EPSO: A Caching-Based Efficient Superoptimizer for BPF Bytecode

Qian Zhu, Yuxuan Liu, Ziyuan Zhu, Shangqing Liu and Lei Bu†
State Key Laboratory of Novel Software Techniques, Nanjing University, Nanjing, Jiangsu 210023, China
Email: {zhuqian, yuxuanliu, mg21330081}@smail.nju.edu.cn, {shangqingliu, bulei}@nju.edu.cn

***Abstract*—Extended Berkeley Packet Filter (eBPF) allows developers to extend Linux kernel functionality without modifying its source code. To ensure system safety, an in-kernel safety checker, the verifier, enforces strict safety constraints (e.g., a limited program size) on eBPF programs loaded into the kernel. These constraints, combined with eBPF's performance-critical use cases, make effective optimization essential. However, existing compilers (e.g., Clang) offer limited optimization support, and many semantics-preserving transformations are rejected by the verifier, which makes handcrafted optimization rule design both challenging and limited in effectiveness.**

Superoptimization overcomes the limitations of rule-based methods by automatically discovering optimal transformations, but its high computational cost limits scalability. To address this, we propose EPSO, a caching-based superoptimizer that discovers rewrite rules via offline superoptimization, and reuses them to achieve high-quality optimizations with minimal runtime overhead. We evaluate EPSO on benchmarks from the Linux kernel and several eBPF-based projects, including Cilium, Katran, hXDP, Sysdig, Tetragon, and Tracee. EPSO discovers 795 rewrite rules and achieves up to 68.87% (avg. 24.37%) reduction in program size compared to Clang's output, outperforming the state-of-the-art BPF optimizer K2 on all benchmarks and Merlin on 92.68% of them. Additionally, EPSO reduces program runtime by an average of 6.60%, improving throughput and lowering latency in network applications.

***Index Terms*—eBPF, Synthesis, Superoptimization**

I. Introduction

Extended Berkeley Packet Filter (eBPF) is a Linux kernel technology that enables safe and high-performance execution of user-defined programs within the kernel, allowing flexible extension of kernel functionality without modifying its source code or loading modules [1]. Originally developed for efficient packet filtering in Unix systems [2], eBPF has since been widely adopted in areas such as high-performance networking [3]–[5], observability [6]–[9], and security [10]–[12].

eBPF programs, typically written in high-level languages (e.g., C) and compiled to bytecode (e.g., via Clang), are loaded into the kernel after passing a static verifier. The verifier enforces strict safety constraints like size limits, guaranteed termination and aligned memory access. While essential for kernel security, the verifier also introduces challenges. Its size restrictions may exclude well-structured but lengthy programs, and conservative analysis can yield false negatives, rejecting safe programs and hindering efficient eBPF development.

The strict size limitations of eBPF programs, coupled with their increasing adoption in high-performance systems, make effective optimization essential. Optimizing eBPF programs not only allows larger programs to pass the verifier but also improves system throughput and reduces latency in performance-critical tasks. Although modern compilers like Clang provide basic optimizations, they mainly depend on manually crafted rules and consequently miss numerous optimization opportunities. The minimal bytecode size difference between -O2 and -O3 underscores this untapped potential. Moreover, the verifier's constraints impose additional challenges, making optimization rule design more labor-intensive and less effective.

Recent efforts in eBPF optimization can be broadly classified into three categories: synthesis-based, rule-based, and domain-specific approaches. Synthesis-based methods, exemplified by K2 [13], utilize superoptimization to identify more efficient rewrites. However, K2 depends on stochastic search, which may overlook valid optimizations. Moreover, its high synthesis latency significantly limits practicality for large-scale eBPF programs. Rule-based approaches [14] employ manually crafted optimization rules, enabling fast and lightweight transformations. Nevertheless, they demand considerable engineering effort to maintain verifier compliance and often fail to generalize beyond predefined rewrite rules, constraining optimization quality. Domain-specific techniques [15], [16] focus on specialized scenarios, such as substituting target program regions with optimized implementations or merging chains of pre-verified programs. While effective within their domains, these methods do not directly optimize eBPF bytecode and lack broad applicability to general-purpose workloads.

To address these challenges, we propose EPSO, a caching-based superoptimizer that efficiently delivers high-quality optimizations for BPF bytecode. Inspired by prior works [17], [18], EPSO offloads the expensive superoptimization process to an offline phase, where it enumeratively explores the program space to collect rewrite rules. These rules are cached and later applied to real-world eBPF code optimization through efficient rule matching, combining the high optimization quality of synthesis-based approaches with the low runtime overhead of rule-based methods. To enhance rule generality and applicability, EPSO introduces a tailored abstraction and slicing strategy: abstraction merges semantically similar rewrites into unified representations, while slicing extracts relevant instructions into self-contained units. Together, these techniques significantly enhance the generality of rewrite rules and improve optimization performance during the rule matching phase.

We evaluate EPSO on a diverse benchmark set from the Linux kernel [19] and several eBPF-based projects, including

†Corresponding author.

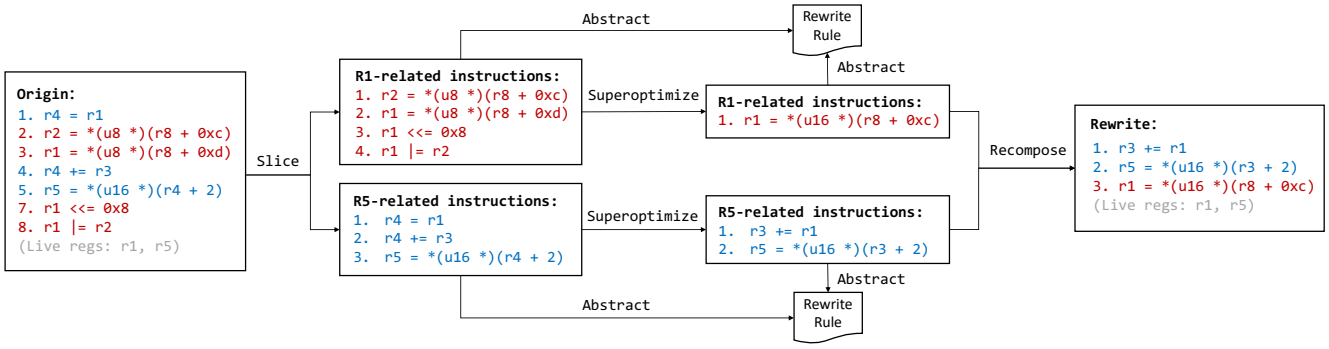


Fig. 1: A demonstrative example of EPSO's workflow.

Cilium [20], Katran [21], hXDP [22], Sysdig [23], Tetragon [24], and Tracee [25]. EPSO identifies and applies 795 rewrite rules, achieving up to 68.87% (average 24.37%) program size reduction compared to Clang's output, outperforming the state-of-the-art BPF optimizer K2 on all benchmarks and Merlin on 92.68% of them. For certain network-oriented programs, EPSO reduces runtime by an average of 6.60%, improving throughput and latency. Importantly, all optimized programs meet safety constraints and pass the BPF verifier.

In summary, our contributions are as follows:

- A novel optimization framework for BPF bytecode that achieves high-quality optimizations efficiently. By offloading superoptimization offline, we precompute a repository of high-quality rewrite rules for efficient reuse during real-world eBPF bytecode optimization. To further enhance rule generality, we introduce novel instruction abstraction and slicing strategies, significantly improving the reusability and applicability of the discovered rewrite rules.
- A superoptimization method that synthesizes compact, efficient, and safe rewrite rules. Unlike prior approaches that rely on stochastic search [13], our method integrates enumerative search with a set of carefully designed pruning strategies, enabling the efficient discovery of a broader set of optimizations during the rewrite rule generation phase.
- A comprehensive evaluation on a broad set of benchmarks from diverse sources and scales shows that our approach achieves up to 68.87% (avg. 24.37%) program size reduction, and up to 19.05% (avg. 6.60%) program efficiency improvement. Additionally, it reduces optimization overhead by 88.71% compared to K2 in synthesis-only mode and incurs negligible overhead when using rule matching.

II. Background and Motivation

A. Extended Berkeley Packet Filter (eBPF)

eBPF enhances kernel scalability by enabling sandboxed, event-driven programs to execute within the kernel, without requiring kernel source code modifications. These programs are typically written in high-level languages (e.g., C), compiled to bytecode, and loaded via the `bpf()` syscall [1].

Before execution, an in-kernel safety checker, the verifier, statically analyzes each program to enforce safety constraints, ensuring termination, bounded program size, and memory access safety, etc. Once an eBPF program passes the verifier, it can be attached to various system events, such as packet reception [4], [22], system calls [26], or application behaviors [27]. Upon triggering, the program executes in the kernel and communicates with user space through eBPF maps, enabling efficient and fine-grained system introspection.

B. Motivation

The strict size constraints of the eBPF verifier and the growing adoption of eBPF in high-performance systems necessitate effective optimization. Optimization enables larger programs to pass the verifier while boosting throughput and lowering latency in critical workloads. Existing compilers and prior works generally optimize eBPF programs using two approaches: program synthesis and manually designed optimization rules, each with inherent trade-offs. Synthesis-based superoptimizers [13] can discover high-quality optimizations but suffer from high overhead, limiting their scalability. Rule-based methods [14] offer fast and predictable optimizations but often miss optimization opportunities. In the eBPF context, crafting correct and effective rules is especially challenging, as transformations must preserve semantics and comply with the kernel verifier's safety constraints. These limitations reduce the flexibility and effectiveness of rule-based approaches.

To address these challenges, we seek a solution that combines the strengths of synthesis-based and rule-based methods, achieving high-quality optimizations without incurring excessive computational or manual effort. Consequently, we propose a novel BPF bytecode optimization method that uses superoptimization to generate correct, verifier-safe rewrite rules, which are then applied to optimize real-world BPF programs efficiently. To maximize rule generality, we design tailored instruction abstraction and slicing strategies for BPF instructions. This facilitates the construction of a large set of rewrite rules that generalize well and can be applied effectively. Fig. 1 illustrates our workflow using a sample instruction sequence. First, the sequence is divided into semantically related segments. Each segment is then independently superoptimized, producing rewrite rules that are abstracted and stored for future reuse. Finally, the optimized segments are recombined, ensuring the original semantics are preserved.

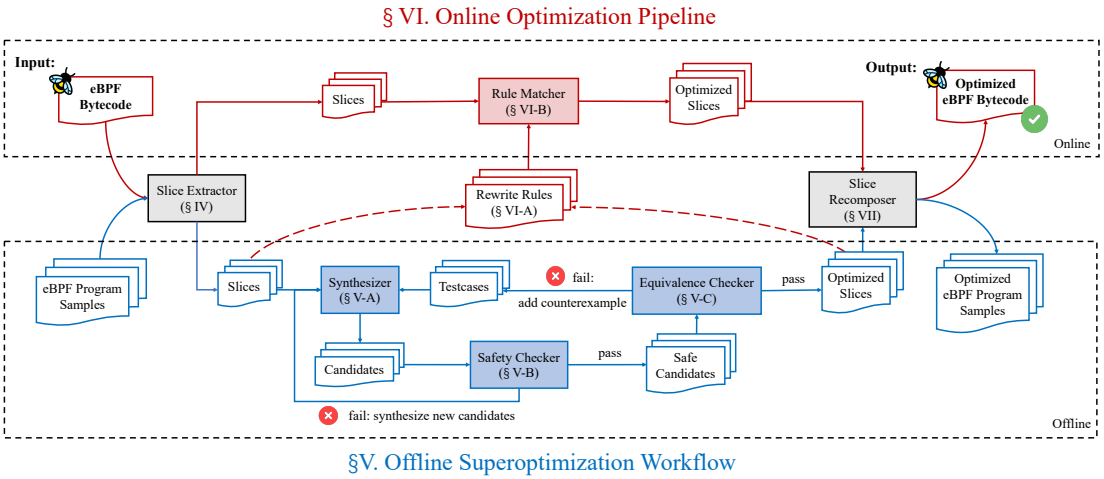


Fig. 2: An overview of EPSO.

III. Overview

Fig. 2 illustrates the overall process of EPSO, comprising an offline superoptimization workflow and an online optimization pipeline. The offline phase employs superoptimization to synthesize rewrite rules from representative eBPF sample programs, which are subsequently applied in the online phase to optimize real-world eBPF bytecode efficiently.

In both offline and online phases, inputs are first processed by the **Slice Extractor** (§IV), while the resulting optimized slices are handled by the **Slice Recomposer** (§VII). The extractor partitions instruction sequences into smaller slices that serve as optimization units, and the recomposer reassembles these slices to preserve semantic integrity.

The offline phase proceeds as follows. Given an instruction slice, the **Synthesizer** (§V-A) conducts a cost-guided enumerative search to identify a candidate program that replicates the original's behavior on a set of input-output testcases. The candidate is first evaluated by the **Safety Checker** (§V-B); if it fails to meet verifier constraints, the search continues. Otherwise, it advances to the **Equivalence Checker** (§V-C) for semantic validation. If the candidate is not equivalent with the original, a counterexample is generated and added to the testcase set to guide subsequent synthesis iterations. Only candidates that pass both safety and equivalence checks are retained. These verified candidates are then paired with their original slices to form **Rewrite Rules** (§VI-A).

Once a substantial set of rewrite rules has been collected from representative eBPF sample programs, the online optimizer can be employed to optimize real-world eBPF instruction slices. The online phase employs the **Rule Matcher** (§VI-B) to apply offline-synthesized rewrite rules, rewriting original slices into more efficient equivalents.

IV. Extracting Instruction Slices

When optimizing instruction sequences extracted from BPF bytecode, using a sliding window to form instruction slices may group unrelated instructions into the same optimization unit, reducing the reusability of the resulting rewrite rules. To enhance the generality and applicability of rewrite rules, we design a specialized slicing strategy that extracts instruction slices by individually tracing the computation chains of each register or memory region. This enables the precise grouping of optimization-relevant instructions into semantically coherent units, which then serve as reasonable optimization units.

Algorithm 1 illustrates our instruction slicing strategy, which takes an instruction sequence as input and outputs extracted instruction slices that function as optimization units. The overall procedure consists of two major phases: slice construction and slice refinement.

The **slice construction** phase (Lines 4–22) builds instruction slices by grouping together instructions that compute values for specific registers or memory locations. For each instruction I in the input sequence, the algorithm assigns an identifier ID representing the destination it writes to. STORE instructions use a memory-based ID encoded from the base register, offset, and access size, while other instructions use the destination register as ID. Instruction I is then added to $Insn\_map[ID]$, along with all instructions related to the registers or memory it reads from, ensuring each slice fully captures the computation chain for its destination. Notably, since we perform only intra-block optimizations, all instruction sequences selected for slicing originate from basic blocks within the BPF program's control flow graph. Consequently, instructions within each sequence are executed sequentially, and control-flow information is considered at the beginning of each basic block but not between instructions within the same block.

The **slice refinement** phase (Lines 23–24) further processes the collected slices to enhance optimization potential and manageability. It first merges slices related to STORE operations targeting adjacent memory regions, preserving opportunities for memory access fusion that might otherwise be lost. Then, to ensure tractability, each merged slice is partitioned into smaller segments based on a fixed window size. The resulting slices form the final output *Slices*, serving as the fundamental

Algorithm 1 Instruction Slice Extraction

Input: Instruction sequence *Insns*
Output: Instruction slices functioning as optimization units *Slices*
Insns_map ← ∅ ▷ Mapping from register/memory IDs to related instructions
for each instruction I in $Insns$ **do**
    ID ← −1
    if I is a STORE instruction **then**
        ID ← *(I.offset << 8) | (I.size() << 4) | I.dst_reg*
    else
        ID ← $I.dst\_reg$
    end if
    Insns_map[ID] ← *I*
    if *I* reads from *I.dst _reg* **then**
        Insns_map[ID] += Insns_map[I.dst_reg]
    end if
    if *I* reads from *I.src _reg* **then**
        Insns_map[ID] += Insns_map[I.src_reg]
    end if
    if I is a LOAD instruction **then**
        LD_ID ← *(I.offset << 8) | (I.size() << 4) | I.src_reg*
        Insns_map[ID] += Insns_map[LD_ID]
    end if
end for
Merge adjacent memory-related slices in *Insns_map*.
Partition each slice in *Insns_map* by window size to obtain the final *Slices* as optimization units.

units for downstream optimization.

After slicing, we obtain optimization units which each focuses on the computation of one single register or memory region. This effectively improves the generality of rewrite rules by excluding instructions irrelevant to the optimization target.

V. Superoptimizing Extracted Slices

After slicing instruction sequences into separate slices, we perform superoptimization on these slices to collect rewrite rules. These rules are later applied in Section VI to optimize real-world BPF programs.

To guarantee the equivalence between the optimized programs and their originals, superoptimization typically follows the CEGIS [28] framework. Within this framework, program equivalence between P and Q is formally defined as:

$$\forall \text{ input } x: \ \text{output}_P(x) = \text{output}_Q(x) \tag{1}$$

where $\text{output}_P(x)$ and $\text{output}_Q(x)$ denote the final runtime states of programs P and Q given the same input state x.

Since directly synthesizing a program equivalent to the original on all inputs is infeasible, CEGIS divides optimization into two stages: candidate synthesis and equivalence verification. In the **synthesis** (§V-A) phase, a finite set of testcases (i.e., input-output pairs) serve as the specification, and the synthesizer searches the program space to find a candidate program that produces the same outputs as the original under these testcases. During the **equivalence check** (§V-C) phase, a SMT solver is used to check whether the candidate is strictly equivalent to the original. If equivalent, the candidate is accepted as the final optimized result. If not, a counterexample is generated and added to the testcase set to enhance the constraints of the next synthesis round. As counterexamples accumulate from failed verifications, the constraints are progressively refined, enabling the synthesis-verification loop to converge in a few iterations and ultimately produce an equivalent candidate.

Additionally, to ensure that the optimized program not only preserves semantics but also adheres to the safety constraints enforced by the in-kernel verifier, we introduce a **safety check** (§V-B) step prior to equivalence verification. This step enforces a set of safety rules that may otherwise be violated in certain optimization scenarios.

A. Program Synthesis Using Enumerative Search

As illustrated above, the synthesis stage aims to generate candidate programs that preserve the original program's behavior across a given set of testcases. Unlike K2, which uses stochastic search that can miss valid optimizations, our approach uses enumerative search [17], [29]–[33]. This search strategy systematically explores the program space to find the optimal rewrite equivalent to the original. Furthermore, our method incorporates carefully designed pruning techniques into the search process, substantially reducing the search space and computational overhead to enable more efficient and scalable synthesis. The complete search procedure and pruning strategies are detailed below.

1) IDDFS-based Search Strategy: During the search for candidate programs, we prioritize those with fewer instructions or shorter execution time, enabling us to quickly return the first candidate that meets the test case constraints. To achieve the above search order, we employ Iterative Deepening Depth-First Search (IDDFS) [33] as the primary search algorithm.

IDDFS begins with a shallow depth limit (initially one instruction) and incrementally increases it until a valid candidate is found. This approach ensures that lower-cost candidates are explored first while effectively limiting the search depth.

2) Pruning Strategies for Speeding Up Search: Exhaustively exploring the entire program space is time-consuming, making effective pruning essential. A simple yet effective heuristic is that an optimized sequence should never exceed the cost of the original program, and candidates that surpass this cost can be pruned early.

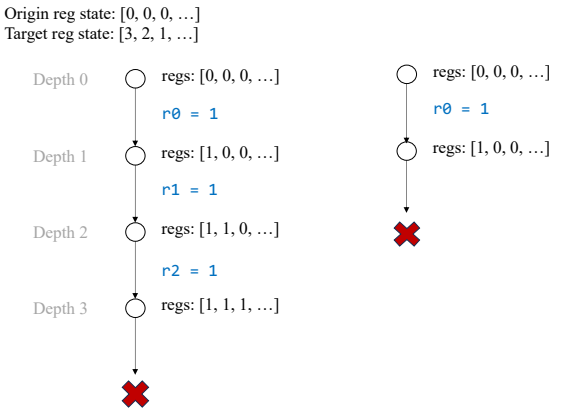
Beyond this, we apply additional pruning strategies specifically designed for the eBPF instruction set. The pruning methods used during the search are as follows:

1. Estimating the *distance* to the terminal state.

To assess the feasibility of a search branch reaching the target terminal state, we define a *distance* metric: the minimum number of instructions needed for the transition from the current state to the target. The formal definition is as follows.

Let the current state $\mathcal{S}$ be represented as $\mathcal{S} = (\mathcal{R}, \mathcal{M})$, where $\mathcal{R} = \{r_i\}_{i=1}^{n}$ denotes the states of registers and $\mathcal{M} = \{m_j\}_{j=1}^{k}$ denotes the states of memory regions. Similarly, let the target state $\mathcal{T} = (\mathcal{R}', \mathcal{M}')$, where $\mathcal{R}' = \{r_i'\}_{i=1}^{n}$ and $\mathcal{M}' = \{m_j'\}_{j=1}^{k}$. We define the *distance* metric $d(\mathcal{S}, \mathcal{T})$ as:

$$d(\mathcal{S}, \mathcal{T}) = \min_{\mathcal{I}} \{|\mathcal{I}| : \mathcal{S} \xrightarrow{\mathcal{I}} \mathcal{T}\} \tag{2}$$



(a) Without distance-estimation. (b) With distance estimation.

Fig. 3: Pruning strategy: distance estimation.

where $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ denotes an instruction sequence, $|\mathcal{I}|$ represents its length, and $\mathcal{S} \xrightarrow{\mathcal{I}} \mathcal{T}$ indicates that applying the instructions $\mathcal{I}$ to state $\mathcal{S}$ results in state $\mathcal{T}$.

Since modifying a register or memory region requires at least one instruction, $d(\mathcal{S}, \mathcal{T})$ is calculated by counting all differences between the current and target register and memory region states:

$$d(\mathcal{S}, \mathcal{T}) = \sum_{i=1}^{n} \mathbf{1}[r_i \neq r'_i] + \sum_{j=1}^{k} \mathbf{1}[m_j \neq m'_j] \tag{3}$$

where $\mathbf{1}[\cdot]$ is the indicator function that equals 1 when the condition is true and 0 otherwise.

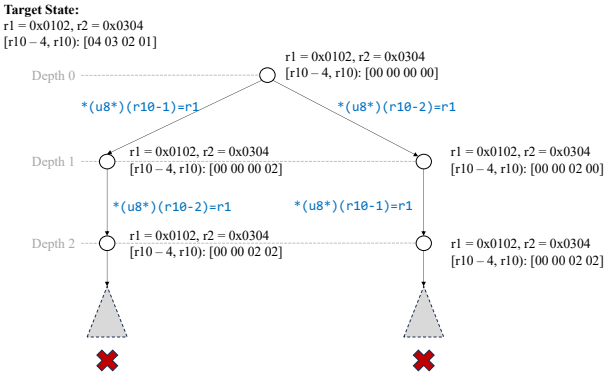
Assuming the optimized sequence does not exceed the original length, we prune any branch whose *distance* exceeds the remaining instruction budget. This heuristic is especially effective for sequences with complex register transformations, where each required state change generally corresponds to at least one instruction, enabling accurate *distance* estimation.

As illustrated in Fig. 3a, without *distance* estimation, the search proceeds until reaching the cost limit. In contrast, Fig. 3b shows that *distance* estimation allows for early termination right after the first invalid instruction. In this example, transforming three registers requires at least three instructions. Once a new instruction fails to reduce the estimated distance, the remaining budget is insufficient and the branch is immediately pruned, preventing unnecessary exploration.

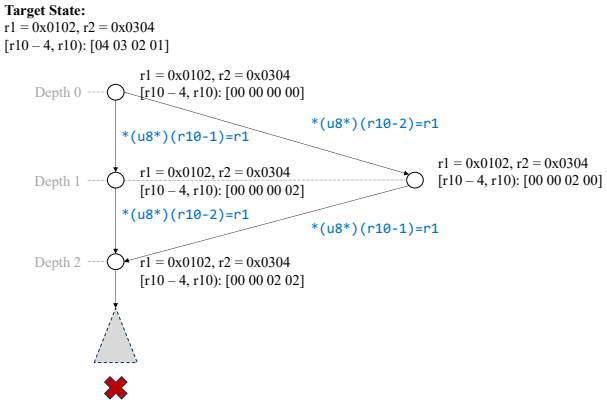
2. Memorizing failed explored states.

While the first pruning strategy works well for register transformations, it is less effective for memory operations. Memory state differences are harder to quantify due to alignment constraints and uncertain instruction granularity, making distance-based pruning less effective.

To address this, we introduce a memoization-based pruning strategy. During synthesis, we cache program states that led to failed branches along with their failure depths. If the same state recurs at an equal or greater depth, the branch is immediately pruned. Unlike distance-based pruning, this method learns



(a) Without failed state memorization.



(b) With failed state memorization.

Fig. 4: Pruning strategy: failed state memorization.

from past failures to avoid redundant exploration.

Consider the following instruction sequence, which stores the lower 16 bits of r_1 at memory region $[r_{10} - 2, r_{10})$:

```
1. *(u8*)(r10 - 2) = r1
2. r1 >>= 1
3. *(u8*)(r10 - 1) = r1
```

A more compact equivalent is:

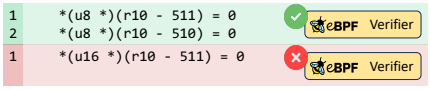
```
1. *(u16*)(r10 - 2) = r1
```

During synthesis, redundant searches often occur when semantically equivalent instruction sequences lead to the same memory state. For example, the sequences `*(u8*)(r10 - 1) = r1; *(u8*)(r10 - 2) = r1` and `*(u8*)(r10 - 2) = r1; *(u8*)(r10 - 1) = r1` produce identical memory states. If the search following the first sequence fails, the second sequence redundantly re-explores the same state and repeats the failure (Fig. 4a).

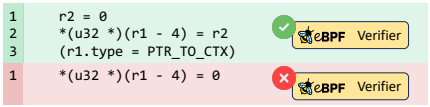
Fig. 4b shows how failed state memoization eliminates this redundancy. By recording failed states with their failure depths, any future visit to the same state at equal or greater depth is pruned immediately. This drastically reduces repeated computations and improves synthesis efficiency.

3. Disallowing Redundant Register Definitions.

During the search, when choosing a destination register, we check whether any previous definition of that register is still



(a) Unsafe optimization due to memory misalignment.



(b) Unsafe Optimization due to pointer type.

Fig. 5: Unsafe optimizations rejected by the verifier.

unused. If such an unused definition exists, the current search path is pruned immediately, as sequences with redundant operations cannot be optimal.

For example, consider this sequence:

```
1. r1 = *(u32 *) (r10 + 0)
2. r2 = *(u32 *) (r10 + 4)
3. r1 = r2
```

At instruction 3, $r1$ is redefined without its previous value being used. A definition-use analysis detects this redundancy, allowing the search to prune this suboptimal path early.

B. Safety Check

Once a candidate is synthesized, we must ensure it meets the verifier's constraints, including program size limits, structural requirements, and instruction legality. Since our optimization operates at the intra-block level, structural and size checks are less critical. The primary challenge lies in ensuring the transformed code complies with the verifier's instruction legality.

To address this, we analyze the verifier's core instruction validation routine, the *do_check* function, and encode its relevant constraints into the synthesis process to ensure all generated candidates can pass the verifier.

In practice, we observed several cases where semantically correct candidates were rejected due to subtle verifier rules. Two representative examples are discussed below.

1. Memory address alignment.

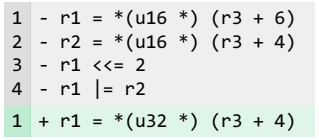
As shown in Fig. 5a, merging two 8-bit store instructions into a single 16-bit store is a valid optimization in principle. However, the verifier enforces strict memory alignment: the offset from the stack base to the 16-bit store address must be a multiple of 16. As a result, this optimization is invalidated, as it violates the verifier's memory access alignment rule.

2. The destination register type of the BPF_ST instruction cannot be PTR_TO_CTX.

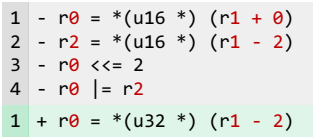
As shown in Fig. 5b, although the optimized code is semantically equivalent to the original when register $r2$ is no longer live, the verifier rejects it due to a strict rule: if the base register in a memory access is a context pointer, the source operand must not be an immediate value.

C. Equivalence Check

After verifying the candidate's safety, we perform final equivalence checking to ensure it is functionally equivalent to the original. The equivalence between program P and program Q is defined in Eq. 1, and the negation of this formula is:



(a) Original rewrite rule



(b) Abstracted rewrite rule

Fig. 6: Example of rewrite rule abstraction.

$$\exists \text{ input } x: \ \text{output}_P(x) \neq \text{output}_Q(x) \tag{4}$$

By proving Eq. 4 false, we can establish the equivalence of the two programs. If Eq. 4 is satisfiable, it indicates that there exists at least one input for which programs P and Q produce different outputs, thereby disproving their equivalence. Conversely, if Eq. 4 is unsatisfiable, no such input exists, and the equivalence of P and Q is proven.

Based on Eq.4, we formulate the final constraint for the SMT solver as shown in Eq.5. To further expand the optimization space, we incorporate additional constraints into this formula in practice, including register and memory liveness, as well as register value ranges.

$$\begin{aligned} &\text{input}_P = \text{input}_Q \\ &\wedge \text{execution behavior of program P} \\ &\wedge \text{execution behavior of program Q} \\ &\wedge \text{output}_P \neq \text{output}_Q \end{aligned} \tag{5}$$

VI. Representing and Matching Rewrite Rules

A. Representing Rewrite Rules

After optimizing the extracted instruction slices, we obtain a set of rewrite rules, each comprising a pair of original and rewritten instruction slices. Using concrete operand values to represent these rules leads to overly specific rewrite rules that are difficult to match in practice. In reality, the effectiveness of a rewrite rule largely depends on factors such as instruction opcodes and relative memory offsets, rather than specific register identifiers or accurate memory addresses.

To enhance the generality of rewrite rules, we abstract away non-essential operand values, retaining only those that directly influence the optimization. This abstraction allows the rules to be applied more broadly across diverse instruction instances.

We implement this abstraction through a normalization process. Registers are renamed in order of appearance using symbolic identifiers (e.g., r_0, r_1, etc.). For memory offsets, the first offset relative to a base register is normalized to 0, and subsequent offsets from the same base are expressed as relative differences. An example is illustrated in Fig. 6.

B. Matching Rewrite Rules

Since rewrite rules are stored in abstracted form, the instruction sequence to be optimized must undergo the same

Algorithm 2 Instruction Slice Recomposition

```
1: Input: Optimized instruction slices Slices
2: Output: Optimized instruction sequence Sequence
3: Insns_set ← ∅
4: In_edges ← ∅
5: Out_edges ← ∅
6: for each slice (id, insns) in slices do
7:     for each instruction I in insns do
8:         ID ← definition ID of I
9:         for each instruction J in insns after I  do
10:            if J uses ID then
11:                In_edges[J]+ = I
12:                Out_edges[I]+ = J
13:            end if
14:            if J redefines ID then
15:                break
16:            end if
17:        end for
18:        if I is identical to instruction J in insns_set then
19:            Redirect I related edges to J
20:        else
21:            for each instruction J in insns_set do
22:                if I and J come from different slices then
23:                    Add cross-slice dependency edges
24:                    based on def-use relations
25:                end if
26:            end for
27:            Add I to insns_set
28:        end if
29:    end for
30: end for
31: Sequence ← TopologicalSort(in_edges, out_edges)
32: Append unsorted nodes from insns_set to Sequence
33: return Sequence
```

abstraction process to enable successful rule matching. After a match is found, a de-abstraction step is performed to reconstruct the concrete optimized sequence. This step leverages the mapping between abstract and real registers, as well as the recorded base offsets, both collected during the original abstraction phase. Additionally, after each rule application, we perform formal equivalence and safety checks to ensure that all transformations preserve program semantics and safety.

VII. Recomposing Optimized Instruction Slices

With extracted instruction slices optimized, we need to further recompose the optimized instruction slices into a complete instruction sequence. To ensure the correctness of the reassembled slices, the recombination must respect the def-use relationships of registers and memory regions.

Algorithm 2 presents a def-use-aware instruction slice recomposition strategy. The core idea is to construct a def-use graph of instructions and reorganize instructions from different slices following a topological ordering. The process is structured as follows:

The **intra-slice dependency construction** phase (Lines 9–17) analyzes data dependencies for each slice. It tracks each instruction's defined value (e.g., register or memory location) and adds edges to subsequent instructions that use this value, stopping when a redefinition is encountered.

The **cross-slice deduplication and dependency construction** phase (Lines 18–28) merges identical instructions across slices to reduce redundancy. To preserve semantics correctness, it also adds cross-slice dependency edges based on def-use relationships.

The **topological ordering** phase (Lines 31–32) applies topological sort to determine a valid instruction order based on the dependency graph. Unrelated instructions not included in the graph are appended at the end.

VIII. Evaluation

We evaluate the performance of EPSO by answering the following questions:

- RQ1: How effectively does EPSO reduce program size to improve compactness? (§VIII-A)
- RQ2: How effectively does EPSO improve program efficiency? (§VIII-B)
- RQ3: What is the optimization overhead introduced by EPSO, and how do different pruning strategies impact it during the superoptimization phase? (§VIII-C)
- RQ4: How effective are the rewrite rules generated during the superoptimization phase? (§VIII-D)

Experimental Setup. We conducted all experiments on a workstation with Intel Core i7-12700H CPU (2.30GHz), 32GB RAM, running Ubuntu 22.04.1 LTS with Linux kernel v5.15.0.
Benchmark Selection. The benchmark suites used in this paper are sourced from several open-source projects containing eBPF programs: the Linux kernel [19] Cilium [20] Katran [21] Tetragon [24] Tracee [25] Sysdig [23] and hXDP [22].

As different RQs focus on different aspects, and existing works such as K2 and Merlin also adopt distinct benchmark selections. To comprehensively compare the performance of our tool EPSO with K2 and Merlin, we select tailored benchmark suites for each RQ. Specifically:

For **RQ1 (program compactness)**, we use three types of benchmark suites:

- K2 benchmark suite: 19 programs selected by K2 as benchmarks, sourced from the Linux kernel [19], Cilium [20], Katran [21], and hXDP [22].
- Merlin benchmark suite: 519 eBPF programs selected by Merlin as benchmarks, drawn from Tetragon [24], Sysdig [23], and Tracee [25].
- Diverse benchmark suite: 30 eBPF programs from various sources, including the Linux kernel [19], Tetragon [24], Cilium [20], and Katran [21].

Due to differences in tool implementation and input format support, Merlin cannot be evaluated on K2's benchmarks, which are encoded using a special format in K2's artifact (supported by EPSO but not by Merlin, which only accepts IR or bytecode as input). Conversely, K2 is not evaluated on Merlin's benchmarks because they are generally large-scale, and K2's optimization process is prohibitively slow (e.g., nearly 48 hours for a program with over 1700 instructions).

To ensure a fair comparison, we evaluate EPSO against K2 using the K2 benchmark suite and against Merlin using the

TABLE I: Experimental configurations for each research question (RQ).

RQ	Baseline(s)	Compiler Optimization Level	Optimization Mode	Rule-extraction Set	Optimization Set
RQ1	K2	-Os	Offline	–	K2 benchmark suite
	Merlin	-O2	Offline	–	Merlin benchmark suite
	K2 & Merlin	-O2	Offline	–	30 eBPF programs from diverse sources
RQ2	K2 & Merlin	-O3	Offline	–	6 performance-oriented eBPF programs
RQ3	K2	-Os	Offline	–	K2 benchmark suite
			Online	K2 benchmark suite	
			Hybrid	K2 benchmark suite	
RQ4	EPSO in offline mode	-O2	Online	20% of Merlin benchmark suite	80% of Merlin benchmark suite

Merlin suite. Additionally, we introduce a third benchmark suite of 30 diverse eBPF programs spanning multiple sources and program sizes, enabling a direct comparison of EPSO, K2, and Merlin in terms of program size reduction.

For **RQ2 (program efficiency)**, we select 6 high-performance eBPF programs from the Linux kernel [19], comparing runtime reductions of EPSO, K2 and Merlin.

For **RQ3 (optimization overhead)**, we compare EPSO and synthesis-based K2 using the K2 benchmark suite from RQ1. This evaluates their optimization overhead and EPSO's performance across different configurations. Additionally, we use the first 18 small programs from the suite to study the impact of pruning strategies, as optimization overhead can be excessive for large eBPF programs without pruning.

For **RQ4 (rewrite rule effectiveness)**, we use 20% of the Merlin benchmark suite from RQ1 as the rule-extraction set and the remaining 80% as the optimization set, where the extracted rewrite rules are applied to perform optimizations. The optimization results are compared with direct superoptimization on the same benchmarks, demonstrating the generality and effectiveness of the learned rules.

Experimental Configurations. Since different RQs focus on distinct evaluation aspects and baselines, their experimental setups differ in compiler optimization level, optimization mode (offline or online), and dataset partitioning for rule-extraction and optimization. Table I summarizes these configurations.

For RQ1 and RQ2, which evaluate optimization effects, we apply direct superoptimization to all benchmarks. Consequently, as shown in Table I, the rule-extraction set is empty and the validation set corresponds to the benchmark suite directly optimized through superoptimization.

For RQ3, which investigates the optimization overhead of EPSO under different configurations, evaluations are conducted in offline, online, and hybrid modes. In offline mode, benchmarks are directly superoptimized. In online mode, rewrite rules are first extracted from the target benchmarks, and then applied in online optimization to measure its overhead. In hybrid mode, the strategy is similar, except that superoptimization is applied when rule matching fails. As the rule-extraction and optimization sets are identical, EPSO achieves the same optimization effects across all three configurations. In RQ4, we will examine differences in optimization effects using distinct rule-extraction and optimization sets.

For RQ4, which examines the transferability of rewrite rules, we evaluate EPSO in the online mode using 20% of Merlin's benchmark suite as the rule-extraction set and the remaining 80% as the optimization set. The results are then compared with those obtained through direct superoptimization, allowing us to assess the effectiveness and generalizability of the collected rules on unseen benchmarks.

Finally, the compiler optimization level is set to align with baseline configurations for fair comparison and to accommodate the specific optimization objectives.

A. RQ1: Program Compactness

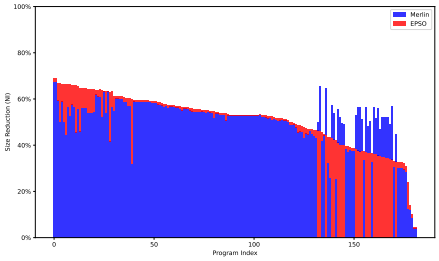
Fig. 7 presents a comparison of program compactness improvements achieved by EPSO, K2, and Merlin. In the EPSO vs. K2 comparison (Fig. 7d), EPSO improves compactness by 6.25%–25.00% (avg. 14.40%) and matches or outperforms K2 on all 19 benchmarks, highlighting the limitations of K2's stochastic search, which can miss potential optimizations.

In the EPSO vs. Merlin comparison (Fig. 7a–Fig. 7c), EPSO achieves optimization rates ranging from 4.59% to 68.87% (avg. 51.30%) on Sysdig, 0%–37.79% (avg. 10.12%) on Tracee, and 0%–52.24% (avg. 13.67%) on Tetragon. Overall, EPSO outperforms Merlin on 97.28% of benchmarks on Tracee, 94.82% on Tetragon, and 86.59% on Sysdig. Our analysis shows that EPSO performs worse on some samples mainly due to large optimization units that exceed the 10-minute synthesis timeout. To address this, we plan an adaptive timeout strategy that adjusts limits based on program traits, enabling discovery of more complex rewrite rules.

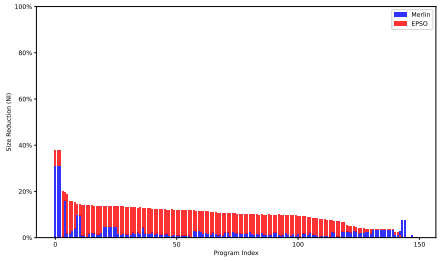
In the comprehensive comparison (Fig. 7e), EPSO outperforms K2 on all benchmarks and Merlin on 29 of 30 benchmarks. EPSO achieves program compactness improvements from 3.05% to 25.00% (avg. 11.12%), outperforming K2 and Merlin by 82.30% and 48.46%, respectively. The only benchmark where EPSO lags behind Merlin involves inter-block optimization, which EPSO does not yet support.

Regarding the comparison between K2 and Merlin, each excels in different scenarios. K2 often outperforms Merlin on smaller programs through cost-driven search, but its stochastic nature can miss optimization opportunities. Its performance is also sensitive to parameters. In our experiments, we evaluated 16 different parameter settings and reported the best results.

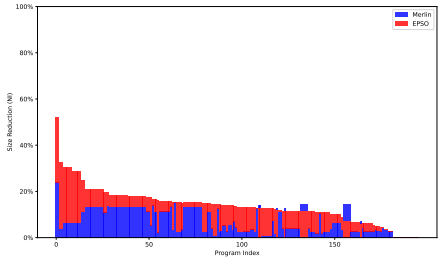
RQ1: EPSO achieves up to 68.87% (avg. 24.37%) program compactness improvement, outperforming K2 on all benchmarks and Merlin on 92.68% of them. It may underperform Merlin when inter-block optimizations are needed or optimization units exceed synthesis timeouts.



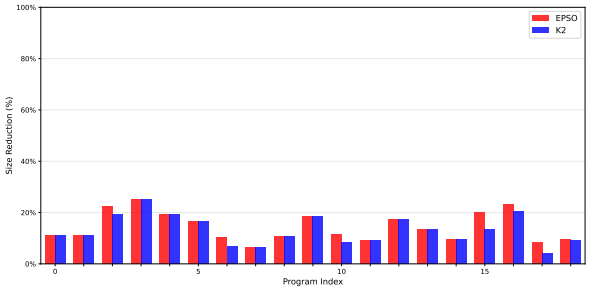
(a) Compactness comparison between EPSO and Merlin on Sysdig.



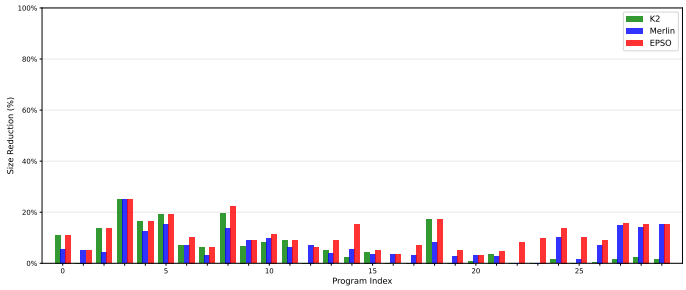
(b) Compactness comparison between EPSO and Merlin on Tracee.



(c) Compactness comparison between EPSO and Merlin on Tetragon.



(d) Compactness comparison between EPSO and K2 on the K2 benchmark suite.



(e) Compactness comparison between EPSO, K2 and Merlin on a diverse benchmark suite.

Fig. 7: Comparison of program compactness improvement achieved by EPSO, K2 [13] and Merlin [14]across benchmarks from the Linux kernel [19], Cilium [20], Katran [21], Sysdig [23], Tetragon [24] and Tracee [25].

TABLE II: Improvements of *estimated* program runtime on benchmarks from Linux kernel and Cilium.

Benchmark	Program Runtime(ns)						
	Clang (-Os)	EPSO	Compression	K2	Compression	Merlin	Compression
(1) xdp1_kern/xdp1	25.58	23.80	6.96%	24.55	4.03%	24.27	5.12%
(2) xdp2_kern/xdp1	30.88	29.20	5.44%	29.76	3.63%	29.57	4.24%
(3) xdp_fwd	51.56	41.74	19.05%	43.73	15.19%	48.96	5.04%
(4) xdp_pktcntr	12.32	11.85	3.82%	11.85	3.82%	12.21	0.89%
(5) cgroup/recvmsg4	65.54	65.05	0.75%	65.46	0.11%	65.05	0.75%
(6) cgroup/sendmsg4	297.37	286.75	3.57%	295.85	0.51%	297.93	-0.19%
Avg. of all benchmarks			6.60%		4.55%		2.64%

B. RQ2: Program Efficiency

In efficiency-oriented optimizations, direct execution of candidate programs to measure performance is infeasible, as most are rejected by the verifier. Following K2's methodology, we estimate program runtime as the sum of instruction latencies, with each instruction profiled by executing its opcode millions of times and recording the average runtime. In real-world networks, efficiency gains boost throughput and reduce packet latency. Our tool, deployed industrially, shows 5.5% higher QPS under high load, improving service response speed. Table II shows EPSO's efficiency improvements on six high-performance benchmarks from the Linux kernel [19] (1–4) and Cilium [20] (5–6), compared with K2 and Merlin.

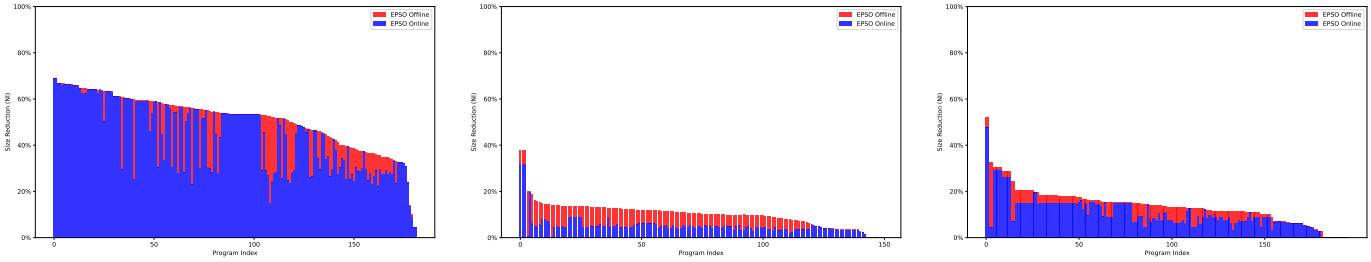
Experiments show that EPSO achieved optimization rates of 0.75%–19.05% (avg. 6.60%), outperforming K2 and Merlin by 45.05% and 149.77%, respectively. While K2 and Merlin match EPSO on some benchmarks, K2's stochastic search and Merlin's limited rule coverage lead to gaps on others. Notably, Merlin produced a negative rate (-0.19%) on *cgroup/sendmsg4*, indicating occasional regressions.

RQ2: EPSO improves program efficiency by 0.75% to 19.05% (avg. 6.60%), outperforming both K2 and Merlin across all six benchmarks. In contrast, K2 and Merlin miss certain optimization opportunities due to reliance on stochastic search methods or limited rule coverage.

C. RQ3: Optimization Overhead

We evaluate EPSO's optimization overhead under three configurations: (1) synthesis-only (using only superoptimization); (2) rule-matching-only (using only pre-collected rewrite rules); and (3) hybrid (rule matching first, then synthesis if no rules apply). For comparison, we also report the optimization overhead of K2, which uses synthesis-based techniques. Merlin, as a rule-based optimizer with negligible runtime overhead, is excluded from this analysis.

Fig. 9a shows the optimization overhead of K2 and EPSO under three configurations across 19 benchmarks. In synthesis-only mode, EPSO reduces overhead by 88.71% on average versus K2. In rule-matching-only mode, where rewrite rules are

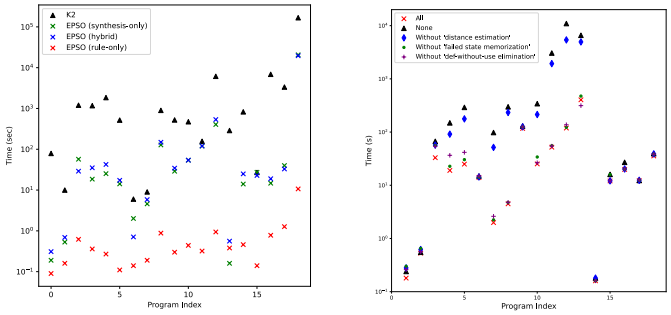


(a) EPSO compactness on Sysdig [23]: offline vs. online modes.

(b) EPSO compactness on Tracee [25]: offline vs. online modes.

(c) EPSO compactness on Tetragon [24]: offline vs. online modes.

Fig. 8: Evaluation of learned rewrite rule effectiveness: offline vs. online.



(a) Comparison of optimization overhead between K2 and EPSO (under 3 configurations.)

(b) Optimization overhead of EPSO with different pruning strategies applied.

Fig. 9: Optimization overhead of K2 and EPSO under different configurations.

pre-synthesized and reused, EPSO incurs negligible overhead, making it highly effective for large-scale optimizations. The hybrid mode, which prioritizes rule matching and falls back to synthesis, outperforms synthesis-only on some benchmarks but has slightly higher overhead on others due to low rule-match rates. Finally, since the rule-extraction and optimization sets used for online optimization are identical, EPSO achieves consistent optimization effects across all three configurations.

To evaluate how the three pruning strategies reduce optimization overhead, we conducted ablation experiments by disabling each one separately and measuring EPSO's runtime. We also measured overhead when no pruning strategies were used. The results are presented in Fig. 9b.

Experiments show that all three pruning strategies significantly reduce optimization overhead, with distance estimation being the most effective. Without pruning, synthesis time can increase up to 91×, highlighting pruning's critical role in improving efficiency.

RQ3: EPSO reduces optimization overhead by 88.71% on average versus K2 in synthesis-only mode, with negligible overhead via rule matching. Pruning, especially distance estimation, cuts synthesis time significantly. Without pruning, synthesis time can increase up to 91×.

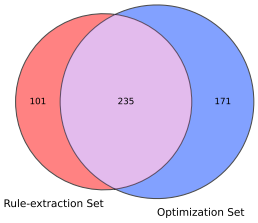


Fig. 10: Overlap of discovered rewrite rules between rule-extraction and optimization sets.

D. RQ4: Rewrite Rule Effectiveness

To evaluate collected rewrite rules, we run superoptimization on 20% of the Merlin benchmark suite and apply learned rules to the remaining 80% to test generality. Fig. 8 compares online optimization and direct superoptimization across three projects. Results show rule matching alone reproduces 28.37%–100% (avg. 86.69%), 2.21%–100% (avg. 55.63%), and 13.56%–100% (avg. 79.03%) of superoptimization's effects on Sysdig, Tracee, and Tetragon, respectively, proving the rules' strong generality and effectiveness.

During the experiments, we observed that the discovered rewrite rules vary in effectiveness, frequency of application, and ease of discovery. We highlight two representative examples. Example 1 yields significant reductions in instruction count and runtime and is frequently applied, thereby contributing substantially to overall optimization. Example 2 improves register allocation, which is difficult to identify manually but can be automatically uncovered through superoptimization.

Example1:

```
1. r1 = *(u32 *)(r0 + 8)
2. r2 = *(u32 *)(r0 + 12)
3. r2 <<= 32
4. r2 |= r1
=>
1. r2 = *(u64 *)(r0 + 8)
(r1 is not used afterwards)
```

Example2:

```
1. r3 = r7
2. r3 += r1
3. r3 = *(u16 *)(r3 + 2)
4. r1 = r4
=>
1. r1 += r7
2. r3 = *(u16 *)(r1 + 2)
3. r1 = r4
```

Fig. 10 presents a Venn diagram of rewrite rules from the rule-extraction and optimization sets, where the intersection indicates rules matched during online optimization. Despite only

57.88% of validation-set rewrite rules being discovered during rule extraction and applied in online optimization, frequent use of high-impact rules enables online optimization to achieve, on average, 75.04% of the effects of direct superoptimization, illustrating variability in rule effectiveness and usage.

RQ4: Evaluating 336 rewrite rules on an unseen benchmark suite, rule matching reproduces 2.21%–100% (avg. 75.04%) of the optimizations achieved by superoptimization, demonstrating the strong generality and effectiveness of the collected rules on unseen programs.

IX. Discussion

Prior works K2 [13] and Merlin [14] optimize eBPF programs using synthesis-based and rule-based approaches, respectively, but both exhibit limitations: K2 incurs high optimization overhead, whereas Merlin achieves limited optimization quality. To achieve high-quality optimizations efficiently, our approach adopts a synthesis-based superoptimization strategy like K2, but introduces several key enhancements.

First, we turn superoptimization offline for rewrite rule collection and employ efficient rule matching when optimizing real-world programs, reducing optimization overhead to negligible levels. This is enabled by novel techniques that enhance rule generality and increase successful matches. In contrast, although K2 collects optimization cases during superoptimization, it lacks mechanisms to improve the generality of the collected rules, which is essential for effective reuse. Consequently, the narrowly scoped rules rarely match new code, resulting in repeated and costly synthesis and, therefore, high optimization overhead. Second, unlike K2's stochastic search, we employ enumerative search to uncover more optimization opportunities. Coupled with our pruning strategies, synthesis is further accelerated, making EPSO more efficient than K2 even as a pure superoptimizer.

While our approach demonstrates efficient and effective eBPF optimization, several extensions merit further exploration. First, supporting inter-block optimization could enable deeper transformations by allowing modifications across basic block boundaries, rather than restricting them to intra-block scopes. Second, adaptive adjustment of window size and timeout based on program structure could further reduce overhead and enhance optimization quality. Currently, discovering diverse rewrites requires multiple superoptimization rounds with varying window sizes. In our experiments, 795 well-generalized rewrite rules were identified with window sizes ranging from 1 to 62. As synthesis time depends more on rewrite size than window size, EPSO often completes within the timeout even for large programs, provided rewrites are small. However, a fixed timeout may impede the discovery of more complex rewrites that require longer synthesis. Developing adaptive strategies for window sizing and timeout tuning thus represents a promising direction to further improve both efficiency and optimization quality.

X. Related Work

Program synthesis [34] aims to generate programs that satisfy given specifications, such as input-output examples, demonstrations, or natural language descriptions. In the context of code optimization, the specification is the original program, and the goal is to produce functionally equivalent code with improved performance.

Superoptimization [29] leverages program synthesis to find more efficient equivalents by exhaustively exploring the program space. Common strategies include enumerative search [17], [29]–[32], [35]–[37], constraint solving [38]–[43], and stochastic search [13], [44]. Although superoptimization produces high-quality results, its practicality is often limited by high computational cost, especially for large-scale programs. Here, we adopt enumerative search with pruning for efficient optimizations. In addition to established pruning techniques [29]–[31], we also introduce a novel distance-estimation-based pruning strategy tailored for eBPF's register-centric semantics, which is highly effective in the BPF context.

Despite various pruning strategies, program synthesis remains time-consuming due to its exhaustive search nature and scales poorly to large programs. Inspired by prior work [17], [18], we offload superoptimization to an offline phase to extract high-quality rewrite rules, which are then efficiently applied to real-world BPF programs via rule matching. To enhance rule generality and reusability, we design a specialized slicing strategy and instruction abstraction mechanism tailored for BPF instructions, both proven effective.

XI. Conclusion

We propose an optimization approach for eBPF programs that shifts the costly superoptimization process to an offline phase for discovering rewrite rules, while employing lightweight rule matching to optimize real-world BPF programs. This approach effectively combines the strengths of synthesis-based and rule-based techniques, achieving high-quality optimizations with negligible runtime overhead. Across diverse BPF programs of varying sources and scales, EPSO discovers a total of 795 rewrite rules. It achieves up to 68.87% (avg. 24.37%) improvement in program compactness compared to Clang's output, outperforming the state-of-the-art BPF optimizer K2 on all benchmarks and Merlin on 92.68% of them. Furthermore, EPSO reduces program runtime by an average of 6.60%, thereby enhancing throughput and lowering latency in network applications. Additionally, it reduces optimization overhead by 88.71% compared to K2 in synthesis-only mode, while incurring negligible overhead when employing rule matching.

Acknowledgments

We are grateful for the constructive feedback of all anonymous reviewers to improve this manuscript. The authors are supported in part by the National Key Research and Development Program of China (2024YFB2505604) and the National Natural Science Foundation of China (No.62232008, 62172200).

REFERENCES

[1] "ebpf documentary," 2024, accessed: 2024-06-01. [Online]. Available: https://ebpf.io/what-is-ebpf/

[2] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[3] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.

[4] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[5] S. Miano, X. Chen, R. B. Basat, and G. Antichi, "Fast in-kernel traffic sketching in ebpf," *ACM SIGCOMM Computer Communication Review*, vol. 53, no. 1, pp. 3–13, 2023.

[6] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 1. IEEE, 2018, pp. 209–217.

[7] D. Calavera and L. Fontana, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[8] B. Gregg, *BPF performance tools*. Addison-Wesley Professional, 2019.

[9] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso, "ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond)," *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023.

[10] S. Kerner, "Bpf for security—and chaos—in kubernetes," 2019, accessed:2024-06-01. [Online]. Available: https://lwn.net/Articles/790684/

[11] S.-Y. Wang and J.-C. Chang, "Design and implementation of an intrusion detection system by using extended bpf in the linux kernel," *Journal of Network and Computer Applications*, vol. 198, p. 103283, 2022.

[12] J. Nam, S. Lee, P. Porras, V. Yegneswaran, and S. Shin, "Secure inter-container communications using xdp/ebpf," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 934–947, 2022.

[13] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman, "Synthesizing safe and efficient kernel extensions for packet processing," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 50–64.

[14] J. Mao, H. Ding, J. Zhai, and S. Ma, "Merlin: Multi-tier optimization of ebpf code for performance and compactness," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 639–653.

[15] M. Bonola, G. Belocchi, A. Tulumello, M. S. Brunella, G. Siracusano, G. Bianchi, and R. Bifulco, "Faster software packet processing on {FPGA}{NICs} with {eBPF} program warping," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 987–1004.

[16] H.-C. Kuo, K.-H. Chen, Y. Lu, D. Williams, S. Mohan, and T. Xu, "Verified programs can party: optimizing kernel extensions via post-verification merging," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 283–299.

[17] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 394–403, 2006.

[18] Z. Liu, S. Mada, and J. Regehr, "Minotaur: A simd-oriented synthesizing superoptimizer," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1561–1585, 2024.

[19] "Bpf samples from linux," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/torvalds/linux/tree/master/samples/bpf

[20] "Cilium," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/cilium/cilium

[21] "Katran," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/facebookincubator/katran

[22] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hxdp: Efficient software packet processing on fpga nics," *Communications of the ACM*, vol. 65, no. 8, pp. 92–100, 2022.

[23] "Sysdig," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/draios/sysdig

[24] "Tetragon," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/cilium/tetragon

[25] "Tracee," 2025, accessed: 2025-03-19. [Online]. Available: https://github.com/aquasecurity/tracee

[26] "bpftrace," 2025, accessed: 2025-04-27. [Online]. Available: https://github.com/bpftrace/bpftrace

[27] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli, "Kernel-level tracing for detecting stegomalware and covert channels in linux environments," *Computer Networks*, vol. 191, p. 108010, 2021.

[28] A. Solar-Lezama, *Program synthesis by sketching*. University of California, Berkeley, 2008.

[29] H. Massalin, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, pp. 122–126, 1987.

[30] T. Granlund and R. Kenner, "Eliminating branches using a superoptimizer and the gnu c compiler," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, 1992, pp. 341–352.

[31] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 297–310.

[32] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 2017, pp. 319–336.

[33] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[34] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[35] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1159–1174.

[36] R. Alur, P. Černỳ, and A. Radhakrishna, "Synthesis through unification," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 163–179.

[37] V. Srinivasan and T. Reps, "Synthesis of machine code from semantics," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 596–607.

[38] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 304–314, 2002.

[39] A. Jangda and G. Yorsh, "Unbounded superoptimization," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017, pp. 78–88.

[40] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster, "Adaptive concretization for parallel program synthesis," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 377–394.

[41] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 62–73, 2011.

[42] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in smt," in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II 27*. Springer, 2015, pp. 198–216.

[43] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 215–224.

[44] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.