further reduces the search space using capability provenance. Conventional alias analysis asks whether two pointers may refer to the same memory location. Under CHERI, however, two capabilities may contain the same numerical address while differing in their bounds, permissions, or derivation history. Address equality alone is therefore insufficient to determine whether they may safely participate in the same capability derivation.

For each basic block, CapOpt maintains a directed acyclic graph (DAG) describing capability derivations. Each node represents a capability value, and an edge from C_A to C_B indicates that C_B is derived from C_A, for example through `CIncOffset`, `CSetBounds`, or a capability copy. Two capabilities are marked as provenance-disjoint only when the graph establishes that they originate from distinct source capabilities. If their ancestry is not visible within the block, as is often the case for live-in capabilities, CapOpt conservatively treats their relationship as unknown.

During synthesis, `ProvenanceAliasPrune` in Algorithm 2 examines how a candidate uses capability values. If two capabilities are known to come from disjoint provenance lineages, PGSS rejects candidates that use one capability as an input to the derivation of the other or otherwise introduce a provenance relationship absent from the source program. Candidates involving capabilities with a shared ancestor, or with unknown ancestry, are retained for formal verification.

Algorithm 2 PGSS: Provenance-Guided Stratified Synthesis

Require: Block B, strata $\mathcal{S}$ (ordered from simplest/safest to most complex), level $\in$ {IR, ASM}
Ensure: Rewritten block B' with reduced cost, preserving capability semantics

$\mathcal{L} \leftarrow$ LoadPatternLibrary(level) ▷ *Load library of verified rewrite patterns*
$G \leftarrow$ BuildCapAliasDAG(B) ▷ *Nodes: capability values; edges: derivation (parent → derived)*
$B' \leftarrow B$; $improved \leftarrow$ **true**
while *improved* **do**
  $improved \leftarrow$ **false**
  for $s \in \mathcal{S}$ **do** ▷ *Iterate over strata in order*
    $C \leftarrow$ EnumerateCandidates($B', s, \mathcal{L}$)
    $C \leftarrow$ ProvenanceAliasPrune(C, G)
    for $R \in C$ **do** ▷ *R is a local rewrite candidate*
      if CapAwareVerify($B' \Rightarrow R$) **then** ▷ *Check correctness*
        $\Delta \leftarrow$ Cost(R) − Cost(current region)
        if $\Delta < 0$ **then** ▷ *Does the candidate improve the cost?*
          $B' \leftarrow$ Apply(B', R)
          $G \leftarrow$ UpdateCapAliasDAG(G, R)
          $improved \leftarrow$ **true**
    if *improved* **then**
      break ▷ *Restart at first stratum after any successful rewrite (stratified discipline)*
return B'

Function: ProvenanceAliasPrune

Require: Candidates C, capability-alias DAG G
Ensure: Filtered candidates C' consistent with provenance constraints

$C' \leftarrow \emptyset$
for $R \in C$ **do**
  $P \leftarrow$ CombinedPairs(R) ▷ *Pairs (c_i, c_j) where R makes the value of one capability an input to the derivation of the other*
  if $\exists (c_i, c_j) \in P$: DisjointProvenance(c_i, c_j, G) **then**
    continue ▷ *Prune: crossing proven-disjoint lineages clears the tag*
  $C' \leftarrow C' \cup \{R\}$ ▷ *SameLineage and unknown-ancestry pairs are retained*
return C'

Function: CapAwareVerify (SMT-based capability-aware equivalence)

Require: Source region S, candidate rewrite R
Ensure: **true** iff R preserves all functional outputs and capability metadata of S

Encode machine state: integer registers/memory, and fields of each capability (address, base, limit, permissions, tag)
Symbolically execute S and R to obtain formulas Φ_S, Φ_R
Assert equality of all non-capability outputs
Assert for each resulting capability C_S (from S) and C_R (from R):
  base(C_S) = base(C_R);
  bounds(C_R) ⊆ bounds(C_S);
  perms(C_R) ⊆ perms(C_S);
  tag(C_S) = tag(C_R);
Assert monotonicity of derivation for all capability flows in R ▷ *no new rights introduced; valid provenance*
return (Z3 proves $\Phi_S \wedge \Phi_R \Rightarrow$ all assertions hold)

Function: Cost

Require: Rewrite R, weights (w_{lat}, w_{bnd}, w_{perm}, w_{tag})

$c_{\text{lat}} \leftarrow$ EstimateLatency(R)
$c_{\text{bnd}} \leftarrow$ AggregateBoundWidth(live caps after R)
$c_{\text{perm}} \leftarrow$ AggregateGrantedPerms(live caps after R)
$c_{\text{tag}} \leftarrow$ TagOpPenalty(R) ▷ *e.g., penalty for sealing/unsealing operations*
return $w_{\text{lat}} c_{\text{lat}} + w_{\text{bnd}} c_{\text{bnd}} + w_{\text{perm}} c_{\text{perm}} + w_{\text{tag}} c_{\text{tag}}$

This analysis is deliberately conservative. It does not attempt to prove that retained candidates are correct; rather, it removes candidates that can already be shown to violate the structure of the source capability derivations. The remaining candidates are passed to the capability-aware SMT verifier.

In our evaluation, provenance-guided pruning substantially reduces the number of candidates requiring solver checks. For example, when a block contains independent capabilities C_1 and C_2 from distinct derivation lineages, a provenance-unaware search may still generate transformations that attempt to derive one using the other. PGSS removes such candidates before symbolic verification, reducing solver workload without weakening the final correctness check.

3.4 The Cost Model

Traditional superoptimizers commonly rank candidates using performance-related metrics such as instruction count or estimated latency. In a capability architecture, however, transformations may also reduce unnecessary capability authority by narrowing bounds or removing permissions. CapOpt therefore uses a security-aware cost function that combines a performance objective with capability-authority metrics. The cost function is a scalarization rather than a true multi-objective optimization procedure: each candidate receives a single weighted score, and PGSS searches for candidates with lower scores.

We define the cost of a candidate rewrite R as

$$\mathrm{Cost}(R) = w_{\mathrm{lat}} c_{\mathrm{lat}}(R) + w_{\mathrm{bnd}} c_{\mathrm{bnd}}(R) + w_{\mathrm{perm}} c_{\mathrm{perm}}(R) + w_{\mathrm{tag}} c_{\mathrm{tag}}(R). \tag{5}$$

The weights determine the relative importance of the four components, described as follows.

Latency cost, $c_{\mathbf{lat}}(R)$. This component estimates the execution cost of the candidate instruction sequence. The static model uses instruction count and per-operation latency estimates to rank candidates. On Morello, selected candidates may additionally be evaluated using measured hardware cycles. For CHERI-RISC-V under QEMU, which does not provide an accurate model of pipeline timing, we use the retired instruction count rather than the simulated execution latency. This component captures the conventional performance objective of superoptimization.

Bounds-width cost, $c_{\mathbf{bnd}}(R)$. This component penalizes the memory authority represented by live capabilities after the rewrite. A capability with narrower bounds contributes a lower cost than one that can access a larger region. The term therefore allows the search to prefer transformations that remove unnecessary spatial authority, provided that doing so remains consistent with the capability-aware correctness condition.

Permission cost, $c_{\mathbf{perm}}(R)$. This component reflects the permissions retained by live capabilities. A rewrite that safely removes permissions not required by the observable computation receives a lower permission cost. This encourages least-authority representations without allowing a candidate to introduce permissions that are absent from the original program.

Capability-metadata operation cost, $c_{\mathbf{tag}}(R)$. This component penalizes comparatively expensive or semantically complex operations that modify capability metadata, including sealing and unsealing operations such as `CSeal` and `CUnseal`. The purpose of this term is to discourage unnecessary metadata manipulation when a simpler equivalent sequence is available.

The weights $(w_{\mathrm{lat}}, w_{\mathrm{bnd}}, w_{\mathrm{perm}}, w_{\mathrm{tag}})$ allow the search to place different emphasis on execution cost and additional reductions in capability authority. These weights affect candidate ranking only; they cannot relax the capability-equivalence requirements in §3.1. A candidate that widens bounds, adds permissions, introduces invalid provenance, or otherwise violates the correctness condition is rejected regardless of its cost.

In the current implementation, the cost model guides candidate generation and ranking, while the final selected rewrite is also validated empirically against the original code as described in §4. A rewrite that does not provide the required measured performance improvement is not retained. Thus, although the cost model can favor candidates that additionally narrow bounds or remove

permissions, the evaluated configuration does not trade measured performance for additional authority reduction.

By considering capability authority alongside conventional performance metrics, the cost model allows CapOpt to prefer efficient transformations that also remove unnecessary bounds or permissions, while the formal correctness constraints ensure that no accepted rewrite gains authority relative to the original program.

4 Implementation

4.1 System Overview

We implemented CapOpt as a Python package with a C++ backend, which exposes two cooperating optimizers: an *IR-level optimizer* that rewrites LLVM IR and an *assembly-level optimizer* that rewrites CHERI assembly. Both share a verification backend built on the Z3 SMT solver [13], accessed through its Python bindings. CapOpt plugs into existing CHERI toolchains (for both ARM Morello and CHERI-RISC-V) without patching the compiler: the IR optimizer is invoked around `clang` on the emitted IR, and the assembly optimizer runs as a post-compilation rewriting step.

IR-level optimizer. The IR optimizer consumes the LLVM IR that CHERI-LLVM emits and processes it as text, one basic block (straight-line code, no internal control flow) at a time. Rather than registering as in-tree compiler passes, it parses the IR into its own instruction graph, recognizes capability operations through the `@llvm.cheri.cap.*` intrinsic family (so capability metadata is reasoned about directly instead of through integer casts), and lowers `getelementptr` to capability-offset arithmetic. For each block, the PGSS search engine (§3) synthesizes lower-cost, capability-preserving substitutions; every candidate that survives pruning is scored by the security-aware cost model (§3.4) and then discharged by an Alive2-style refinement check (§4.2) that enforces the capability-equivalence conditions (§3.1) before it may replace the original block. Aliasing is resolved by CapOpt's own capability-aware provenance DAG (§3.3) rather than the compiler's alias analysis. The optimized IR then proceeds through LLVM's normal code generation. Memory accesses are handled by *verbatim slicing*: every `alloca`, `load`, and `store` is treated as an opaque, order-paired observable event, and the rewritten block must perform the same event sequence, on provably equal pointers and with equal stored values, and only the pure compute slices between events may be rewritten. This realizes the observability convention of §3.1 directly and lets the verifier check the memory-dominated blocks that `-O0` compilation actually emits, without committing to a full memory model.

Assembly-level optimizer. The second optimizer rewrites the compiled instruction sequence of a function or basic block. It consumes and emits the assembly as text and models the CHERI-ARM (Morello) and CHERI-RISC-V instruction sets. To reason about CHERI instructions, it uses a symbolic-execution engine that tracks the architectural state: which registers hold capabilities and how each instruction transforms the base, bounds, permissions, object type, sealed bit, and tag. Executing a candidate sequence symbolically yields the formulas Φ_S and Φ_R that the verifier compares (§4.2), letting the optimizer exploit target-specific substitutions that are invisible at the IR level, including those involving raw machine details, such as instruction mode bits or specialized load/store forms. The assembly optimizer emits an optimized binary by patching in the improved instruction sequences, effectively acting as a CHERI-aware binary rewriter that complements the IR-level optimization.

Cost model. The security-aware cost model described in §3.4 operates at two levels. During the offline search, candidates are statically scored using an instruction-count baseline, per-opcode latency weights derived from the instruction text, and security-related terms for bounds width,

permissions, and tag manipulation. The latency weights are currently estimated rather than vendor-provided cycle counts, but they are configurable. This static model serves as a proxy for ranking candidate transformations and may not accurately predict wall-clock execution time. An optional dynamic model can be used to re-rank the leading candidates chosen by the static model using execution time measurements. To this end, CapOpt compiles and profiles each candidate over multiple runs on an ARM Morello board, computes the median execution time, and combines it with the static latency estimate using a configurable trust weight. Dynamic re-ranking affects only the final ranking and does not change candidate generation. Regardless of whether dynamic re-ranking is enabled, the final rewrite outcome is empirically validated against the original code and retained only if it is measurably faster (§6). The end-to-end results in §6 use the same measurement harness. On CHERI-RISC-V, the dynamic metric is retired instruction count because QEMU does not model pipeline timing. On Morello, performance is measured using hardware cycle counts.

4.2 Verification Engine

For a source sequence S and a candidate rewrite R, CapOpt's symbolic-execution engine generates two formulas, Φ_S and Φ_R, which encode their behavior over symbolic variables representing the initial machine state (registers, capability fields, and memory). The verifier then asks whether there exists an initial state σ for which the final state of R violates the CHERI equivalence conditions with respect to the final state of S, i.e., a counterexample to $R \equiv_{CHERI} S$. If Z3 reports that the query is unsatisfiable, no such state exists and R is formally proven to be equivalent to S; otherwise the returned model exhibits an input in which S and R differ, and R is rejected.

Encoding full machine semantics for SMT solving is challenging, especially for memory and the most intricate instructions, so we combine several modeling techniques to keep verification tractable without sacrificing soundness. In the assembly-level verifier, integer memory is modeled as per-location symbolic cells: a load reads the cell for its (base register, offset) location, such as forwarding a value stored earlier in the window, or a fresh symbol for a location the window has not written, and a store updates it. Capability-backed memory and other opaque architectural state (for example, the contents read back through a capability, special capability registers, and the link values produced by sealed invocation and indirect control flow) are instead modeled with *uninterpreted functions* keyed by the accessing authority, so distinct capabilities yield distinct symbols. This abstraction sidesteps a state that has no tractable bit-level definition while still preserving the provenance distinctions on which the equivalence check depends.

This memory model avoids alias reasoning *by construction* rather than by enumerating alias cases. A naive encoding must consider every possible pointer-aliasing relationship between accesses in the two sequences, leading to quadratic growth in the number of constraints. CapOpt instead requires the rewrite to preserve the source's *ordered memory-event trace*, i.e., the same loads and stores, to the same locations, in the same order (the observability rule of §3.1), under which both sequences see the identical memory view and no cross-access alias case split ever needs to be generated. Rewrites are thus confined to pure compute slices between memory events, and the provenance analysis of §3.3 independently prunes candidates that would combine unrelated capability lines before any solver query is issued. Together, these keep CapOpt tractable on memory-intensive sequences that a pairwise alias-aware encoding could not handle.

IR-level refinement check. As established in §3.1, IR-level correctness is the capability-equivalence relation with its functional condition read as Alive2-style *refinement*, because LLVM IR values carry `poison` and `undef` semantics. CapOpt realizes this obligation directly in Z3: for each output, it asserts $\neg poison_S \Rightarrow (\neg poison_R \wedge eq)$, where eq is the equality of the values for ordinary results and the metadata-restrictions plus the tags' preservation conditions for the capability-typed results

(the two sides paired through the data-flow mapping $\mathcal{M}$). Ordinary IR semantics follow Alive2's LLVM model, while the `@llvm.cheri.cap.*` intrinsic semantics are derived from the CHERI-LLVM backend and the CHERI-Sail specification.

Two-sided validation of the encoding. Because this encoding is hand-written, CapOpt validates it against two independent oracles. *Against Alive2:* on the capability-free fragment of the IR (integer arithmetic, casts, comparisons, `select`, `freeze`, in-bounds GEPs), CapOpt differential-tests its verdicts against `alive-tv`; the two checkers must agree, and any disagreement is logged as a soundness incident for inspection. Vanilla Alive2 has no semantics for the `@llvm.cheri.cap.*` intrinsics or for capability-typed (`addrspace(200)`) values, so this cross-check necessarily excludes capability operations; it is best-effort — off by default, degrading gracefully when `alive-tv` is absent — and, in its default configuration, never overrides CapOpt's verdict. *Against Sail:* the CHERI intrinsic encodings are cross-checked against the official CHERI Sail specification [15] using the Isla symbolic-execution engine, whose `isla-footprint` mode extracts each instruction's capability transition directly from the Sail model; we then prove that transition equivalent, field by field, to CapOpt's Z3 transition. Of the 82 CHERI-RISC-V instructions in our catalog, 48 are cataloged as bit-exact equivalents to Sail (47 of these carry a completed mechanical proof), 7 are documented hand-model gaps, and 27 remain unmodeled (capability atomics, DDC-relative loads and stores, and assembler-only `cjal`). Both validations are one-time, offline soundness assurances; neither Alive2 nor Sail/Isla is invoked inside the optimization loop.

4.3 PGSS in Practice: Guided Optimization

CapOpt applies the PGSS methodology in two complementary modes to maximize its effectiveness:

Learning. In an initial *one-time* phase (ahead of optimizing any particular program), CapOpt runs the PGSS algorithm in a special *learning mode*. The goal here is to systematically **pre-explore a library of core instruction rewrites** and their verified semantics. Essentially, the tool uses PGSS to synthesize and prove correct optimized sequences for many primitive CHERI operations, then saves these results for reuse. For example, CapOpt might synthesize a short instruction sequence that implements the semantics of obtaining a capability's length (`CGetLen`) using more fundamental instructions, or an optimal sequence for adjusting a capability's bounds under certain conditions. Each such sequence is verified and then recorded in an internal library of known-correct patterns. This offline process is computationally intensive (involving deep synthesis and solver calls), but it only needs to be done once. The resulting library of rewritten *recipes* can be persistently stored (e.g., in an external database or cache) and reused across optimization runs.

Optimization. When CapOpt is later invoked to optimize a user program, it takes advantage of the learned rules to **guide the synthesis** engine. Essentially, the accumulated knowledge from learning biases the search toward known-good transformations. If the code being optimized contains a recognizable CHERI idiom or sub-pattern that was seen in the library, CapOpt will preferentially apply the corresponding rewrite rather than trying to rediscover it from scratch. For instance, suppose that a portion of the input code is doing a bound-checked pointer increment (perhaps via an elaborate sequence of instructions). The synthesizer thus avoids exploring a long chain of low-level integer arithmetic transformations (which might involve casting a capability to an integer, adding an offset, and reconstituting a capability — a path likely to hit dead-ends due to tag invalidation). Instead, it instantly recognizes the higher-level operation and substitutes the previously verified rewrite (e.g., a direct `CIncOffset`). This makes the online search more efficient.

In summary, PGSS employs a two-step method: first, a learning phase equips the system with strategies for effective capability manipulations; second, a guided optimization phase uses this knowledge to accelerate code optimization. This ensures CapOpt does not start from scratch but

rather builds on previously established transformations, reducing later search costs. Our tests show that this approach significantly speeds up superoptimization and allows CapOpt to manage complex programs efficiently while ensuring correctness.

4.4 Other Optimizations

CapOpt performs *syntactic canonicalization*: instruction forms are normalized, for example, by imposing a canonical operand order on commutative operations so that `x+y` and `y+x` share a single representation. This reduces the search space by collapsing equivalent variants. CapOpt also memorizes verification results in a normalization-keyed cache, allowing previously checked candidates and subproblems to reuse stored SMT outcomes rather than re-invoking the solver. Both techniques are particularly effective for the recurring capability idioms common in CHERI code.

5 Experimental Setup

We evaluate CapOpt on two distinct CHERI-enabled platforms to answer four questions: (i) how much performance can CapOpt recover relative to CHERI-LLVM while preserving CHERI capability security properties; (ii) whether combining IR- and assembly-level optimization provides greater benefits than either level alone; (iii) how effectively PGSS reduces the synthesis search space; and (iv) whether the optimization benefits generalize across CHERI-enabled ISAs.

5.1 Hardware Platforms

We evaluated CapOpt on two CHERI-enabled systems.

ARM Morello board: This is a CHERI prototype system-on-chip that integrates 128-bit CHERI capabilities into a quad-core, 2.5 GHz ARMv8.2-A Neoverse N1 processor with out-of-order superscalar cores. Each core has 64 KB L1 instruction and data caches, both four-way set-associative with 64-byte cache lines. Each core also has an eight-way set-associative L2 cache of 1 MB, with 64-byte cache lines. The four cores share a 1 MB last-level cache.

CHERI-RISC-V simulator: We use QEMU-CHERI [44], an extension of QEMU [43] that supports CHERI-aware software on the CHERI-RISC-V and CHERI-MIPS architectures. It emulates systems that enforce CHERI capability integrity and access control. We use its CHERI-RISC-V configuration to evaluate the cross-ISA generality of CapOpt. The simulator runs on an Ubuntu 24.04.2 LTS host with a 128-core Intel Xeon Gold 6438Y+ processor operating at up to 4.0 GHz and 512 GB of DDR5 memory at 4800 MT/s.

5.2 Baseline Compiler and CapOpt Integration

We compile all code using the CHERI-LLVM compiler toolchain [54], version 17.0.0 at commit `9e82d2969a6`. For both ARM Morello and CHERI-RISC-V, CapOpt's search itself starts from the compiler's `-O0` output, whose assembly and IR serve as input to the superoptimizer; §6.4 evaluates the alternative of starting from `-O3` code. The resulting binaries are then measured against the CHERI-LLVM `-O3` baseline. Unless stated otherwise, the compiler produces pure-capability executables in which pointers are represented as CHERI capabilities.

CapOpt operates at two levels. Its IR-level optimizer processes LLVM IR before normal target code generation, and its assembly-level optimizer processes the resulting CHERI assembly before final assembly and linking.

5.3 Benchmarks

Our evaluation targets benchmarks sensitive to CHERI-related performance overheads, with an emphasis on low-level, pointer-intensive computation.

Hacker's Delight: This benchmark suite contains bit-manipulation and arithmetic microkernels drawn from prior work [17, 56]. These loop-free kernels test the optimizer's ability to generate efficient instruction sequences for integer operations. They provide a focused evaluation of the performance potential of CapOpt on CHERI code.

SPEC CPU2017 subset: We use 12 C/C++ benchmarks from SPEC CPU2017 [8]: `510.parest_r`, `519.lbm_r`, `520.omnetpp_r`, `523.xalancbmk_r`, `531.deepsjeng_r`, `541.leela_r`, `544.nab_r`, `557.xz_r`, `620.omnetpp_s`, `623.xalancbmk_s`, `631.deepsjeng_s`, and `641.leela_s`. These benchmarks make extensive use of pointers and dynamic memory and represent realistic workloads in which capability operations and checks may affect performance.

LLaMA.cpp inference engine: LLaMA.cpp [26] is a C/C++ framework for inference of large language models. It uses large memory allocations and pointer-rich data structures in a performance-critical setting, making it a useful case study to evaluate CapOpt in a practical codebase.

5.4 Evaluation Metrics and Methodology

We collect multiple metrics to quantify the performance benefits and overhead of CapOpt.

Performance improvement: Our primary metric is the performance improvement of code optimized by CapOpt relative to code produced by the baseline CHERI-LLVM compiler. For a benchmark b, we report the percentage reduction in execution time as

$$\Delta(b) = \frac{T(b, \text{baseline}) - T(b, \text{CapOpt})}{T(b, \text{baseline})} \times 100\%.$$

A higher value indicates a greater performance improvement. On ARM Morello, performance is measured directly on hardware. On CHERI-RISC-V, we use retired instruction count because QEMU does not accurately model pipeline timing. The CHERI-RISC-V results are therefore used to assess whether the observed optimization benefits generalize across ISAs rather than to predict hardware execution time.

Optimization time: We report the wall-clock time spent by CapOpt on superoptimization as a measure of compilation overhead. Depending on the experiment, this time is reported per function or per program. Although CapOpt improves runtime performance, it introduces additional compilation cost. We quantify this cost to assess its suitability for offline and ahead-of-time optimization.

Methodology: For each benchmark, we produce a baseline version compiled using CHERI-LLVM `-O3` and an optimized version produced by running CapOpt's two-level pipeline (from the `-O0` search input) with the same toolchain. We execute the ARM binaries on Morello hardware. We profile each binary ten times and report the median measurement. We find that there is little variance across runs, $\sim$ 0.69%. All improvements are calculated relative to the corresponding baseline on the same platform and under the same measurement conditions.

6 Experimental Results

6.1 Performance Improvements

CapOpt consistently delivers measurable speedups over CHERI-LLVM. As summarized in Table 2, every benchmark category we tested sees improved performance with CapOpt-optimized code. In the 25 Hacker's Delight arithmetic programs, CapOpt found instruction sequences that matched or exceeded the performance of the baseline code (CHERI-LLVM -O3), essentially improving the performance of CHERI with a 2.9-3.8% speedup. For the pointer-heavy SPEC CPU2017 benchmarks, CapOpt achieved a geometric-mean speedup of approximately 2.5% on Morello, with per-benchmark improvements of up to 4.1%. It removes several redundant bound-checks and address manipulations, cutting runtime compared to the baseline (CHERI-LLVM). CapOpt increased LLaMA's inference

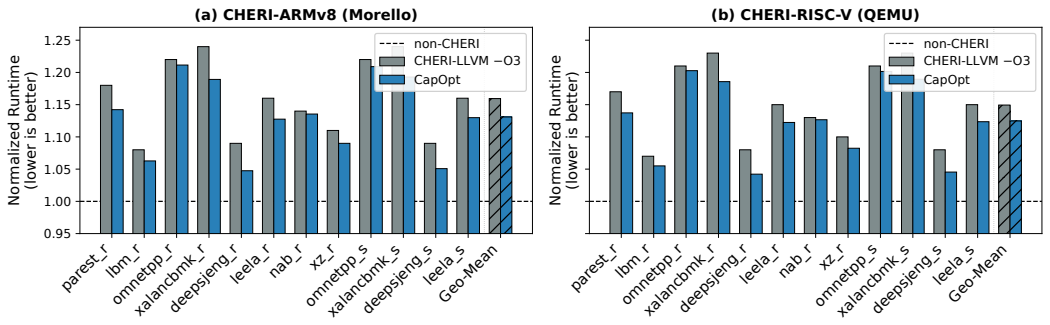


Fig. 2. SPEC CPU2017 normalized execution cost on (a) Morello, measured in cycles, and (b) CHERI-RISC-V, measured in retired instructions. All runtimes are normalized to the non-CHERI baseline compiled under −O3, shown by the dashed line at 1.0; lower is better. Gray bars show CHERI-LLVM −O3, which incurs a capability-enforcement overhead over non-CHERI, while blue bars show CapOpt, which reduces the residual overhead.

speed by up to 0.4% compared to the baseline (CHERI-LLVM -O3). This is significant for an already optimized application like LLaMA.cpp, as CapOpt improved pointer arithmetic and capability operations without changing the model's results.

CapOpt is designed to avoid performance degradation in the input program. The static cost model is used only to rank candidates and is not assumed to predict wall-clock time (§4.1). Instead, CapOpt profiles each cost-model-selected rewrite against the original code on the target platform and retains the original whenever the rewrite is not measurably faster. Consequently, when no profitable rewrite is found, performance remains at least at the baseline level (e.g., CHERI-LLVM -O3); the no-regression property comes from this empirical validation step, not from the cost model.

Figure 2 breaks down the per-benchmark SPEC CPU2017 results for both platforms. The pointer-intensive C++ workloads benefit the most: on ARM, 523.xalancbmk_r achieves a 4.1% speedup and the deepsjeng variants gain 3.6-3.9%, because these programs exercise extensive pointer dereferencing and dynamic memory management, precisely where CHERI imposes the most overhead and where CapOpt's capability-aware optimizations eliminate the most redundancy. The pointer-heavy 510.parest_r (~3.2%) and the compression utility 557.xz_r (~1.8%) show moderate gains, reflecting their lighter pointer workloads. The compute-bound 519.lbm (1.6% on ARM) and the near-parity omnetpp and nab workloads mark the lower end, where the validation gate retains the original code whenever no rewrite is measurably faster. RISC-V results follow the same trend at slightly lower magnitudes, confirming that the benefits generalize across ISAs. This shows that CapOpt recovers much of the performance lost in CHERI's capability enforcement, making hardware memory safety substantially cheaper in practice.

6.2 Two-Level vs. Single-Level Optimization (Ablation Study)

To validate the benefits of CapOpt's two-level optimization strategy, we compare three configurations in an ablation study: (1) **LLVM IR-only** synthesis, (2) **Assembly-only** synthesis, and (3) the full two-level CapOpt pipeline combining both. As shown in Table 2, the two-level approach delivers the highest performance improvements on the SPEC CPU2017 and LLaMA.cpp workloads on both CHERI-ARM and CHERI-RISC-V platforms; on the small Hacker's Delight kernels the margins between configurations are narrow, and assembly-only synthesis is slightly ahead on RISC-V. For the pointer-intensive workloads, this confirms that combining IR-level and assembly-level

Table 2. LLVM IR-level vs. Assembly-level Superoptimization: execution-cost improvement (%) for each configuration. Higher is better.

	LLVM IR only (%)		Assembly only (%)		CapOpt (%)	
Benchmark	ARM	RISC-V	ARM	RISC-V	ARM	RISC-V
Hacker's Delight						
Geometric mean (25 prog.)	0.13	0.88	0.00	1.75	0.15	1.35
SPEC CPU2017						
510.parest_r	1.10	1.00	2.60	2.30	3.20	2.80
519.lbm_r	1.40	1.20	1.60	1.40	1.60	1.40
520.omnetpp_r	0.10	0.10	0.50	0.40	0.70	0.60
523.xalancbmk_r	1.40	1.20	3.50	3.00	4.10	3.60
531.deepsjeng_r	1.90	1.60	3.80	3.40	3.90	3.50
541.leela_r	0.90	0.80	2.30	2.00	2.80	2.40
544.nab_r	0.10	0.10	0.30	0.20	0.40	0.30
557.xz_r	0.50	0.50	1.40	1.20	1.80	1.60
620.omnetpp_s	0.20	0.20	0.70	0.60	0.90	0.70
623.xalancbmk_s	1.20	1.00	3.20	2.80	3.80	3.30
631.deepsjeng_s	1.60	1.40	3.30	2.90	3.60	3.20
641.leela_s	0.90	0.80	2.10	1.80	2.60	2.30
Geometric mean	**0.94**	**0.83**	**2.12**	**1.84**	**2.46**	**2.15**
LLaMA.cpp						
Inference engine	0.30	0.20	0.90	0.70	1.20	1.00

optimization yields superior results. Each level contributes unique, complementary optimizations: the IR-level stage captures high-level algebraic simplifications that are invisible after lowering, whereas the assembly-level stage exploits low-level, ISA-specific idioms not exposed at the IR level. By leveraging both types of transformations, CapOpt recovers consistently more performance than either approach alone.

Specifically, in the Hacker's Delight bit-manipulation benchmarks the gains are modest on both platforms: on ARM, CapOpt recovers a 0.15% geometric-mean gain, compared to 0.00% from assembly-only and 0.13% from IR-only; on RISC-V, CapOpt achieves 1.35%, while assembly-only alone reaches 1.75% and IR-only 0.88%, indicating that for these small loop-free kernels the assembly-level pass captures most of the available improvement. SPEC CPU2017 shows a clearer gap – CapOpt yields a 2.46% geometric-mean speedup on ARM, surpassing 2.12% from assembly-only and 0.94% from IR-only optimizations; on RISC-V, CapOpt improves performance by 2.15% versus 1.84% (assembly-only) and 0.83% (IR-only). Finally, CapOpt improves LLaMA.cpp's performance by 1.20% on ARM compared to 0.90% from assembly-only optimization and only 0.30% from IR-level optimization. In RISC-V, CapOpt similarly achieves 1.00%, versus 0.70% and 0.20% from assembly-only and IR-only, respectively. Across the SPEC CPU2017 and LLaMA.cpp benchmarks, two-level synthesis outperforms single-level alternatives, confirming that each level targets complementary inefficiencies. The IR-level pass eliminates high-level redundancies and algebraic overhead, while the assembly-level pass discovers target-specific instruction substitutions (for CHERI instructions and other ISA-level idioms) that the IR pass alone cannot realize. In combination, these effects enable CapOpt's full pipeline to recover more performance than either level achieves in isolation, validating the synergy of two-level optimization in CHERI systems.

Table 3. Representative categories among the 782 rewritten instances.

Class (ratio)	Example (before → after)	Effect
Capability-walk folding (32.0%)	`a = CGetAddr c; a' = a + 16; c' = CSetAddr(c, a')` → `c' = CIncOffset(c, 16)`	multi-instruction address walk folded to one.
Permission/bounds restriction (21.9%)	`p = CGetPerms c; p' = p & mask; c' = CSetPerms(c, p')` → `c' = CAndPerm(c, mask)`	fused into one monotonic permission-restriction operation.
Arithmetic normalization (4.9%)	`x - (-5)` → `x + 5`; `x + (-1)` → `x - 1` (inside `x & (x-1)`)	negative immediates are normalized to canonical positive forms;
Boolean/compare simplification (2.4%)	`icmp eq i1 %c, true` → `icmp eq i1 %c, 1`	boolean literal canonicalized to its numeric form.
Generic enumerative / other (38.9%)	`icmp eq %x, %x` → `true`; `icmp eq i1 %c, true; ret` → `ret %c`	SMT-proven identities with no hand-written pattern.

6.3 Search-Space Pruning and Diversity of Discovered Rewrites

How much is pruned before the solver? The stratified search discipline and the provenance-guided alias pruning of §3.3 are what make CapOpt's search tractable, and their effect is measurable: across our workloads, stratification and provenance-guided pruning together filter out ~91.7% of candidate rewrites before any SMT query is issued. The observed filtering mainly comes from: (i) canonicalization-based deduplication and (ii) provenance-guided alias pruning, with the stratified discipline bounding enumeration in the first place. Because SMT solving dominates the synthesis time (§6.5), discarding roughly eleven of every twelve candidates ahead of the solver is the difference between the practical optimization times reported below and an intractable search.

What does CapOpt actually find? To characterize the diversity of the optimizations discovered, we classified the 782 rewritten instances (Table 3) that CapOpt accepted in our optimization runs.

6.4 Comparison with Conventional Superoptimizers

Because existing superoptimizers, including STOKE [46] and Souper [45], do not model CHERI capability metadata, their conventional correctness models cannot directly establish the validity of rewrites involving CHERI capabilities. We therefore compare CapOpt against prior superoptimizers in a setting where both apply: conventional non-CHERI RISC-V. In this configuration, CapOpt and Souper achieve comparable speedups across Hacker's Delight kernels, differing by at most ±2%. Souper (commit `963d4df4`) runs with its default parameters (candidate harvesting, Z3 integration, and solver timeouts as shipped). Each candidate is fully resolved by the solver, which requires between 58 and 158 SMT queries per kernel. This result suggests that CapOpt preserves the optimization potential of generic enumerative synthesis, while adding capability-aware reasoning required for correctness on CHERI.

Starting from `-O3` compiled code: Following STOKE [46], CapOpt's pipeline starts the search from the compiler `-O0` output (§5). To assess the alternative, we also ran the CapOpt assembly phase directly on CHERI-LLVM `-O3` output for the Hacker's Delight suite on Morello (Table 4). CapOpt finds no provably improving rewrite across the 25 kernels when starting from `-O3` output; the residual per-kernel deltas in Table 4 (at most ±0.36%) reflect run-to-run measurement variation on binaries that are identical or near-identical to the baseline.

Table 4. The CapOpt assembly phase applied to CHERI-LLVM -O3 output for Hacker's Delight

Kernels p01–p13				Kernels p14–p25			
Kernel	-O3 (cycles)	+CapOpt (cycles)	Δ (%)	Kernel	-O3 (cycles)	+CapOpt (cycles)	Δ (%)
p01	8,509,342	8,494,868	+0.17	p14	8,585,046	8,557,776	+0.32
p02	8,452,498	8,452,498	0.00	p15	8,574,854	8,574,444	+0.00
p03	8,495,676	8,471,590	+0.28	p16	8,596,129	8,565,367	+0.36
p04	8,502,376	8,491,363	+0.13	p17	8,451,190	8,451,190	0.00
p05	8,451,249	8,451,249	0.00	p18	8,573,998	8,573,998	0.00
p06	8,447,148	8,447,148	0.00	p19	8,690,103	8,684,994	+0.06
p07	8,524,089	8,500,706	+0.27	p20	8,484,718	8,484,718	0.00
p08	8,461,771	8,461,771	0.00	p21	8,720,142	8,720,142	0.00
p09	8,570,906	8,570,906	0.00	p22	8,501,058	8,489,300	+0.14
p10	8,624,806	8,616,802	+0.09	p23	8,433,198	8,433,198	0.00
p11	8,592,650	8,570,126	+0.26	p24	8,475,317	8,475,317	0.00
p12	8,583,462	8,552,956	+0.36	p25	8,592,550	8,577,260	+0.18
p13	8,473,165	8,461,028	+0.14				

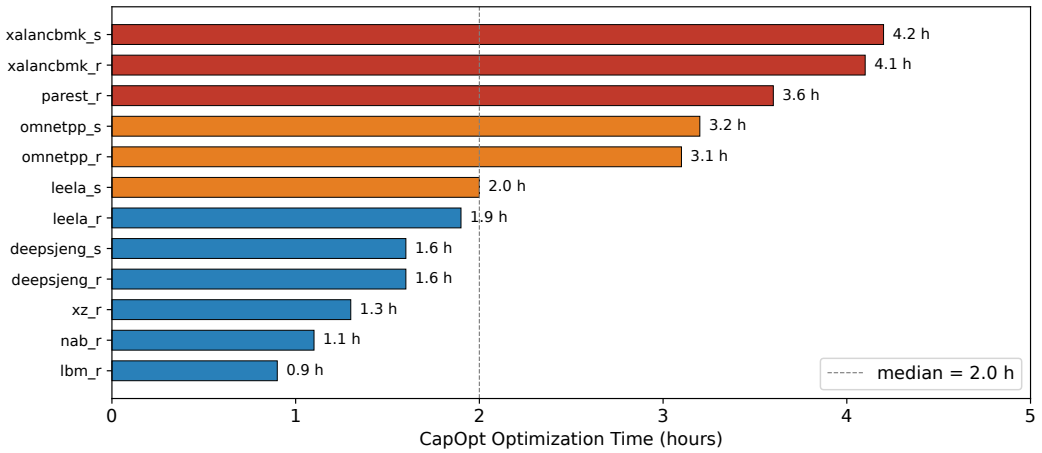


Fig. 3. Per-benchmark CapOpt optimization time for the 12 SPEC CPU2017 workloads (full two-level IR + assembly pipeline). Bars are sorted by decreasing time: blue for benchmarks completing within about 2 hours, orange for intermediate times of roughly 3 hours, and red for the slowest programs at 3.5 hours or more.

6.5 Compilation Overhead

CapOpt's optimization cost divides into two parts: (i) constructing and verifying the pattern library, a one-off cost per ISA and CHERI-LLVM version, reused unchanged across programs; and (ii) per-program superoptimization, paid once per program ahead of time and amortized over all executions of the optimized binary. The measurements below cover only the second cost.

For Hacker's Delight programs, CapOpt optimized each small bit-manipulation kernel within around ten minutes. These microbenchmarks have straight-line, arithmetic-heavy code, yielding a limited search space that the solver can explore quickly.

For larger real-world code from SPEC CPU2017, the optimization of CapOpt took a few hours per program. Figure 3 shows the per-benchmark optimization time for the 12 SPEC CPU2017 workloads. The largest programs, such as 623.xalancbmk_s (4.2 hours) and 523.xalancbmk_r (4.1 hours), incur

Table 5. Comparison with prior superoptimizers.

Framework	Target ISA(s)	Optimization Level	Search Strategy	Correctness Condition
CapOpt	CHERI ARM CHERI RISC-V	LLVM IR & Assembly	Synthesis (PGSS)	Capability-aware
Souper [45]	LLVM-supported (x86, ARM)	LLVM IR	Synthesis final(CEGIS)	Functional (UB-aware)
STOKE [46], CSTOKE [50]	x86-64	Assembly	Stochastic (MCMC)	Functional (conditional)
Bansal & Aiken [5]	x86	Assembly	Enumerative (brute force)	Functional (via testing)
Denali [21]	Alpha EV6	Assembly	Goal-directed (axiomatic)	Functional
GreenThumb [40]	ARMv7, GA	LLVM IR/ARM/GA	Stochastic (MCMC) with learned distribution	Functional (symbolic)

the highest optimization time, reflecting their large codebases and complex pointer-traversal patterns. The pointer-heavy omnetpp and parest benchmarks require 3.1-3.6 hours. In contrast, smaller benchmarks such as 519.lbm_r (52 minutes) and 544.nab_r (65 minutes) complete optimization in about an hour. The median optimization time in all 12 benchmarks is approximately 2 hours. The increased code size and more complex control flow (with numerous pointer operations and memory accesses) naturally lead to longer solver runtimes. We found that most of the solver time is spent in the deepest parts of the search (assembly-level synthesis), but the overall compile-time overhead for SPEC benchmarks remained in hours, not days.

For LLaMA.cpp, the full two-level pipeline required approximately 10.5 hours on our 128-core host, reflecting the application's large codebase and pervasive pointer-rich data structures. Although this is a substantial one-time cost, it is amortized over all subsequent executions of the optimized binary and is well within the range acceptable for ahead-of-time compilation scenarios such as firmware deployment or release builds.

7 Related Work

CapOpt builds on research in superoptimization [21, 30, 40, 45–47, 50], secure compilation [1, 39, 55], and formal methods for ISA specification, verified compilation, and translation validation [7, 15, 25, 29, 33, 42]. Its main distinction is a new program-equivalence criterion for a capability-based ISA. This criterion extends correctness to include hardware-enforced security metadata, a requirement not addressed by prior superoptimizers.

7.1 Superoptimization and Program Equivalence

Classic superoptimization, from Massalin's foundational exhaustive search [30] to goal-directed methods such as Denali [21] and enumerative methods such as those of Bansal and Aiken [5, 6], established functional equivalence as the main correctness criterion. Under this model, two instruction sequences are equivalent if they produce the same observable register and memory states.

Modern scalable frameworks follow this model. STOKE [46, 47] uses stochastic MCMC search [2] for x86-64 assembly, while CSTOKE [50] extends this approach to conditional correctness. Souper [45] uses SMT-backed counterexample-guided inductive synthesis (CEGIS) [52] for LLVM IR. GreenThumb [40] supports superoptimization across multiple ISAs through bidirectional search [41] and a learned proposal distribution, allowing it to remove infeasible candidates more efficiently than naive exhaustive search. However, these equivalence models are capability-unaware: they represent pointers as untyped bitvectors without semantic fields for bounds, permissions, or provenance. CHERI fundamentally invalidates this assumption.

For example, a `pointer-to-integer-to-pointer` cast may preserve the numerical address and therefore appear functionally correct on a conventional ISA. On CHERI, however, the same

transformation can clear the capability tag and break provenance. CapOpt addresses this semantic gap through *capability-aware equivalence*, which requires preservation of (1) functional behavior for non-capability state, (2) provenance validity, including the capability tag, and (3) metadata monotonicity. These additional requirements motivate Provenance-Guided Stratified Synthesis (PGSS), which structures the larger search space created by metadata-manipulating instructions.

7.2 Rewrite-Rule Synthesis and Equality Saturation

Nötzli et al. [36] use syntax-guided enumeration to synthesize rewrite rules, while Pal et al. [38] use equality saturation to explore structured rewrite spaces. These approaches are related to CapOpt's offline learning phase and PGSS. However, CapOpt additionally verifies CHERI provenance and metadata constraints. Learning-based and LLM-based methods [9, 51] could also serve as alternative proposal mechanisms, with all generated candidates checked by CapOpt's SMT-based verifier.

7.3 Software-Based Memory-Safety Mechanisms

Software-based memory-safety mechanisms for C and C++ are well established. SoftBound [32] provides spatial safety by associating each pointer with separate base and bounds metadata. This protection has a substantial cost, with an average runtime overhead of 67% in its full-checking mode. AddressSanitizer (ASan) [48] uses shadow memory to mark redzones around allocated objects. Instrumented memory accesses check the shadow state, and an access to a marked byte reports an error, such as an out-of-bounds access or a use-after-free error. ASan can introduce considerable runtime and memory overhead, making it primarily suitable for testing and debugging rather than production deployment. SAFECode [14] uses static analysis to prove memory accesses safe and removes the corresponding runtime checks when possible. In contrast to these software approaches, CHERI moves memory-safety enforcement into the hardware. Each capability operation and memory access is checked against its bounds and permissions. CapOpt exploits these explicit capability semantics to remove redundant operations while preserving hardware-enforced guarantees.

7.4 Hardware-Assisted Memory-Safety Extensions

Several architectural extensions have been proposed to improve memory safety. Intel MPX [37] added bounds registers and instructions for checking pointer ranges at runtime. It was later deprecated because of its high overhead, complex compiler integration, and poor interaction with compiler optimization and multithreaded programs. Intel Control-flow Enforcement Technology (CET) [18, 19] provides hardware support for forward- and backward-edge control-flow integrity. CET operates independently of data and memory safety and does not change pointer representation, aliasing behavior, or the compiler's memory model. It therefore requires fewer changes to traditional optimization strategies. ARM Memory Tagging Extension (MTE) [3] uses small tags associated with pointers and memory regions to detect some spatial and temporal memory-safety violations. CHERI differs from MPX, CET, and MTE because it changes the semantics and representation of pointers themselves. A CHERI capability contains bounds, permissions, and provenance metadata that are directly enforced by hardware. Optimizing CHERI code therefore requires a capability-aware equivalence model such as the one introduced by CapOpt.

7.5 Formal Verification and Secure Compilation

CapOpt builds on the formal modeling of the CHERI ISA, which has been specified and analyzed using formalisms such as Sail [15] and Isabelle/HOL [34]. These models support proofs of fundamental CHERI properties, including capability monotonicity. Our goal differs from that of verified compilers such as CompCert [25], which provides an end-to-end, machine-checked proof of semantic preservation for the complete compilation chain. Our approach also differs from formal

extensions such as ArchMage and CompCertCast [23], which support and verify the compilation of C programs containing unsafe `intptr_t` casts. Rather than verifying an entire compiler, CapOpt uses an SMT solver to validate each proposed rewrite individually.

Overall, CapOpt specifically addresses superoptimization for capability-based hardware. It combines program synthesis with a capability-aware equivalence model to discover CHERI-specific optimizations that are unavailable to conventional superoptimizers and general-purpose compilers. Its security-aware cost model further allows performance improvements to be considered alongside tighter bounds and reduced permissions, without weakening CHERI's core security guarantees.

8 Discussion

Our evaluation demonstrates that CapOpt enhances the quality of code produced by the LLVM-based CHERI toolchain while maintaining CHERI's capability-safety guarantees. These improvements are achieved through capability-aware equivalence checking and a search strategy specifically designed to respect CHERI's provenance and authority model. More generally, our results indicate that capability-aware optimization can restore or improve performance without compromising hardware-enforced security properties.

Offline and per-program optimization costs. CapOpt incurs two distinct costs. First, the construction of the verified rewrite-pattern library constitutes a one-time, offline expense for a given ISA, after which the resulting patterns can be reused across arbitrary programs. The library must be regenerated only when there are changes to the underlying semantics, the compiler's intermediate representation, or the space of supported optimizations.

Second, processing each program involves region extraction, pattern instantiation, program-specific synthesis when needed, and SMT-based verification. As shown in §6, this pipeline is much more expensive than a conventional compiler pass, so CapOpt is mainly intended as an ahead-of-time or offline optimization framework. The cost is best amortized on long-running, widely deployed, or security-critical software.

Extending the optimization scope. The current prototype targets straight-line code within individual basic blocks. This captures many local CHERI inefficiencies, including repeated address rematerialization and redundant bounds or permission operations, while keeping synthesis and verification tractable. It does not directly handle general loops beyond small fixed-count unrollings. Extending the approach to loop-level transformations, for example, through invariant synthesis, equality saturation, or integration with a CHERI-aware loop optimizer, is therefore a next step.

Cross-basic-block synthesis is another extension. As discussed in §3, the current design avoids the additional complexity of control flow, memory dependence, and path-sensitive provenance. Souper-style cross-block synthesis [45], combined with capability-aware alias and memory-dependence analysis, could expose opportunities that are invisible within a single block. This would require a richer correctness model for paths, memory effects, and live capabilities.

Scaling candidate generation. Although Provenance-Guided Stratified Synthesis substantially reduces the number of candidates reaching the verifier, the search space still grows rapidly with region size. Scaling to larger functions may require region partitioning, prioritization of promising rewrite classes, repeated local passes, and parallel or portfolio-based search.

Because candidate generation is separated from capability-aware verification, alternative proposal mechanisms can be introduced without weakening correctness. CEGIS [52] could move more of the search into the solver, while equality saturation could retain multiple equivalent forms instead of committing immediately to a greedy rewrite. LLM-based synthesis [9, 51] could also propose candidates in large search spaces.

Profile-guided superoptimization. CapOpt currently relies mainly on static search and uses runtime measurements only to validate or re-rank selected rewrites (§4 and §6). Profile information could instead direct the search toward hot blocks, loops, and capability operations, while cold regions use only reusable patterns or are skipped. This would improve search-budget allocation without changing the capability-equivalence requirements.

In summary, CapOpt shows that capability-aware superoptimization can improve performance while preserving, and in some cases further restricting, capability authority. Broader control-flow reasoning, more scalable candidate generation, improved pattern reuse, and profile-guided search would increase its practicality for larger programs and other capability-enabled architectures.

9 Conclusion

We have presented CapOpt, the first superoptimization framework designed for CHERI hardware that treats capability semantics as a first-class concern. By combining a capability-aware equivalence condition with a two-level synthesis architecture and the PGSS search algorithm, CapOpt can automatically discover code sequences that are both provably correct and more efficient. Our evaluation on two CHERI-enabled ISAs shows that CapOpt improves the performance of CHERI-LLVM code by up to 4.1%. These results demonstrate that incorporating architectural semantics directly into the optimization process enables effective and secure program transformations.

Acknowledgments

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X037304/1 and EP/X037525/1.

Data Availability Statement

The artifact associated with this paper consists of the CapOpt superoptimization framework, which includes an LLVM IR-level optimization pass, an assembly-level binary rewriting tool, a Z3-based capability-aware verification engine, and the scripts used for benchmarking and evaluation. The implementation is primarily a Python package with a C++ backend and orchestration scripts, and it integrates with the CHERI-LLVM compiler toolchain without patching the compiler. The artifact also provides the benchmark setups and evaluation scripts required to reproduce the results presented in this paper for the Hacker's Delight microkernels, the SPEC CPU2017 benchmarks evaluated, and LLaMA.cpp, on both the ARM Morello platform and the CHERI-RISC-V (QEMU-CHERI) simulator. Note that replicating the ARM Morello experiments requires access to a Morello board, including possible remote access to ours, whereas the CHERI-RISC-V experiments can be reproduced using the publicly available QEMU-CHERI emulator. Running the SPEC CPU2017 benchmarks additionally requires a valid SPEC license.

References

[1] Martín Abadi. 1998. Protection in programming-language translations. In *International Colloquium on Automata, Languages, and Programming*. Springer, 868–883. doi:10.5555/646252.686313

[2] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. 2003. An introduction to MCMC for machine learning. *Machine learning* 50, 1 (2003), 5–43. doi:10.1023/A:1020281327116

[3] ARM. [n. d.]. Introduction to the Memory Tagging Extension. https://developer.arm.com/documentation/108035/0100/Introduction-to-the-Memory-Tagging-Extension.

[4] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. 2020. Sound garbage collection for C using pointer provenance. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. doi:10.1145/3428244

[5] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 394–403. doi:10.1145/1168917.1168906

[6] Sorav Bansal and Alex Aiken. 2008. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 177–192. doi:10.5555/1855741.1855754

[7] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert NM Watson, and Peter Sewell. 2022. Verified security for the Morello capability-enhanced prototype Arm architecture. (2022), 174–203. doi:10.1007/978-3-030-99336-8_7

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42. doi:10.1145/3185768.3185771

[9] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. (2023). arXiv:2309.07062 [cs.PL] https://arxiv.org/abs/2309.07062

[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35. doi:10.1145/75277.75280

[11] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490. doi:10.1145/115372.115320

[12] Brooks Davis, Robert NM Watson, Alexander Richardson, Peter G Neumann, Simon W Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, et al. 2019. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. (2019), 379–393. doi:10.1145/3297858.3304042

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[14] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing alias analysis for weakly typed languages. *ACM SIGPLAN Notices* 41, 6 (2006), 144–157. doi:10.1145/1133255.1133999

[15] Kathryn E Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. 2017. *The Sail instruction-set semantics specification language*. Technical Report. Technical report published by Cambridge University. https://www.cl.cam.ac.uk/~pes20/sail/manual.pdf

[16] Richard Grisenthwaite, Graeme Barnes, Robert NM Watson, Simon W Moore, Peter Sewell, and Jonathan Woodruff. 2023. The arm morello evaluation platform—validating cheri-based security in a high-performance system. *IEEE Micro* 43, 3 (2023), 50–57. doi:10.1109/MM.2023.3264676

[17] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73. doi:10.1145/1993498.1993506

[18] Intel. [n. d.]. Complex Shadow-Stack Updates (Intel Control-Flow Enforcement Technology. https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html.

[19] Intel. 2020. A Technical Look at Intel Control-Flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[20] Abhinav Jangda and Greta Yorsh. 2017. Unbounded superoptimization. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 78–88. doi:10.1145/3133850.3133856

[21] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (2002), 304–314. doi:10.1145/543552.512566

[22] Paul Ashley Karger. 1988. *Improving security and performance for capability systems*. Technical Report UCAM-CL-TR-149. University of Cambridge, Computer Laboratory. doi:10.48456/tr-149

[23] Yonghyun Kim, Minki Cho, Jaehyung Lee, Jinwoo Kim, Taeyoung Yoon, Youngju Song, and Chung-Kil Hur. 2025. Archmage and CompCertCast: End-to-End Verification Supporting Integer-Pointer Casting. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 1326–1354. doi:10.1145/3704881

[24] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: verifying real-world C idioms with integer-pointer casts. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–32. doi:10.1145/3498681

[25] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. doi:10.1007/s10817-009-9155-4

[26] LLaMA.cpp. 2025. LLM inference in C/C++. https://github.com/ggml-org/llama.cpp/.

[27] LLVM. 2026. Alias Analysis Infrastructure. https://llvm.org/docs/AliasAnalysis.html.

[28] LLVM. 2026. Auto-Vectorization. https://llvm.org/docs/Vectorizers.html.

[29] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79. doi:10.1145/3453483.3454030

[30] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126. doi:10.1145/36177.36194

[31] Kayvan Memarian, Victor BF Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert NM Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32. doi:10.1145/3290380

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258. doi:10.1145/1542476.1542504

[33] George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94. doi:10.1145/358438.349314

[34] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Berlin, Heidelberg. doi:10.1007/3-540-45949-9

[35] Robert M. Norton. 2016. *Hardware support for compartmentalisation*. Technical Report UCAM-CL-TR-887. University of Cambridge, Computer Laboratory. doi:10.48456/tr-887

[36] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-guided rewrite rule enumeration for SMT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, Cham, 279–297. doi:10.1007/978-3-030-24258-9_20

[37] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. (2018). doi:10.1145/3219617.3219662

[38] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality saturation theory exploration á la carte. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1034–1062. doi:10.1145/3622834

[39] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36. doi:10.1145/3280984

[40] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Greenthumb: Superoptimizer construction framework. In *Proceedings of the 25th International Conference on Compiler Construction*. 261–262. doi:10.1145/2892208.2892233

[41] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–310. doi:10.1145/2872362.2872387

[42] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–166. doi:10.1007/BFb0054170

[43] QEMU. 2025. a generic and open source machine & userspace emulator and virtualizer. https://github.com/qemu/qemu.

[44] QEMU-CHERI. 2025. a version of QEMU that can be used to run CHERI code for both RISC-V and MIPS. https://github.com/CTSRD-CHERI/qemu.

[45] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A synthesizing superoptimizer. (2017). arXiv:1711.04422 [cs.PL] https://doi.org/10.48550/arXiv.1711.04422

[46] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316. doi:10.1145/2451116.2451150

[47] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices* 49, 6 (2014), 53–64. doi:10.1145/2666356.2594302

[48] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318. doi:10.5555/2342821.2342849

[49] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 391–406. doi:10.1145/2509136.2509509

[50] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally correct superoptimization. *ACM SIGPLAN Notices* 50, 10 (2015), 147–162. doi:10.1145/2814270.2814278

[51] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. (2023). arXiv:2302.07867 [cs.SE] https://doi.org/10.48550/arXiv.2302.07867

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages*

and operating systems. 404–415. doi:10.1145/1168857.1168907

[53] Xiaoyang Sun, Jeremy Singer, and Zheng Wang. 2025. Sweet or sour CHERI: Performance characterization of the Arm Morello platform. (2025), 423–438. doi:10.1109/IISWC66894.2025.00042

[54] The CHERI LLVM Compiler Infrastructure. 2025. Fork of LLVM adding CHERI support. https://github.com/CTSRD-CHERI/llvm-project.

[55] Son Tuan Vu, Albert Cohen, Arnaud De Grandmaison, Christophe Guillon, and Karine Heydemann. 2021. Reconciling optimization with secure compilation. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. doi:10.1145/3485519

[56] Henry S Warren. 2002. *Hacker's delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. https://dl.acm.org/doi/10.5555/515297

[57] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, and Jonathan Woodruff. 2015. *Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide.* Technical Report UCAM-CL-TR-877. University of Cambridge, Computer Laboratory. doi:10.48456/tr-877

[58] Robert N. M. Watson, Jessica Clarke, Peter Sewell, Jonathan Woodruff, Simon W. Moore, Graeme Barnes, Richard Grisenthwaite, Kathryn Stacer, Silviu Baranga, and Alexander Richardson. 2023. *Early performance results from the prototype Morello microarchitecture.* Technical Report UCAM-CL-TR-986. University of Cambridge, Computer Laboratory. doi:10.48456/tr-986

[59] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9).* Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. doi:10.48456/tr-987

[60] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. 2020. *CHERI C/C++ Programming Guide.* Technical Report UCAM-CL-TR-947. University of Cambridge, Computer Laboratory. doi:10.48456/tr-947

[61] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W Filardo, A Theodore Markettos, et al. 2019. Cheri concentrate: Practical compressed capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. doi:10.1109/TC.2019.2914037

Received 2026-03-17; accepted 2026-08-06