Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness

Jinsong Mao
jinsongmao@umass.edu
University of Massachusetts Amherst
USA

Hailun Ding
hailun.ding@rutgers.edu
Rutgers University
USA

Juan Zhai
juanzhai@umass.edu
University of Massachusetts Amherst
USA

Shiqing Ma
shiqingma@umass.edu
University of Massachusetts Amherst
USA

Abstract

eBPF (extended Berkeley Packet Filter) significantly enhances observability, performance, and security within the Linux kernel, playing a pivotal role in various real-world applications. Implemented as a register-based kernel virtual machine, eBPF features a customized Instruction Set Architecture (ISA) with stringent kernel safety requirements, e.g., a limited number of instructions. This constraint necessitates substantial optimization efforts for eBPF programs to meet performance objectives. Despite the availability of compilers supporting eBPF program compilation, existing tools often overlook key optimization opportunities, resulting in suboptimal performance. In response, this paper introduces Merlin, an optimization framework leveraging customized LLVM passes and bytecode rewriting for Instruction Representation (IR) transformation and bytecode refinement. Merlin employs two primary optimization strategies, i.e., instruction merging and strength reduction. These optimizations are deployed before eBPF verification. We evaluate Merlin across 19 XDP programs (drawn from the Linux kernel, Meta, hXDP, and Cilium) and three eBPF-based systems (Sysdig, Tetragon, and Tracee, each comprising several hundred eBPF programs). The results show that all optimized programs pass the kernel verification. Meanwhile, Merlin can reduce number of instructions by 73% and runtime overhead by 60% compared with the original programs. Merlin can also improve the throughput by 0.59% and reduce the latency by 5.31%, compared to state-of-the-art technique K2 , while being 10^6 times faster and more scalable to larger and more complex programs without additional manual efforts.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651387

CCS Concepts: • **Hardware** → *Emerging languages and compilers*; • **Software and its engineering** → **Software performance**.

Keywords: eBPF Optimization, LLVM

ACM Reference Format:
Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651387

1 Introduction

Extended Berkeley Packet Filter (eBPF) is a Linux kernel technology that allows safe, sandboxed execution of user-provided programs to efficiently observe and trace various kernel objects like networking stacks, system calls, and file systems [8]. eBPF provides a programmable interface for users to customize monitoring and performance analysis without having to modify kernel source code. Since its introduction in Linux kernel 4.4, eBPF has been widely adopted in major cloud providers and companies to build efficient observability tools [1, 2, 5, 6, 12]. These tools greatly boost security and visibility of cloud services [1, 6] and enables better network scalability and performance [2, 5, 12].

The eBPF bytecode executed inside the kernel has an independent architecture from native code and can be directly interpreted or Just-In-Time (JIT) compiled to native instructions for efficiency. eBPF programs are written in C/C++ and compiled into eBPF bytecode via the LLVM compiler toolchain. The Linux kernel provides a restricted virtual machine via eBPF hooks like tracepoints, kprobes, and sockets to execute the bytecode safely. Before loading any eBPF program into the kernel, a verifier statically analyzes it to ensure safety and validity. The verifier enforces constraints on instruction sequences, data types, memory accesses, loop bounds etc., to guarantee the program does not violate kernel memory protections or crash the system.

The limited bytecode format coupled with stringent verifier checks necessitates heavily optimizing eBPF programs. First, eBPF bytecode has a restricted length and limited number of instructions per program. The verifier rejects any program exceeding these limits to prevent unbounded execution in kernel. Without optimization, many useful programs risk tripping these constraints leading to failed loads. Second, eBPF hooks deeply integrate with critical kernel paths like networking and syscalls. Any inefficiency in attached eBPF programs can significantly impact overall system performance and throughput.

However, existing compiler toolchains like LLVM have limited eBPF-specific optimization capabilities. General purpose compilers lack intricate knowledge of eBPF virtual machine intricacies and verifier constraints needed to perform suitable optimizations. Customizing them requires extensive manual effort and deep kernel expertise. Further, the verifier's stringent validation frequently rejects valid programs as it evolves to protect kernel integrity. This renders many conventional optimizations inapplicable and makes enhancing eBPF code highly challenging.

Currently, only few researches are done to resolve the optimization issue. K2 [57] formalizes equation check and verification of eBPF programs, and uses stochastic search to find a more efficient program. However, this approach is time-consuming, potentially requiring months to identify an optimal solution for extensive eBPF programs. KFuse [36] enhances performance by merging post-verification programs through the detection of program chains, thereby minimizing the penalties of indirect calls. Nonetheless, this method does not apply direct optimizations to the code itself.

To address these issues, we analyze the Linux eBPF documentation and implementation [10], identify missing optimizations, and design an optimization framework that holistically integrates custom optimizations into the standard eBPF build process. Our key insight is leveraging two complementary techniques: ① Transforming LLVM IR via custom optimization passes to exploit eBPF-aware improvements early in compilation pipeline. ② Refining final eBPF bytecode to target verifier constraints and utilize eBPF virtual machine features. The IR optimizations inject domain knowledge like registers and alignment into the compiler IR. The bytecode refinements directly optimize bytecode right before loading into kernel. This two-phase approach allows exploiting optimization opportunities at different granularities. We introduce optimizations like instruction merging, data alignment, and use of eBPF atomic operations to generate efficient code. Our customized LLVM passes and bytecode analyzers integrate seamlessly into the standard eBPF build process without any workflow disruption or failing eBPF verification.

We implement the proposed techniques in a system called Merlin and evaluate it on various real-world eBPF programs from domains like networking, observability, and security. The results show that Merlin can reduce the NI (Number of Instructions) by 73% and runtime overhead by 60% compared with the original programs. Meanwhile, Merlin can also improve the throughput by 0.59% and reduce the latency by 5.31%, compared to state-of-the-art technique K2. Our fine-grained IR and bytecode optimizations significantly enhance the efficiency of eBPF program without failing the verification. The compiler integrated approach also simplifies adoption. The improved performance and compactness have positive implications for eBPF's use in latency-sensitive cases.

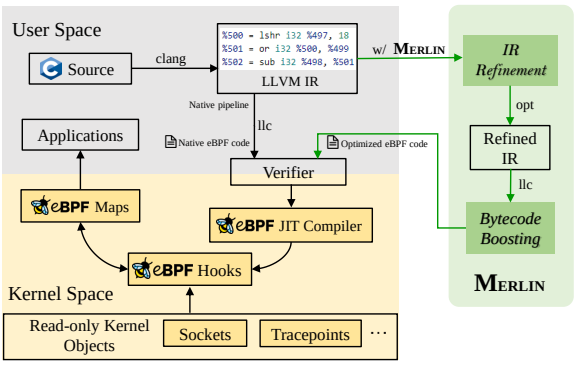


Figure 1. eBPF Workflow

In summary, we make the following contributions:

- We introduces novel optimizations by customizing IR for eBPF and making eBPF instructions efficient.
- We implement our prototype, Merlin, which showcases the practical application of our research.
- We evaluate Merlin with network programs and security applications, illustrating effectiveness and efficiency compared to existing works.

2 Background and Motivations

2.1 eBPF

Extended Berkeley Packet Filter, or eBPF, is a kernel extension that allows observing kernel objects. The eBPF user space (in gray background) and kernel space (in yellow background) are shown in Fig 1. Users can write eBPF programs with high level programming languages (e.g., C), and compile them via the compiler (e.g., LLVM – Low Level Virtual Machine) tool chain into eBPF bytecode (via `clang` and `llc`). Lastly, users can load the program with the `bpf()` syscall, which will first invoke a verifier to check if the program satisfies designed constraints by static analysis and verification techniques. eBPF kernel space is a virtual machine that has an independent architecture and offers the flexibility of direct interpretation or translation into native code through a JIT (Just-In-Time) compiler. If the program is safe, the compiler will compile/interpret it and then attach it to desired places via eBPF hooks, e.g., tracepoints in the kernel or network devices. Collected information will be forwarded to user space applications via eBPF maps.

Fig 2 shows an example including the user source code written in C (up), corresponding LLVM IR (middle), and eBPF

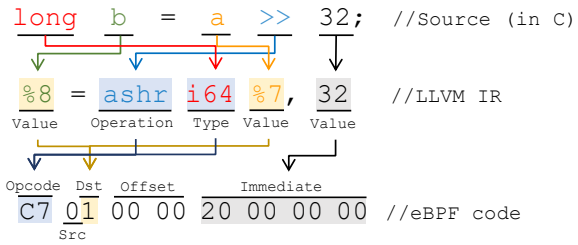


Figure 2. Example on Compiling C to eBPF Code

bytecode (bottom). The source code performs a simple shift, while the corresponding LLVM IR and eBPF bytecode use their own operations and opcode to represent these operators. We mark the corresponding variable and operators in the same font color (from C to LLVM IR) or background color (LLVM IR to eBPF bytecode). In particular, the LLVM IR consists of operators, variables including their types, as well as values (e.g., concrete numbers). It uses the format `%X` where `X` is a numerical number to represent individual variables. LLVM IR is using static single assignment (SSA), and thus, every variable can be assigned only once. A single eBPF instruction consists of 8 bytes, where the first byte is opcode, indicating what operation to do. The second byte represents the source and the target registers. The third and the fourth bytes show the offsets, and the last 4 bytes contain a 32-bit immediate number. In the whole eBPF pipeline, compilers like `clang` compile the source code into LLVM IR, and `llc` translates the LLVM IR into eBPF bytecode.

2.2 Related Works

eBPF Applications. Since eBPF can provide a good performance on data planes, and has high flexibility, it has been widely applied in performance-critical network applications [43, 48, 56, 58–60]. For example, Xhonneux et al. [56] demonstrate the possibility of eBPF for custom network functionalities. After that, Miano et al. [43] emulates iptable semantics with eBPF, providing a noticeable performance boost compared to current iptables. For more complex scenarios, Polycube [46] provides the framework to create arbitrary complex network functions inside the kernel. Besides these applications, hXDP [17] accelerates packet processing on FPGA NICs with eBPF-based software. Electrode [60] then improves distributed protocols by implementing userspace functions in kernel space with eBPF.

Besides network-related tasks, eBPF has also been used in security tasks because it can provide guarantees for program safety [18, 26, 32, 39, 50, 52, 53]. For example, eAudit [50] and saBPF [39] apply eBPF in provenance systems, enabling more secure and scalable data provenance. Bpfbox [26] also proves that eBPF-based confinement is more flexible and provides more access control options than previous systems. Jia et al. [32] allow users to install their own advanced filters by creating a new Seccomp eBPF program type.

Domain-specific Optimization. Although eBPF has been widely adapted in existing systems, the runtime overhead of eBPF is high, and the optimizations on eBPF have not been investigated well yet. Researchers have developed their own techniques to help further improve the performance and availability of eBPF programs. eAudit [50] introduces an efficient encoding method to reduce the cost of user-kernel communication. Miano et al. [44] proposed expanding the bounded loop with LLVM IR to avoid rejection from the verifier. After that, Miano et al. [45] proposed high-level optimization for traffic control, including hash functions, random bits generations, and methods to reduce map lookups. Although these optimizations perform well in specific tasks, they are domain-specific and can not be generalized to all eBPF programs, thus offering limited optimizations.

Code Optimizations. Code optimization is a common technique to improve the performance of the program [19, 21, 25, 28, 37, 38, 40, 51]. Compared with domain-specific optimizations, code optimizations usually can be applied to different programs in certain language. For example, there are general code optimizations like constant propagation [55] and dead code elimination [35] that does the optimizations at multiple levels, e.g. profiling at high level (source code level) [51], or link-time optimization at low level (bytecode level) [22]. However, such architecture-specific optimizations take a lot of places and are not practical for general programs. Also, following this line of work, the eBPF optimizations for security-related tasks are rarely investigated. Zachary H. Jones [33] and Scholz et al. [49] analyze the performance of a subset of eBPF programs, but no optimizations are proposed. K2 [57] formalizes eBPF programs and searches for more efficient substitutions at the bytecode level but suffers from robustness and time efficiency. Not to mention, it only supports XDP programs. Compared with existing systems, our system MERLIN aims to build a multi-tier solution. By applying optimizations at multiple levels, MERLIN can produce stable, fast, and safe results.

2.3 Motivation

Optimizing eBPF programs is challenging. Due to its critical role in the system, eBPF programs frequently interact with core system subjects and objects. To ensure the safety of the kernel, eBPF programs have to satisfy specific safety constraints, posing extra complexity for developers. In particular, eBPF leverages a kernel verifier to carefully check several conditions of the eBPF programs, including but not limited to program termination, usage of loops, program length, and memory access. Fig 3 gives an example to show the uniqueness of eBPF. The two code pieces are semantically equivalent, i.e., zeroing values in the given addresses. However, the left-hand-side code can be accepted by the verifier while the right-hand-side one will be rejected. Because of the verifier, the goal of optimizing eBPF programs

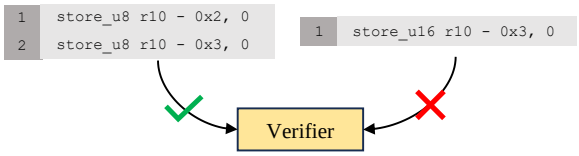


Figure 3. Example on Failed Optimizations

also includes reducing the size of the program (i.e., compactness) besides performance due to the length constraint. Since kernel 5.2, the limit is 1 million instructions, which is not sufficient for large programs [7]. Developers must navigate these constraints while optimizing eBPF programs, balancing performance, size, and adherence to the verifier's safety checks. The unique nature of eBPF necessitates careful consideration and specialized optimization techniques.

Existing compilers, e.g., LLVM – the most commonly used one for eBPF development, can only perform basic and naive optimizations, missing many opportunities. State-of-the-art eBPF optimization method, K2 [57], leverages a machine learning method. It leverages program synthesis to produce optimized code and manually written models to check program safety. As such, it requires non-trivial manual efforts to write, maintain, and update these models corresponding to the eBPF implementation. Moreover, program synthesis techniques tend to be time consuming. As shown in [57], optimizing the xdp-balancer program that has 1771 instructions with K2 takes two days, and the time grows exponentially with the growth of program size.

In this paper, we tackle the program of optimizing eBPF program by starting from analyzing its design documentation [10]. The document has listed general principles that eBPF programs should follow (and any reasonable eBPF verifier should check), which makes it a reliable source for designing eBPF optimizations. We carefully select and customize a set of optimization methods that satisfy the principle of eBPF design, with the goal of better performance and compactness. To be clear, these optimizations (or variants) have been proposed in prior work and our contribution is identifying proper ones and customizing them in the eBPF tool chain.

3 System Design

3.1 Overview

In this paper, we propose an eBPF optimization framework Merlin, which uses a customized LLVM infrastructure with new optimization techniques to emit more efficient eBPF program. A schematic overview of the Merlin pipeline can be gleaned from Fig 1. Our approach seamlessly integrates hybrid optimization techniques during the compilation phase. When the LLVM IR is derived from source code, the IR first undergoes optimizations by `clang` and then Merlin IR optimizers, where we introduce additional optimizers, i.e., LLVM passes. Once the optimized IR is generated, the original `llc` will compile it to eBPF bytecode as usual. But before it is loaded via `bpf()` syscall, Merlin applies another level of optimization, which is the bytecode refinement to optimize the program. Merlin is applied before the verifier and does not change the kernel verifier or other components.

```
1 - b7 01 00 00 01 00 00 00    // movq $1, %r1
2 - 7b 1a c0 ff 00 00 00 00    // movq %r1, -0x40(%r10)
1 + 7a 0a c0 ff 01 00 00 00    // movq $1, -0x40(%r10)
```

Figure 4. Constant Propagation and Dead Code Elimination

3.2 Optimizations

eBPF is a register-based virtual machine with ten 64-bit registers. For safety and design purposes, it has several constraints on how it manipulates the data, e.g., variables should be aligned, and higher bits need to be zeroed out upon shifting. These details are abstracted from users, and it is not obvious for general compilers, e.g., LLVM, to identify optimizations that are specific to eBPF. To evaluate the potential redundancy of instructions caused by data manipulation constraints, we designed two types of optimizations categories:

Category I: Instruction merging is one optimization technique that combines multiple instructions into fewer instructions (§ 3.3).

Category II: Strength reduction substitutes expensive operations with computationally cheaper ones (§ 3.4).

3.3 Category I: Instruction Merging

Instruction merging is a program optimization technique aimed at improving code efficiency by reducing redundancy in instruction sequences. The basic idea involves identifying and consolidating similar or identical instructions within a program to create more compact and streamlined code. This process seeks to eliminate unnecessary computations and redundancies, subsequently enhancing the overall performance of the program. By merging instructions that perform equivalent operations, the compiler reduces the number of executed instructions, minimizing both runtime overhead and memory consumption. Instruction merging is particularly advantageous in scenarios where repetitive sequences of operations exist, as it enables the compiler to generate more concise and optimized code, ultimately leading to improved program speed and resource utilization. eBPF programs have a lot of redundant instructions, and we mainly use two optimization strategies to handle them.

Optimization 1: Constant propagation (CP) and dead code elimination (DCE). This optimization involves two optimization methods that typicall work together. Constant propagation replaces usages of variables with constant values if the compiler can determine the variable's value is constant. As a register-based virtual machine, eBPF operates on registers and LLVM loads every variable and value into a register before operating on them. This guarantees that all variables are correctly loaded, but it unnecessarily translates every `store` instruction of immediate values by first loading

```
- 62 0a fc ff 00 00 00 00    // movl $0, -0x4(%r10)
- 62 0a f8 ff 01 00 00 00    // movl $1, -0x8(%r10)
+ 7a 0a f8 ff 01 00 00 00    // movq $1, -0x8(%r10)
```

Figure 5. Superword Level Merging

them into a register. CP identifies such redundant operations, and DCE removes these unnecessary `load` instructions.

As demonstrated in Fig 4, the original code snippet intends to store the immediate value 1 to memory address `r10 - 0x40`. It first loads 1 into `r1`, and then, stores `r1` to the address `r10 - 0x40`. The optimization identifies that the value in the `store` instruction is a constant 1 and replaces the register `r1` with the value 1. Then, the DCE removes the `load` instruction. As such, the optimized eBPF code, given in Fig 4, directly stores the immediate value 1 to the destination address, reducing one line of code. This also translates into saved CPU cycles due to the removal of the unnecessary `load` operation. The verifier also benefits from it. Although the two programs have the same final states in the verifier, the optimized one directly update the memory value and does not need the record the status of `r1`.

These two are well-known program optimization techniques, which can be implemented by using the control flow graphs and the `IN` and `OUT` sets. We reuse existing algorithms to implement them. □

Optimization 2: Superword level merging (SLM) optimizes identifies vectors of independent instructions operating on adjacent data and converts them into fewer instructions. For instance, the two separate operations in Fig 5 in red respectively stores `u32 0` and `u32 1` to stack `r10 - 0x4` and `r10 - 0x8`. However, this pair of operations can be combined into a single action: storing 1 as `u64` value to `r10 - 0x8`. When doing so, the 8-byte memory region is effectively initialized to the same value, replicating the result of two distinct store operations. In this case, the merged code is 2x smaller than original code. It is particularly advantageous when combining smaller data types like `u8` to a large `u64`). Similar to optimization 1, both programs have the same verifier states, but after optimization the verifier updates the states faster using fewer steps.

Such optimizations can be identified by monitoring neighboring addresses. When operators on adjacent addresses fall into certain types, with the most represented example being `mov`, we can merge them with fewer instructions. In many cases, this is due to the fact that eBPF uses 64-bit registers. □

3.4 Category II: Strength Reduction

Strength reduction is a fundamental optimization technique employed in compilers to enhance the efficiency of programs by substituting expensive operations with less resource-intensive equivalents. The primary objective is to replace computationally expensive expressions with simpler, faster alternatives. This often involves transforming high-cost operations, such as multiplication or exponentiation, into less

```
- %525 = load i16, i16* %110, align 1
+ %525 = load i16, i16* %110, align 2
```

(a) IR

```
- 71 02 25 00 00 00 00 00 // movzbl 0x25(%r0), %r2
- 67 02 00 00 08 00 00 00 // shll $0x8, %r2
- 71 01 24 00 00 00 00 00 // movzbl 0x24(%r0), %r1
- 4f 21 00 00 00 00 00 00 // orl %r2, %r1
+ 61 01 24 00 00 00 00 00 // movzwl 0x24(%r0), %r1
```

(b) Optimized IR

Figure 6. Data Alignment Optimization

resource-demanding ones, like addition or bit-shifting. By identifying opportunities for strength reduction, compilers can exploit the mathematical properties of operations to generate code that is both more concise and executes more rapidly. This optimization is particularly beneficial in performance-critical sections of code, leading to improved runtime efficiency and reduced computational overhead. Strength reduction underscores the importance of leveraging arithmetic equivalences to streamline code execution and maximize the efficiency of computational resources.

Optimization 3: Data alignment optimization aligns data to match the word size of the processor, allowing accessing the data in a single instruction. Unaligned access may require multiple instructions to assemble a word from parts. As such, instructions with proper alignments can significantly reduce the computing cost.

Consider the IR depicted in Fig 6a as an example. The original IR (in red) tends to load an `i16` from pointer `%110`. It translates to loading a 16-bit unsigned integer (`u16`) from the address `r0 + 0x24`. This operation by default is specified with `align 1`, namely, the address can only be byte-aligned. Consequently, the `u16` load operation is decomposed into a sequence of 4 eBPF operations, as shown in Fig 6b: ① The first operation loads the lower 8 bits of the target 16-bit value from the original address (`r0 + 0x24`) and stored in the register `r1`. ② The second phase retrieves the upper 8 bits of the target value by loading another `u8` from the address offset by one byte (`r0 + 0x25`) and storing this value in `r2`. And ③ and ④ concatenate `r1` and `r2` to form the complete 16-bit target value. The third operation shifts `r2` left by 8 bits, correctly positioning this byte. Subsequently, line 4 employs a bitwise `or` operation to merge the lower byte (held in `r1`) into `r2`. This results in `r2` holding the fully assembled 16-bit target value after completing these operations. A properly adjusted alignment can significantly speedup the loading. As shown in Fig 6a with green, our optimization change the alignment to be `align 2`, which allows loading the value with only one `load_u16` instruction.

In this case, although `align 1` and `align 2` yield the same end result, the former forces the compiler to load the value byte-by-byte and leads to the unnecessary assembling

```
- %131 = load i64, ptr %128, align 8
- %132 = add i64 %131, %130
- store i64 %132, ptr %128,  align 8
+ %132 = atomicrmw add ptr %128, i64 %130
          monotonic, align 8
```

(a) IR

```
- 79 02 00 00 00 00 00 00  // movq 0x0(%r0), %r2
- 0f 12 00 00 00 00 00 00  // addq %r1, %r2
- 7b 20 00 00 00 00 00 00  // movq %r2, 0x0(%r0)
+ db 10 00 00 00 00 00 00  // xaddq %r1, 0x0(%r0)
```

(b) Bytecode

Figure 7. Macro-op Fusion (read-modify-write example)

process, which is non-optimal. This increases the total execution time costs and intensify cache pressure due to more frequent memory accesses. This optimization helps reduce verification time. The verifier need to update the states of `r2`, shifted `r2`, `r1` and `r4` separately for the original one, while simply updating `r4` in the optimized one.

For this optimization, Merlin calculates the offset of every pointer to infer and adjust the maximum possible alignment for memory instructions. The optimization in the aforementioned example resulted in 4x smaller code and performance improvement, which can be more substantial when dealing with longer types, e.g., `u64`. □

4 Implementation

4.1 IR Refinement

As shown in Fig 1, Merlin consists of a IR refinement component to optimize LLVM IR code, which is the typical procedure of optimizing eBPF code. Optimizations at the IR level are implemented as LLVM passes compiled into shared object (`.so`) libraries and dynamically loaded by LLVM `opt` during optimization phase.

Optimization 4 (Category I): Macro-op fusion refers to the optimization that combines multiple dependent instructions into a single macro-instruction. We use the RMW (Read-Modify-Write) consolidation as an example, which identifies a set of read, modify, and write instructions, which operate on the same address, into a single AtomicRMW instruction. Fig 7 demonstrates the concept of RMW optimization. The red parts of Fig 7a and Fig 7b respectively show the original IR and eBPF code, consisting of three steps:

load This instruction loads from the memory located at `r10` with an offset of `0x10`. The 1st line of eBPF code in Fig 7b corresponds to this operation and stores the loaded value in register `r2`.

add The instruction adds `r1` to loaded variable, i.e., increasing `r2` by `r1`, as seen in the 2nd line of Fig 7b.

store The instruction stores the result back at the original address. In Fig 7b line 3, the updated value of `r2` is written back to the address `r10 + 0x10`.

Fig 7a illustrate our optimization with green color. We employ a singular `atomicrmw` instruction, targeting the same pointer and value. This optimization efficiently consolidates the original IR's processes – reading, adding, and then writing back to `ptr %128` – into a single instruction. Correspondingly, the resultant eBPF code, displayed in Fig 7b in green, also undergoes similar transformation. Notice that here, we use the adding variables as an example, and the value can be immediate value as needed. Subsequently, our optimization introduces the `xadd` operation to execute the combined read-add-store function. Similarly, the operation is not limited to `add` but also includes instructions that like `or`, `and`, or `xor`.

This optimization will reduce the eBPF code by two lines for each applicable read-modify-write IR set. In verifier, the atomic operation has the same state as the original `store` operation. Thus, the optimization saves the verification time of `load` and `add` operations. It not only simplifies the code but also decreases the frequency of separate memory accesses, consequently lessen cache misses. The process also ensures locked atomic operations, guaranteeing atomicity. Note that while atomic operations typically introduce additional latency, many modern processors has dedicated hardware support[23, 29], minimizing this delay. Indeed, the latency of these operations in recent models is very close to that of non-locked operations[27], that it generally does not pose a significant concern.

Our implementation scans sequences of IR instructions looking for compatible instructions to fuse, e.g., load-operate pairs, and small sequences of arithmetic/logic operators. Merlin replaces a sequence of fusible instructions with fused ones that encodes the same semantics. □

4.2 Bytecode Refinement

Merlin has a component of bytecode refinement, which operates on the bytecode level to optimize the program. This is because certain eBPF instructions cannot be directly generated from LLVM IR, implying inherent constraints in the scope of optimizations that can be applied at the IR level. This limitation is significant as it points out the boundaries of IR refinement, especially when targeting the performance-critical eBPF environment. Our bytecode refinement in Merlin is designed to resolve this issue. There are two typical scenarios that we need bytecode level optimizations. One is that existing compilers do not support the use of some instructions, e.g. ALU-32 operations (Example 5). Although ALU-32 operations can be enabled by *mattr=+alu32* option, the verifier may encounter difficulties in accurately tracking these operations in kernel versions prior to 5.13, as noted in [3]. Furthermore, one of the applications we analyzed was found to be rejected with ALU-32 operations in kernel version 5.15, yet functioned correctly when this feature was disabled. Moreover, since there are still servers using kernel version < 5.0, which does not support v3 instructions, these optimizations aim to provide key

```
- 67 00 00 00 20 00 00 00  // shlq $0x20, %r0
- 77 00 00 00 20 00 00 00  // shrq $0x20, %r0
+ bc 00 00 00 00 00 00 00  // movl %r0, %r0
```

Figure 8. Code Compaction with `movl`

improvements introduced in v3 instructions. And the other one is that the optimization pattern is obvious in bytecode level but far more complex in the IR level (Example 6).

Optimization 5 (Category I & II): Code compaction with unsupported instructions. In this example, the program operates on a `u32` variable. Notice that all eBPF registers are 64-bit and eBPF is a register-based virtual machine. Thus, the program has instructions to prepare a `u32` value from the 64-bit register. The initial bytecode, given in Fig 8 in red, showing the LLVM generated code for this process, where `r3` is first shifted left by 32 bits, then shifted right by the same amount to clear the upper half of the register.

Our optimization directly uses the 32-bit ALU operation `movl` in Fig 8. This operation efficiently transfers the lower half of one register to the target register, achieving the same outcome as the shifting method but with greater conciseness. `movl` is semantically equivalent but more compact, achieving 50%-100% latency reduction and 4-6x throughput in modern CPUs [27]. Similar to previous optimizations, the verifier can skip the status update of `shlq` and reach to the final state, saving verification time.

This optimization is known as code compaction, which looks for opportunities to use smaller instruction encodings where possible. This optimization cannot be deployed at IR level because no LLVM IR instruction directly translates to `movl`. Thus, we implement this optimization in the bytecode level. As popular types in existing programming language have 32-bit length, extracting lower half of the register is a common operation, making this optimization useful. □

Optimization 6 (Category I & II): Peephole optimizations in bytecode. In some cases, the bytecode shows easy-to-optimize code patterns, while modifying the corresponding LLVM IR is significantly hard. In this case, we use a peephole optimization as an example. eBPF only provides 64-bit registers. Compilers have to generate code that masks to unnecessary bits for smaller-sized values like `i32` to guarantee safety of arithmetic operations. In many cases, the general masking is verbose.

In Fig 9a, we show the original IR for logically shifting a 32-bit integer (i.e., `i32`) by 28 bits. It takes two instructions for the eBPF code to load a 64-bit long immediate value into the register `r3`. The loaded values are masks that will clear the upper half of the register and bits that will be removed by the shift operation. Subsequently, the next operation applies this mask to the value register `r8` (instruction 3), after which, `r8` is safe for the 28-bit right shift performed in the final step. The optimized code is depicted in Fig 9b. Line 1 shifts the value in `r8` left by 32 bits, and line 2 shifts it right by 60 bits.

```
- %855 = lshr i32 %850, 28
+ %851 = zext i32 %850 to i64
+ %852 = shl i64 %851, 32
+ %853 = lshr i64 %852, 60
+ %854 = trunc i64 %853 to i32
```

(a) IR

```
- 18 03 00 00 00 00 00 f0 // movq $0xf0000000, %r3
- 00 00 00 00 00 00 00 00 //(takes two intructions)
- 5f 38 00 00 00 00 00 00 // andq %r3, %r8
- 77 08 00 00 1c 00 00 00 // shrq $28, %r8
+ 67 08 00 00 20 00 00 00 // shlq $32, %r8
+ 77 08 00 00 3c 00 00 00 // shrq $60, %r8
```

(b) Bytecode

Figure 9. Peephole Optimization

Masking and left shifting both address the issue effectively while our method further speeds up the process by eliminating the need to load a mask. Note that it will save two lines of code instead of one due to the fact that load 64-bit immediate values require twice the instruction length in eBPF architecture. Additionally, by dispensing with the mask, we free up a register, opening up opportunities for further optimizations. It is also worth noting that shift operations are frequently used in hash functions in form of ror and rol. Applying such optimization can greatly streamline hashing functions, which is useful in network applications.

This optimizations is straightforward in the bytecode level. However, the corresponding IR level code is shown in Fig 9a. It extends the original one IR instruction to four. The first one converts the original value to an 64-bit integer `i64` using the zero-extension instruction (`zext`), followed by shifting the extended value left by 32 bits to discard any unnecessary bits, and then, shifting right by 60 bits. Lastly, the value is truncated back to the intended 32-bit integer `i32` type with the `trunc` instruction. This is because LLVM IR knows that the original variable `%850` is 32-bit and the original version does not need to do any type casting. Until the code generation phase, the compiler leverages the information that eBPF only supports 64-bit registers to generate bytecode. Modifying the code generation logic requires significant efforts including the understanding of eBPF (e.g., ISA) and details of compiler (e.g., register allocation). As such, Merlin implements this in the bytecode level. □

5 Evaluation

We evaluate the performance of Merlin by answering the following research questions (RQs):

- **RQ1:** How much can Merlin improve in terms of code compactness and verification costs? (§ 5.2)
- **RQ2:** What are the improvements Merlin made to network programs in terms of throughput and latency? (§ 5.3)
- **RQ3:** How much can Merlin reduce the runtime overhead of other domain-specific systems (e.g., sysdig)? (§ 5.4)

- **RQ4:** What are the additional compilation costs of Merlin and its individual optimizers compared to the original compilation chain? (§ 5.5)
- **RQ5:** What are the impacts of individual optimizer in Merlin? (§ 5.6)

5.1 Evaluation Setup

Hardward and Software. Our prototype is built upon Python 3.8 and LLVM 17. All eBPF programs are jitted. We compile them with clang and flag O2. We use $mcpu = v3$ for support of 32-bit instructions if the compiled program is accepted by the verifier. Otherwise, we use $mcpu = v2$. All of the programs used for experiments pass the verification from kernel 4.15 to 6.5. The presented results are consistent across kernel versions unless specified. Code compactness, throughput, and latency are tested on two xl170 nodes of CloudLab [24]. On cloudlab, we were able to retrieve the image and config shared by Xu et al. [57]. Thus, the configuration is identical to the settings of K2 and leads to a more straightforward result. The nodes are also equipped with 10-core Intel E5-2640v4 processors with PCIe 3.0 bus, 64 GB of memory, and Mellanox ConnectX-4 adapters and are connected to a high-speed 25G network. Other experiments (i.e., runtime and compilation overhead) are conducted on a Ubuntu 22.04 (kernel version 5.15) bare metal machine equipped with a Ryzen 8 cores 6800h CPU and 16 GB DDR5 4800 single stick memory. For best precision, we collect instructions for micro benchmarks with Intel PT [29] with the same system but on an Intel 7700K CPU and 16GB DDR4 3200 dual stick memory.

Benchmark. We evaluate Merlin on 19 XDP programs from the Linux kernel, Meta, hXDP [17] and Cilium, and three complex systems, i.e., Sysdig [11], Tetragon [13], and Tracee [14]). We show the details of benchmarks in Table 1. The first column shows the type of programs and the second column presents the number of programs of its kind. The largest, average, and smallest sizes of programs are shown in the third column. Besides these, we show the mcpu version we used for each kind of programs in the last column.

Metrics. We evaluate Merlin and existing systems by measuring their improvements on code compactness, throughput and latency (for network programs), general runtime overhead (for other domain-specific systems), and additional compilation costs. These metrics are widely used in existing systems[30, 41, 43, 47, 48, 57] and we denote them below.

- *Compactness.* We use the reduction ratio of the Number of Instructions (NI) to measure the compactness of programs (i.e., the size of the programs). Specifically, the NI is defined as the number of eBPF instructions of a section associated with a function symbol. Since each eBPF instruction takes 8 bytes, NI is calculated as Size In Bytes over 8.
- *Verification Cost.* We use the Number of Processed Instructions (NPI) to measure the change of instructions processed inside verifier. Note that NPI is always larger then NI because branches lead to multiple verifications per instruction. The NPI and other verification properties (e.g., time costs) are reported by kernel verifier and collected via $bpf()$ syscall with *log_level=4*. There are 7 local functions [9] that cannot be verified alone. They are verified within the main function.

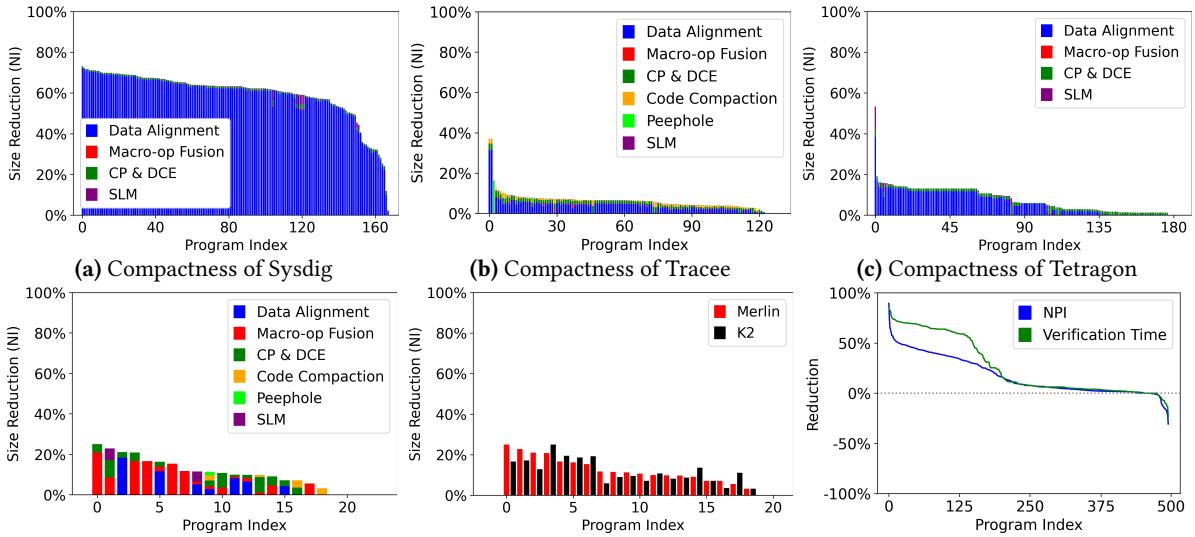
Table 1. Details of Benchmarks

	Number of Programs	Program Size (NI)			mcpu
		Largest	Smallest	Average	
XDP	19	1771	18	141	v2
Sysdig	168	33765	180	1094	v3
Tetragon	186	15673	21	3405	v3
Tracee	129	16633	29	2654	v2

- *Throughput and Latency.* We measure the throughput and latency of programs to evaluate the performance of Merlin on network-related tasks. The throughput is reported as the maximum loss-free forwarding rate (MLFFR [16]) of a single core, in millions of packets per second (Mpps) at 64-byte packet size. We setup two PCs (i.e. PC1 and PC2) to measure the performance. On PC1, T-Rex[20] traffic generator generates the traffic. PC2 is the device under test (DUT). The traffic moves from PC1 to PC2, and loops back to PC1. The latency is calculated as the time spent to go over the loop. We report the latency results under different workloads: ① **low**: lower than the throughput under an unoptimized pipeline; ② **medium**: equals to the throughput under an unoptimized pipeline; ③ **high**: equals to the throughput with the best-found program; ④ **saturate**: higher than 'high' setting.
- *Runtime Overhead.* Runtime overhead refers to the additional execution time used by the optimized programs compared to the original systems. More specifically, we denote the reduction of such costs as outlined in Equation 1, where t_v is the execution time of the program running alone, $t_{w/o}$ is the execution time with the security system active in the background, and $t_{w/}$ is the execution time with the Merlin-optimized system running in the background. We test such time costs and reduction with micro-tests and macro-tests. We use lmbench[42] for micro-tests and postmark[34] for macro-tests. Lmbench offers a wide variety of low-level tests, such as installing signal handlers and executing basic system calls. Postmark simulates the typical file system workload of web and mail servers to provide insight into performance under real-world scenarios. We run each test for ten trials and then report the average results. Sysdig, Tracee, and Tetragon are invoked with default settings, and the outputs are redirected to /dev/null. Runtime overhead is important for frequently active as well as long-running systems, and a good optimization system should have a low runtime overhead because a high runtime overhead can lead to an unacceptable accumulation of operational costs.

$$OverheadReduction = 1 - (\frac{t_{w/}}{t_v} - 1)/(\frac{t_{w/o}}{t_v} - 1) \quad (1)$$

- *Other Performance Metrics.* We use *perf* to collect hardware performance counters to show the improvements of Merlin



(a) Compactness of Sysdig **(b)** Compactness of Tracee **(c)** Compactness of Tetragon

(d) Compactness of XDP Programs **(e)** Compactness Comparison **(f)** Impact on Verifier

Figure 10. Compactness and Verifier Stats of All Programs. 7 programs are program-local functions.

Table 2. Limitation of K2 and MERLIN. Size* is NI of program that can be optimized in reasonable time (e.g. < 2 days).

	Instructions Set	Helper Functions	Maps	Size*
K2	v2	XDP only	Limited	<2000
MERLIN	-	-	-	1 Million

on hardware devices. For network applications, we report context switches, cache misses and branch misses of DUT in 5 seconds under low and saturate workloads. For security applications, we report instructions, CPU cycles, cache misses and branch misses of each single test. Specifically, instructions are collected with Intel PT for micro benchmarks, and general PMU event for macro benchmarks. To show the statistics uniformly, context switches are reported in percentage of clang version. Instructions and CPU cycles are calculated as percentage of original program.

- *Compilation Cost.* The compilation cost is presented as the additional time spent by MERLIN to finish the program compilation compared to the compilation without any optimization. It is worth mentioning that we usually only need to pay the compilation costs once. Compared with runtime overhead, systems are less sensitive to compilation costs.

5.2 RQ1: Compactness and Verification Cost

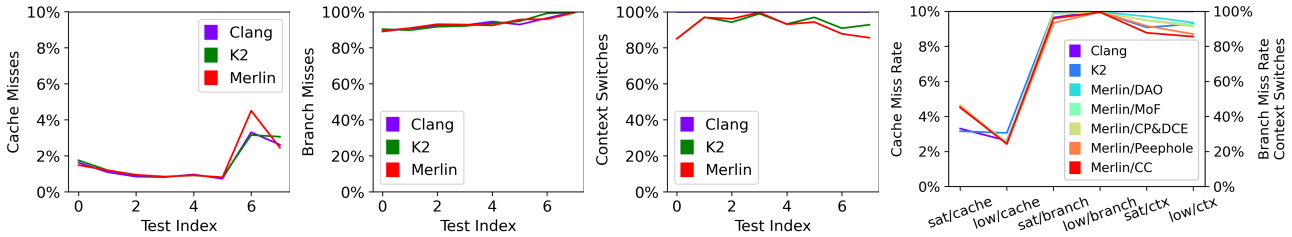
Compactness. The results of code compactness are reported in Fig 10a to Fig 10e. The x-axis of all figures shows the program indexes, and the y-axis denotes the reduction ratio of NI. Specifically, the black bars in Fig 10e show the results of the state-of-the-art system K2. Other different colored bars represent the contribution of different optimizers. Notice that K2 cannot provide optimization for other programs except XDP programs, as K2 did not formalize all eBPF helper functions, as shown in Table 2. Therefore, we compare MERLIN with K2 only on XDP programs.

We first find that all optimized programs can pass the verification. Overall, MERLIN can improve the code compactness of different programs (Fig 10). Across all XDP programs (Fig 10d), we notice that MERLIN can reduce the NI by up to 22.22%. On other systems (Fig 10a to Fig 10c), the NI can be reduced even more (i.e., a maximum of 73.08%; on average, 59.81%, 7.48% and 6.20% respectively for Sysdig, Tetragon, and Tracee). The result demonstrates the benefits and generalization of MERLIN. MERLIN is effective and can achieve good results on different programs because MERLIN offers comprehensive optimization from high-level observations to significantly reduce the redundancy of code and such observations can generalize to different program cases.

We find MERLIN produces more compact code than K2. Specifically, MERLIN outperforms K2 in 10 programs out of 19. Although MERLIN cannot outperform K2 in some cases, it is understandable because our implementation performs general optimization while the optimization of K2 is application-specific. Some complex optimizations that have been used by K2 for XDP programs are not integrated in MERLIN by default, but can be compatible with MERLIN to further improve the performance. Therefore, the results still prove the advantages of MERLIN compared with existing systems. We also would like to emphasize that the effectiveness of MERLIN is particularly evident in larger programs by our design. For instance, on the largest XDP program, xdp-balancer, our improvement is 26.2% higher than that of K2 , and the average difference on smaller programs is only 0.26%. The reason is that the time costs of K2 can increase exponentially with the size of the program, and K2 stops searching before getting the optimal program. Considering real-world programs usually have a large size, we believe MERLIN can be more scalable and practical than existing systems in the real scenarios.

Table 3. Throughput and Latency

	Throughput (Mpps)			Latency (ms)															
				Low				Medium				High				Saturate			
	clang	k2	Merlin	load	clang	k2	Merlin	load	clang	k2	Merlin	load	clang	k2	Merlin	load	clang	k2	Merlin
xdp2	9.814	10.101	9.940	9	21.080	19.829	20.787	9.81	47.833	20.219	22.568	10.1	89.523	42.958	96.481	10.3	103.872	97.754	98.90
xdp_router_ipv4	1.496	1.496	1.496	1	63.323	59.834	60.032	1.5	84.450	76.929	77.560	1.5	84.450	76.929	77.560	1.8	619.291	610.119	612.018
xdp_fwd	4.984	5.072	5.075	4.4	32.272	30.358	28.957	5	87.291	71.645	49.031	5.075	180.985	172.190	71.786	5.2	192.936	188.199	187.841
xdp-balancer	3.292	3.389	3.409	3	38.650	37.152	34.523	3.3	73.319	55.741	50.407	3.41	220.320	141.434	109.342	3.7	296.405	292.376	291.752



(a) Cache Misses of XDP Programs.

(b) Branch Misses of XDP Programs.

(c) Context Switches of XDP Programs.

(d) Hardware counters stats of xdp-balancer.

Figure 11. Hardware Performance Counter of All XDP Programs

We also find that all optimizers can provide improvements in code compactness. More specifically, data alignment optimization, macro-op fusion, CP & DCE, code compaction, peephole optimization, and SLM can provide 17.85%, 1.44%, 1.59%, 0.61%, 0.13% and 0.17% NI reduction on average respectively. The results show effectiveness of each optimizer. We also find that the specific effectiveness of different optimizers can vary on different programs since they target different code redundancies. For example, data alignment optimization usually provides the most significant improvement because memory access is the most common behavior among most programs, and it is designed to reduce the costs of such memory accessing codes. We can conclude that all optimizers are useful, and their designs are reasonable.

Verification cost. To show the impacts on verification costs, we measure the NPI and time costs of programs in verification. The results are included in Fig 10f. The x-axis is the index of our tested programs. The y-axis shows the reduction ratio of NPI and the verification time costs.

We find that Merlin can reduce both NPI and time costs of verification. Specifically, Merlin can decrease the NPI up to 89.6% and the time costs up to 85.2%. On average, the numbers are 17.1% and 25.4%. The results show that Merlin-optimized programs are efficient during the verification. We want to reiterate that NPI is typically much larger than NI in a program as the verifier walks through every unique path. While the limitation of NPI is set to one million after kernel version 5.2, it still can be easily hit with sufficiently complex programs [4, 7]. During evaluation, we also find all three of security applications have programs that exceed 100000 NPI during verification, and Merlin is capable of reducing NPI (e.g. from 169103 to 147915). The benefits are obvious when the program needs to accommodate more functionality.

5.3 RQ2: Throughput and Latency

We measure the throughput and latency of systems on four XDP programs, which are the only ones that can forward traffic among all programs (the same setting as K2 [57]). We show the results of clang (native compiler), K2 and Merlin in Table 3. The columns show the program names, throughput of each program, and the latency performance under different workloads, which is measured in microseconds.

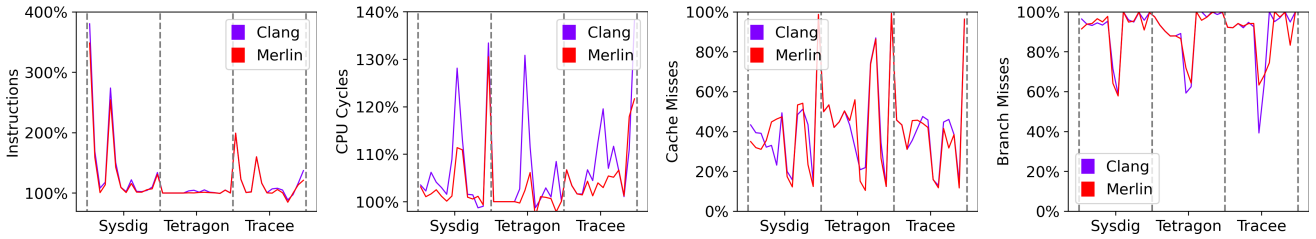
Overall, the results show both Merlin and K2 can effectively improve the throughput compared with clang on small programs (i.e., xdp2, xdp_router_ipv4 and xdp_fwd), and the performance is comparable. Specifically, Merlin can improve the throughput by up to 3.55%, while K2 improves it by up to 2.95%. The results show the effectiveness of Merlin in terms of improving the throughput, and the improvement is comparable with the most advanced existing systems.

For larger programs like xdp-balancer, Merlin provides more improvements on throughput than existing systems. Merlin improves the throughput by 3.55% compared with clang, which is 5.71x larger than the average improvements on small programs. When compared with K2, Merlin gets 0.59% higher throughput (while the difference on small programs are less than 0.2%). Merlin produces programs that have higher throughput on large programs since Merlin has better support for different-sized programs (as explained in § 5.2). Therefore, Merlin is more generalizable compared to size-sensitive random program searching.

When analyzing the latency results, we observe that while K2 already optimized the program latency, Merlin provides more significant improvements. On average Merlin reduces latency by 6.89% with low workload, 29.93% with medium workload, 27.78% with high workload and 2.54% with saturate workload, which produces 10.06% more improvements compared to K2. The results show the efficiency of Merlin in terms of latency reduction. Merlin provide better latency

Table 4. Security Application Benchmarks

Tests		Vanilla	Sysdig			Tetragon			Tracee		
			w/o MERLIN	w/ MERLIN	Overhead Reduction	w/o MERLIN	w/ MERLIN	Overhead Reduction	w/o MERLIN	w/ MERLIN	Overhead Reduction
lmbench (μs)	NULL call	0.06	0.38	0.28	31.25%	0.06	0.06	0%	0.10	0.10	0%
	NULL I/O	0.12	0.60	0.51	18.75%	0.12	0.12	0%	0.17	0.17	0%
	stat	0.36	0.75	0.69	15.38%	0.37	0.37	0%	0.43	0.43	0%
	open/close file	0.79	1.83	1.56	25.96%	0.80	0.80	0%	0.94	0.93	6.67%
	signal install	0.10	0.44	0.34	29.41%	0.10	0.10	0%	0.15	0.15	0%
	signal handle	0.83	1.27	1.17	22.73%	0.83	0.83	0%	0.88	0.87	20%
	fork process	72.87	81.72	81.16	6.33%	96.01	95.45	2.42%	76.09	75.58	15.84%
	exec process	321.53	392.70	353.71	54.78%	333.45	329.53	32.89%	337.39	331.83	35.06%
	shell process	738.76	880.89	823.82	40.15%	758.76	753.81	24.75%	770.36	759.39	34.71%
	file create (0k)	4.78	6.15	6.01	10.22%	4.84	4.82	33.33%	7.31	7.10	8.30%
	file delete (0k)	3.02	3.87	3.77	11.76%	3.12	3.11	10%	5.29	5.28	0.44%
	file create (10k)	9.73	12.74	12.40	11.29%	10.04	9.98	19.35%	13.30	13.21	2.52%
	file delete (10k)	5.00	5.78	5.63	19.23%	5.13	5.09	30.77%	7.36	7.09	11.44%
	AF_UNIX	3.42	6.01	5.68	12.74%	4.03	3.94	14.75%	5.43	5.45	0%
	pipe	5.24	6.59	6.08	37.78%	9.60	7.65	44.72%	9.48	9.65	0%
	Average		23.19%			14.20%			8.67%		
Postmark (s)		58.86	82.54	78.73	16.08%	67.26	64.70	30.44%	59.90	59.27	60%



(a) Instructions of Security Apps **(b)** CPU Cycles of Security Apps **(c)** Cache Stats of Security Apps **(d)** Branch Stats of Security Apps

Figure 12. Hardware Performance Counter of All Security Programs

performance because MERLIN has optimizations that especially benefits hashing and other regular network functions (as mentioned in § 4.2).

Hardware performance counters. Additionally, we measured MERLIN with hardware performance counters, and we show the results in Fig 11a to Fig 11c. The results include the statistics of branch misses, cache misses, and context switches of four XDP programs. The tests are ordered in the same order as Table 3 (i.e. from xdp2 under low workload to xdp-balancer under saturate workload).

For cache stats, we observe that cache misses are consistent before and after optimization for first three programs. For xdp-balancer, we find a noticeable cache miss rate increase. The reason behind this is that the total cache references decreased from 10330 to 8412. The cache miss rate of the same program xdp-balancer under saturate workload also proves this: MERLIN-optimized version provides lowest cache miss rate. For branch misses, we observe that the numbers of branch misses are consistent between three versions of programs. For context switches, MERLIN provides better performance than K2 for xdp2, xdp_fwd and xdp-balancer. MERLIN is especially advantageous on large program like xdp-balancer, in which setting MERLIN lowers the number of context switches to a maximum of 85% whereas K2 can only do 93%. The hardware performance counters suggest that MERLIN hepls to reduce context switches, cache events and cache misses during runtime.

5.4 RQ3: Runtime Overhead.

We measure the additional time costs of running the optimized programs to show the efficiency of MERLIN. The results are shown in Table 4. The first and second columns show the test environments and tested operations. The other columns present the overhead under different systems. Specifically, vanilla indicates the case with no provenance system running. Sysdig, Tetragon or Tracee indicate test cases with the corresponding systems running background. For each kind of system, we report the results of using the original eBPF programs and the results of using MERLIN. We also report the overhead reduction rate of MERLIN compared with the original eBPF settings defined in Equation 1.

From the results, we observe that MERLIN can significantly reduce the runtime overhead compared with the original programs. On average, the overhead is reduced by 8.67% to 23.19% on micro tests and from 16.08% to 60% on macro tests. This result demonstrates the effectiveness of MERLIN regarding reducing the runtime overhead. MERLIN is efficient because existing systems (such as Sysdig) can run in

the background for long periods of time while performing frequent kernel-user data transfers, and the data alignment optimization simplifies such data exchange process.

Hardware performance counters. We also evaluate the performance of Merlin with hardware performance counters for these tests and show the results in Fig 12a to Fig 12d. The results for each security application are listed between two dashed lines, and the order of these tests are the same as listed in Table 4 (i.e. starts from NULL call test, and ends with postmark test. File read and write on 0k/10k are represented by one data point only because of write and read are operated consecutively in lmbench).

On average, we observe that Merlin saves 12.16 instructions / 4.04 CPU cycles in micro benchmarks, and 17019.61 instructions / 17560.84 CPU cycles in macro benchmarks. The results show that Merlin performs considerable improvements on both instructions and CPU cycles. For cache misses and branch misses, we find that for lightweight jobs like Null call or null I/O, the results vary. The reason is that most tests in micro benchmarks lead to a fairly small number of cache events and branch events (e.g. <100). In such scenario, the differences shown in figure is considered insignificant. For macro benchmarks, there are no difference either because there are always near 100% cache misses and branch misses. Therefore, we conclude that Merlin is able to make considerable improvements at runtime due to less instructions and less CPU cycles.

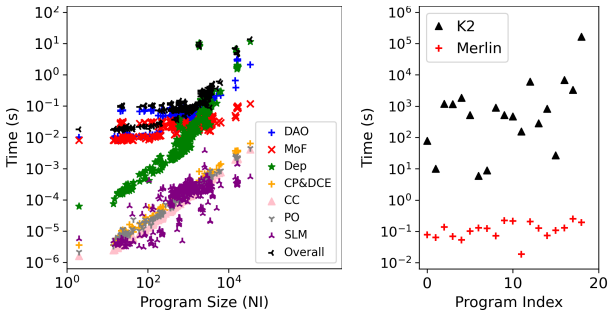
5.5 RQ4:Additional Compilation Costs of Merlin

We measure the additional compilation costs of integrating Merlin with existing programs to answer the question. The results are shown in Fig 13a. The x-axis denotes the NI of the original program, and the y-axis denotes the compilation time cost of introducing certain optimization.

5.5.1 Overall Performance.

For XDP programs, we find that Merlin takes imperceptible time to optimize them, and the compilation costs are lower than those of K2. Specifically, the average cost of Merlin on all XDP programs is only 0.035 seconds, which is trivial compared with the system running time and shows that Merlin is efficient for deployment. Compared with K2, Merlin is 3,201,561x faster on the biggest program and 4937x faster on the smallest programs shown in Fig 13b. On average, Merlin is **six orders of magnitude** faster than K2. The results further show the superior advantages of Merlin compared with existing systems. Not to mention K2 only supports XDP programs.

For observation applications, the compilation costs of Merlin are also not significant. From the results, we find that Merlin only uses 17.224 seconds, 150.557 seconds, and 54.209 seconds on Sysdig, Tetragon, and Tracee. We also observe that the optimization of Tetragon is significantly more time-consuming than that of other systems. The reason is that Tetragon contains variants for different kernel versions,



(a) Time cost of Optimizers. **(b)** Comparison with K2.

Figure 13. Compilation Costs of Merlin. Labels are Data Alignment Optimization (DAO), Macro-op Fusion (MoF), Dependency analysis (Dep), Code Compaction (CC), Peephole Optimization (PO), and Superword Level Merging (SLM).

and those for recent kernels (>=5.3) contain bounded loops or very large programs (NI > 10k), leading to a longer optimization time. For other programs, we find that the time cost grows almost linearly with program size. Merlin is able to keep the time cost of most reasonable-sized program (NI < 10^3) under 0.1s. Even for very large programs (e.g., 33765 NI), Merlin can handle it within 13 seconds. The results further indicate the benefits and superior generalization of Merlin.

5.5.2 Efficiency of Individual Optimizer.

To evaluate the efficiency of individual optimizers, we evaluate the time costs of deploying each optimizer.

We find that the time cost of applying each optimizer is 0.146 seconds on average, which is pretty low considering the normal runtime overhead. Although we consider the highest time cost, the cost is not significant (i.e., 13 seconds for static analysis at bytecode level). The results further prove the efficiency of deploying all of our optimizers. Since the additional compilation of programs using Merlin does not impose a high overhead, Merlin can bring economical enhancements.

For different optimizations, the costs of each component are 0.029 seconds, 0.219 seconds, 0.904 seconds for Macro-op fusion, data alignment optimization, and static analysis, respectively. We find that static analysis is the most time-consuming component. That is because it iterates all possible data flows, including analyzing instruction, expanding loops, and maintaining dependencies, making it especially costly compared to other components. However, the process speeds up other optimizations. As part of the result, the other four optimizations only take less than 0.001 seconds on average.

5.6 RQ5: Case Study

• **Load-Balancer**. We use xdp-balancer to show the impacts of our individual optimizer since it has the most applicable optimizations (i.e., 5 out of 6). Starting with clang-compiled program, we apply optimizations in sequence and record the results after each optimization. The results are shown in Fig 14. The '+' mark shows throughput and lines with different colors show latency under different workloads. The

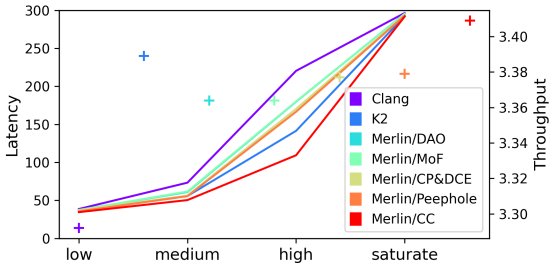


Figure 14. Latency and throughput of xdp-balancer.

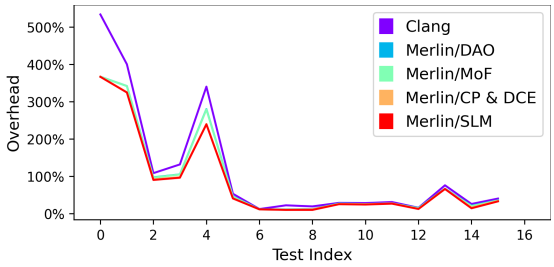


Figure 15. Overhead of Sysdig

stats of hardware counters are shown in Fig 11d. We find that three optimizers contribute to over 90% improvements of performance. Overall, data alignment optimization (DAO) provides 68.2% throughput improvements, 38.0% average latency reduction, 100% cache reference reduction, 100% cache miss reduction and 44.8% context switch reduction. Code compaction provides 21.1% throughput improvements, 45.3% average latency reduction and 9.7% context switch reduction. Peephole optimization accounts for 9.1% throughput improvements and 4.5% average latency reduction and 30.5% context switch reduction. They are efficient because DAO greatly speeds up data transferring for traffic handling. Code compaction and peephole optimization streamlines the frequently used `jhash` functions as discussed in § 4.2.

• **Sysdig** earns great code size improvements and is representative as a domain-specific complex application. We use the same settings as those of Load-Balancer. Fig 15 shows the performance of Sysdig. We find that on average DAO accounts for 72.7% overhead reduction, 97.9% NI reduction, 99.6% verification time reduction and 99.3% NPI reduction. Data alignment optimization is designed dedicatedly for applications that require huge amount of data transferring. During optimization, we find that the average alignment of 18142 memory operations is 3.85, and after optimization it becomes 4.81. Unlike other programs that have multiple versions of kernel probes (e.g. Tetragon designs probes for kernel 5.x and 6.x), the bpf programs in Sysdig have to fit in different environments. Due to the number of system calls that Sysdig needs to support, it is difficult for them to promise alignment at source code level. This case further explains the superiority of MERLIN of having multi-tier optimizations, which can address this problem easily.

6 Discussion

• **Verifier states**. The total and peak states (i.e., the maximal number of states in the verifier at any given time) of the verifier can also reflect the impact of optimization. However, we found them highly dependent on verifier implementations, as shown in Table 5. For different programs and kernel versions, the peak and total state changes (before and after optimization) in different kernel versions can be positive (more states after optimization) or negative. As the development of the eBPF verifier is still an active field [15, 31, 54], we view these two metrics as unstable to use for evaluation.

Table 5. State Change Over Kernel Versions

Program Name	Kernel Version	sys_writev_pwritev_x	execve_family_flags
Peak State Change	5.19	-12.41%	+49.84%
	6.5	+12.27%	-0.36%
Total State Change	5.19	-14.31%	+57.03%
	6.5	+11.88%	-8.72%

This instability also leads to verification time increase for some of the programs.

• **Completeness and robustness of MERLIN**. We acknowledge that due to our limited knowledge, understanding of the eBPF design/principles, and engineering capability, we may have overlooked some techniques that can be adopted in eBPF program optimization, despite our best efforts in adhering to constraints. We try to follow the safety principle to the highest standard to ensure all optimizations in MERLIN can work in the future for any reasonable eBPF verifier implementation. For instance, even though loading 64-bit values and pruning to 48-bit could offer performance improvements, such optimizations were avoided due to unnecessary memory reading. Thus, we believe MERLIN is far from complete but relatively robust, and our results of MERLIN working on kernels from 4.15 to 6.5 demonstrated its capability.

7 Conclusion

This paper explores the potential of customizing IR within the eBPF context and optimizing bytecode for enhanced eBPF performance. Specifically, we integrate such code optimization techniques to improve traditional eBPF programs and propose our solution MERLIN. With ensuring all optimized programs pass the verifier, MERLIN can reduce the NI by 73%, and reduce the general runtime overhead by 60% compared with the original programs. Additionally, improve the throughput by 0.59%, reduce the latency by 5.31% compared to state-of-the-art method K2 , while being 10^6 times faster and scalable to larger and more types of programs. The results show that using our optimizations is promising in improving the performance of eBPF programs.

Acknowledgments

We thank our shepherd David Grove and the anonymous reviewers for their constructive comments. This material is based upon work supported by the IARPA TrojAI W911NF-19-S-0012, NSF 2342250, and NSF 2319944.

References

[1] Bringing ebpf and cilium to google kubernetes engine | google cloud blog. https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine, 2020. Accessed: 2023-10-30.

[2] How alibaba cloud uses cilium for high-performance cloud-native networking. https://cilium.io/blog/2020/10/09/cilium-in-alibaba-cloud/, 2020. Accessed: 2023-10-30.

[3] Nvd - cve-2021-3490. https://nvd.nist.gov/vuln/detail/CVE-2021-3490, 2021. Accessed: 2023-10-30.

[4] c - bpf verifier says program exceeds 1m instruction - stack overflow. https://stackoverflow.com/questions/70841631/bpf-verifier-says-program-exceeds-1m-instruction, 2022. Accessed: 2024-03-04.

[5] Microsoft and isovalent partner to bring next generation ebpf dataplane for cloud-native applications in azure | microsoft azure blog. https://azure.microsoft.com/en-us/blog/microsoft-and-isovalent-partner-to-bring-next-generation-ebpf-dataplane-for-cloudnative-applications-in-azure/, 2022. Accessed: 2023-10-30.

[6] Amazon guardduty eks runtime monitoring expands operating systems and processor support. https://aws.amazon.com/about-aws/whats-new/2023/07/amazon-guardduty-eks-monitoring-systems-processor/, 2023. Accessed: 2023-11-1.

[7] Bpf design q&a — the linux kernel documentation. https://docs.kernel.org/bpf/bpf_design_QA.html#q-what-are-the-verifier-limits, 2023. Accessed: 2024-03-04.

[8] Bpf documentation — the linux kernel documentation. https://docs.kernel.org/bpf/, 2023. Accessed: 2023-10-30.

[9] ebpf instruction set specification, v1.0 — the linux kernel documentation. https://docs.kernel.org/bpf/standardization/instruction-set.html, 2023. Accessed: 2023-11-06.

[10] ebpf verifier - the linux kernel documentation. https://docs.kernel.org/bpf/verifier.html, 2023. Accessed: 2023-11-06.

[11] Sysdig | security for containers, kubernetes, and cloud. https://sysdig.com/, 2023. Accessed: 2023-10-30.

[12] Tencent cloud mesh | tencent cloud. https://www.tencentcloud.com/products/tcm, 2023. Accessed: 2023-10-30.

[13] Tetragon - ebpf-based security observability and runtime enforcement. https://tetragon.io/, 2023. Accessed: 2023-10-30.

[14] Tracee - aqua. https://www.aquasec.com/products/tracee/, 2023. Accessed: 2023-10-30.

[15] Sanjit Bhat and Hovav Shacham. Formal verification of the linux kernel ebpf verifier range analysis, 2022. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf.

[16] Scott Bradner and Jim McQuaid. Benchmarking methodology for network interconnect devices. Technical report, 1999.

[17] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. *Communications of the ACM*, 65(8):92–100, 2022. https://doi.org/10.1145/3543668.

[18] Luca Caviglione, Wojciech Mazurczyk, Matteo Repetto, Andreas Schaffhauser, and Marco Zuppelli. Kernel-level tracing for detecting stegomalware and covert channels in linux environments. *Computer Networks*, 191:108010, 2021. https://doi.org/10.1016/j.comnet.2021.108010.

[19] Howard Chen, Jiwei Lu, Wei-Chung Hsu, and Pen-Chung Yew. Continuous adaptive object-code re-optimization framework. In *Advances in Computer Systems Architecture: 9th Asia-Pacific Conference, ACSAC 2004, Beijing, China, September 7-9, 2004. Proceedings 9*, pages 241–255. Springer, 2004. https://doi.org/10.1007/978-3-540-30102-8_20.

[20] Cisco. T-rex traffic generator. https://trex-tgn.cisco.com/trex/doc/trex_manual.html, 2023. Accessed: 2023-10-30.

[21] Jack W Davidson and Christopher W Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):505–526, 1984. https://doi.org/0164-0925/84/1000-0505.

[22] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1):5–es, feb 2007. https://doi.org/10.1145/1210268.1210273.

[23] Advanced Micro Devices. *Software Optimization Guide for the AMD Zen4 Microarchitecture.* Advanced Micro Devices, Inc., Santa Clara, California, 2023. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/software-optimization-guides/57647.zip.

[24] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019. https://www.usenix.org/conference/atc19/presentation/duplyakin.

[25] Alexis Engelke and Martin Schulz. Robust practical binary optimization at run-time using llvm. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 56–64, 2020. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00011.

[26] William Findlay, Anil Somayaji, and David Barrera. Bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, page 91–103, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3411495.3421358.

[27] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011. https://agner.org/optimize/.

[28] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011. https://doi.org/10.1145/3276495.

[29] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual: Volume 1.* Intel Inc., Santa Clara, California, 2023. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[30] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 377–390, 2017. https://doi.org/10.1145/3133956.3134045.

[31] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 150–157, 2023. https://doi.org/10.1145/3593856.3595892.

[32] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf, 2023. https://arxiv.org/abs/2302.10366.

[33] Zachary H Jones. Performance analysis of {XDP} programs. 2021.

[34] Jeffrey Katcher. Postmark: A new file system benchmark. *TR3022*, 1997. https://www.filesystems.org/docs/auto-pilot/Postmark.html.

[35] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 147–158, New York, NY, USA, 1994. Association for Computing Machinery. https://doi.org/10.1145/178243.178256.

[36] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified programs can party: Optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 283–299, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3492321.3519562.

[37] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code in llvm. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. https://doi.org/10.1145/3276495.

[38] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, pages 308–322, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-13374-9_21.

[39] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. Secure namespaced kernel audit for containers. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 518–532, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3472883.3486976.

[40] Edward S Lowry and Cleburne W Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969. https://doi.org/10.1145/362835.362838.

[41] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *23rd Annual Network And Distributed System Security Symposium (NDSS 2016)*. Internet Soc, 2016. https://doi.org/10.14722/ndss.2016.23350.

[42] Larry W McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996. https://lmbench.sourceforge.net/.

[43] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. *SIGCOMM Comput. Commun. Rev.*, 49(3):2–17, nov 2019. https://doi.org/10.1145/3371927.3371929.

[44] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018. https://doi.org/10.1109/HPSR.2018.8850758.

[45] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. Fast in-kernel traffic sketching in ebpf. *ACM SIGCOMM Computer Communication Review*, 53(1):3–13, 2023. https://doi.org/10.1145/3594255.3594256.

[46] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151, 2021. https://doi.org/10.1109/TNSM.2021.3055676.

[47] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, 2017. https://doi.org/10.1145/3127479.3129249.

[48] Alessandro Rivitti, Roberto Bifulco, Angelo Tulumello, Marco Bonola, and Salvatore Pontarelli. Ehdl: Turning ebpf/xdp programs into hardware designs for the nic. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 208–223, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3582016.3582035.

[49] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 01, pages 209–217, 2018. https://doi.org/10.1109/ITC30.2018.00039.

[50] R Sekar, Hanke Kimm, and Rohit Aich. eaudit: A fast, scalable and deployable audit data collection system. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 87–87. IEEE Computer Society, 2023. http://seclab.cs.stonybrook.edu/seclab/pubs/eaudit.pdf.

[51] T. Simunic, L. Benini, G. De Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings 13th International Symposium on System Synthesis*, pages 193–198, 2000. https://doi.org/10.1109/ISSS.2000.874049.

[52] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed F. Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso. ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond). *IEEE Access*, 11:57174–57202, 2023. https://doi.org/10.1109/ACCESS.2023.3281480.

[53] Dave Jing Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Peter C. Johnson, and Kevin R. B. Butler. Lbm: A security framework for peripherals within the linux kernel. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 967–984, 2019. https://doi.org/10.1109/SP.2019.00041.

[54] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023. https://doi.org/10.1007/978-3-031-37709-9_12.

[55] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, apr 1991. https://doi.org/10.1145/103135.103136.

[56] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 67–72, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3281411.3281426.

[57] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 50–64, 2021. https://doi.org/10.1145/3452296.3472929.

[58] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association. https://www.usenix.org/conference/osdi22/presentation/zhong.

[59] Jianer Zhou, Zengxie Ma, Weijian Tu, Xinyi Qiu, Jingpu Duan, Zhenyu Li, Qing Li, Xinyi Zhang, and Weichao Li. Cable: A framework for accelerating 5g upf based on ebpf. *Computer Networks*, 222:109535, 2023. https://doi.org/10.1016/j.comnet.2022.109535.

[60] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association. https://www.usenix.org/conference/nsdi23/presentation/zhou.