Synthesizing Safe and Efficient Kernel Extensions for Packet Processing

Qiongwen Xu\textsuperscript{1}, Michael D. Wong\textsuperscript{2}, Tanvi Wagle\textsuperscript{1}, Srinivas Narayana\textsuperscript{1}, Anirudh Sivaraman\textsuperscript{3}

\textsuperscript{1}Rutgers University \textsuperscript{2}Princeton University \textsuperscript{3}New York University
k2 Compiler@email.rutgers.edu

ABSTRACT

Extended Berkeley Packet Filter (BPF) has emerged as a powerful method to extend packet-processing functionality in the Linux operating system. BPF allows users to write code in high-level languages (like C or Rust) and execute them at specific hooks in the kernel, such as the network device driver. To ensure safe execution of a user-developed BPF program in kernel context, Linux uses an in-kernel static checker. The checker allows a program to execute only if it can prove that the program is crash-free, always accesses memory within safe bounds, and avoids leaking kernel data.

BPF programming is not easy. One, even modest-sized BPF programs are deemed too large to analyze and rejected by the kernel checker. Two, the kernel checker may incorrectly determine that a BPF program exhibits unsafe behaviors. Three, even small performance optimizations to BPF code (e.g., 5% gains) must be meticulously hand-crafted by expert developers. Traditional optimizing compilers for BPF are often inadequate since the kernel checker’s safety constraints are incompatible with rule-based optimizations.

We present K2, a program-synthesis-based compiler that automatically optimizes BPF bytecode with formal correctness and safety guarantees. K2 produces code with 6–26% reduced size, 1.36–55.03% lower average packet-processing latency, and 0–4.75% higher throughput (packets per second per core) relative to the best clang-compiled program, across benchmarks drawn from Cilium, Facebook, and the Linux kernel. K2 incorporates several domain-specific techniques to make synthesis practical by accelerating equivalence-checking of BPF programs by 6 orders of magnitude.

1 INTRODUCTION

The CPU efficiency of processing packets at servers is of paramount importance, given the increasing volumes of data from large-scale applications, the stagnation of Moore’s law, the monetization of CPU cores in cloud computing, and the stringent throughput and latency requirements of high-end applications. The networking community has responded with several efforts, including innovations in operating system packet-processing [53, 58, 81, 125], user-space stacks [45, 47, 64, 87, 104, 116, 122], and programmable NIC offloads [15, 22, 24, 25, 74, 76].

Recently, extended Berkeley Packet Filter (BPF) has emerged as a popular method to achieve flexible and high-speed packet processing on the Linux operating system. With roots in packet filtering in the early 90s [106], BPF has since evolved into a general-purpose in-kernel virtual machine [59, 83] with an expressive 64-bit RISC instruction set. BPF code\textsuperscript{1} has been widely deployed in production systems—implementing load balancing [56, 130], DDoS protection [71], container policy enforcement [90], application-level proxying [21], intrusion-detection [70], and low-level system monitoring [42, 80]. Every packet sent to Facebook [130] and Cloud-Flare [71] is processed by BPF software.

BPF enables users to extend the functionality of the operating system without developing kernel software [79]. The user writes code in a high-level language (e.g., C, Rust), and uses a standard compiler toolchain (e.g., Clang-9) to produce BPF bytecode. The operating system leverages an in-kernel static checker, which analyzes the BPF bytecode to determine if it is safe to run in kernel context. Specifically, the checker attempts to prove that the program terminates, does not crash, does not access kernel memory beyond safe permitted bounds, and does not leak privileged kernel data. If the program is proved safe by the kernel checker, it is downloaded into kernel memory and run without any additional run-time checks. Otherwise, the program is rejected. BPF programs can be executed within several performance-critical parts of the packet-processing stack [103], like the network device driver [83], traffic control [54], congestion control [92], and socket filters [106].

BPF is unique in the combination of flexibility, safety, and performance it enables for packet processing. Unlike a kernel module that may potentially crash the kernel or corrupt kernel memory, a BPF program accepted by the kernel checker is guaranteed not to misbehave, assuming that the checker and the BPF run-time are bug-free. Unlike kernel-bypass stacks, BPF does not pin CPU cores, and retains the user-kernel privilege separation and standard management tools (e.g., tcpdump) available on operating systems [83, 123].

Despite the promises of BPF, it is not easy to develop high-quality packet-processing code in BPF today. We outline three challenges.

Challenge 1: Performance. Optimizing the performance of BPF code today is tantamount to optimizing assembly code. Userspace profiling tools do not exist. Optimization support in compilers is inadequate. For the benchmarks we tested, the standard compilation toolchain (based on Clang-9) produced identical code under optimization flags –O2 and –O3, missing opportunities available to optimize the –O2 code (88). Anecdotally, it is known that even expert developers have to put in painstaking work to improve performance of BPF code by small margins [49, 91, 121]. Yet, small improvements are worthwhile: reducing even a few clock cycles per packet is crucial to meeting the line rate at high speeds given the limited budget of CPU clock cycles available to process each packet [23, 72, 83]. Further, cutting the CPU usage of networking decreases interference to workloads co-hosted on the same machine [37, 96].

Challenge 2: Size. Running BPF programs beyond a modest size poses challenges. The kernel checker limits the complexity\textsuperscript{2} of

\textsuperscript{1}In this paper, we use the term BPF throughout to denote the extended version of BPF, rather than “classic” BPF used to write packet filters.

\textsuperscript{2}Older kernels (prior to v5.2) rejected programs with more than 40% BPF bytecode instructions. On modern kernels, this limit is still applicable to non-privileged BPF program types [34] such as socket filters and container-level packet filters [35]. Since
the programs that it deems acceptable [27, 88] to keep the time it takes to load user programs small. In practice, programs with even a few thousand instructions end up being rejected [29]. Further, hardware platforms supporting BPF offload are very sensitive to program size, given their limited amount of fast memory to hold the program [121]. Compiler support for code compaction is deficient; for most of our benchmarks, we found that clang -Os produces code of the same size as clang -O2. The only recourse for developers under size pressure is to refactor their program [3, 10, 28].

**Challenge 3: Safety.** It is difficult to get even small programs past the kernel checker. The checker’s static analysis is incomplete and imprecise: it rejects many programs which have semantically-equivalent rewrites that can be accepted (§6). This makes it tricky to develop compilers that produce runnable BPF bytecode. The developers of Clang’s BPF backend work specifically towards producing instruction sequences that the kernel checker will accept, e.g., [14, 16–19]. Producing checker-acceptable code is a major challenge in designing a BPF backend to the gcc compiler [60, 67].

Fundamentally, generating optimized, compact, and safe BPF code is challenging due to the incompatibility between checker-enforced safety restrictions and rule-based optimizations (§2.2). We call this the **phase-ordering** problem in BPF compilation: producing safe, checker-acceptable code precludes many traditional rule-based optimizations. Conversely, applying optimizations produces code that the kernel checker rejects.

**A synthesis-based compiler.** We present K2, a compiler which uses program synthesis to automatically generate safe, compact, and performant BPF bytecode, starting from unoptimized bytecode. Program synthesis is the task of searching for a program that meets a given specification [38]. An example of a specification is that the outputs of the synthesized program must match that of a source program on all inputs. Synthesis works by searching through the space of programs, typically guided by user-provided restrictions on the structure of the synthesized program. For example, the synthesizer may search for programs that fit a user-defined grammar [131, 132], use smaller library components [82, 89], or use a low-level instruction set [41, 112, 117, 126].

While traditional compilers are designed to emit “reasonable” code within a small time budget, synthesis-based compilers can produce high-quality code by searching the space of programs more extensively over a longer time period. We believe that the longer compilation time is worthwhile for BPF programs, given their prevalence in deployed systems, their sensitivity to performance, the difficulty of achieving even small performance gains, and their portability across machines and architectures [20, 113].

K2 makes three contributions.

**Contribution 1: Stochastic synthesis for BPF (§3).** K2 adapts stochastic synthesis [55, 127, 129] to the domain of the BPF instruction set. At a high level, the algorithm runs a Markov chain to search for programs with smaller values of a cost function that incorporates correctness, safety, and performance. A new candidate program is synthesized probabilistically using one of several rewrite rules that modify the current state (program) of the Markov chain. The Markov chain transitions to the new state (synthesized program) with a probability proportional to the reduction in the cost relative to the current program. We show how we set up K2 to optimize programs with diverse cost functions under safety constraints. We have incorporated several domain-specific rewrites to accelerate the search. At the end of the search, K2 produces multiple optimized versions of the same input program.

**Contribution 2: Techniques to equivalence-check BPF programs (§4, §5).** K2 synthesizes programs that are formally shown to be equivalent to the original program. To perform equivalence-checking, we formalize the input-output behavior of BPF programs in first-order logic (§4). Our formalization includes the arithmetic and logic instructions of BPF handled by earlier treatments of BPF [77, 114, 115, 137], and goes beyond prior work by incorporating aliased memory access (using pointers) as well as BPF maps and helper functions (§2). Equivalence-checking occurs within the inner loop of synthesis, and it must be efficient for synthesis to remain practical. We present several domain-specific techniques that reduce the time required to check the input-output equivalence of two BPF programs by five orders of magnitude (§5). Consequently, K2 can optimize real-world BPF code used in production systems.

**Contribution 3: Techniques to check the safety of BPF programs (§6).** At each step of stochastic search, K2 evaluates the safety of the candidate program. K2 incorporates safety checks over the program’s control flow and memory accesses, as well as several kernel-checker-specific constraints. To implement these checks, K2 employs static analysis and discharges first-order-logic queries written over the candidate program.

K2 resolves the phase-ordering problem of BPF compilation by considering both performance and safety of candidate programs at each step of the search. While K2’s safety checks have significant overlap with those of the kernel checker, the two sets of checks are distinct, as the kernel checker is a complex body of code that is under active development [26]. It is possible, though unlikely, that K2 deems a program safe but the kernel checker rejects it. To guarantee that K2’s outputs are acceptable to the kernel checker, K2 has a post-processing pass where it loads each of its best output programs into the kernel and weeds out any that fail the kernel checker. While the existence of this pass may appear to bring back the phase-ordering problem, it is merely a fail-safe: as of this writing, all of K2’s output programs resulting from the search already pass the kernel checker.

K2 can consume BPF object files emitted by clang and produce an optimized, drop-in replacement. We present an evaluation of the compiler across 19 programs drawn from the Linux kernel, Cilium, and Facebook. Relative to the best clang-compiled variant (among -02/-03/-Os), K2 can reduce the size of BPF programs by between 6–26%, reduce average latency by 1.36%–55.03%, and improve throughputs (measured in packets per second per core) by 0–4.75%. This is in comparison to a state of the art where significant effort is required from expert developers to produce 5–10% performance gains [91, 130].
K2 is an existence proof that domain-specific application of program synthesis techniques is a viable approach to automatically optimizing performance-critical packet-processing code. We call upon the community to explore such technology to alleviate the developer burden of improving performance in other contexts like user-space networking and programmable NICs. K2’s source code, including all of our experimental scripts, is available at https://k2.cs.rutgers.edu/.

2 BACKGROUND AND MOTIVATION

2.1 Extended Berkeley Packet Filter (BPF)

BPF is a general-purpose in-kernel virtual machine and instruction set [59] that enables users to write operating system extensions for Linux [79]. A standard compiler (e.g., Clang-9) can be used to turn C/Rust programs into BPF bytecode, whose format is independent of the underlying hardware architecture.

BPF programs are event-driven. BPF bytecode can be attached to specific events within the operating system, such as the arrival of a packet at the network device driver [83], packet enqueue within Linux traffic control [94], congestion control processing [92], and socket system call invocations [103].

Stateful functionality is supported using **BPF helper functions.** Helper functions are implemented as part of the kernel and can be called by the BPF program with appropriate parameters. For example, there are helper functions that provide access to persistent key-value storage known as a map. The map-related helper functions include lookup, update, and delete. The arguments to the map helpers include pointers to memory and file descriptors that uniquely identify the maps. The list and functionality of helpers in the kernel are steadily increasing; there are over 100 helpers in the latest kernel as of this writing [95].

The BPF instruction set follows a 64-bit RISC architecture. Each program has access to eleven 64-bit registers, a program stack of size 512 bytes (referred to by the stack pointer register r10), and access to the memory containing program inputs (such as packets) and some kernel data structures (e.g., socket buffers). The BPF instruction set includes 32 and 64-bit arithmetic and logic operations, signed and unsigned operations, and pointer-based load and store instructions. BPF programs can be executed efficiently by leveraging just-in-time (JIT) compilation to popular architectures like x86, ARM, and RISC-V.

The BPF instruction set includes the following instructions:

- **bpf_mov** rX rY // rY = rX
- **bpf_stx** rX rY // *(rX) = rY
- **bpf_st** rX rY // *(rX) = rY
- **bpf_st_imm** rX off1 // *(rX + off1) = value
- **bpf_st_imm8** rX off1 // *(u8*)(rX + off1) = value
- **bpf_st_imm16** rX off1 // *(u16*)(rX + off1) = value

Example 1. Invalid strength reduction. The sequence

```plaintext
bpf_mov rY 0 // rY = 0
bpf_stx rX rY // *(rX) = rY
```

for some registers rX \(\neq rY\) can usually be optimized to the simpler single instruction

```plaintext
bpf_st_imm rX 0 // *(rX) = 0
```

However, the kernel checker mandates that a pointer into the program’s “context memory” [9] cannot be used to store an immediate value. If rX were such a pointer, the program would be rejected.

**Example 2. Invalid coalescing of memory accesses.** Consider the instruction sequence

```plaintext
bpf_st_imm8 rX off1 0 // *(u8*)(rX + off1) = 0
bpf_st_imm8 rX off2 0 // *(u8*)(rX + off2) = 0
```

where rX is a safely-accessible memory address, and off1 and off2 are offsets such that off2 = off1 + 1. Usually, two such 1-byte writes can be combined into one 2-byte write:

```plaintext
bpf_st_imm16 rX off1 0 // *(u16*)(rX + off1) = 0
```

However, the kernel checker mandates that a store into the stack must be aligned to the corresponding write size [8]. If rX is r10, the stack pointer, and off1 is not 2-byte aligned, the checker will reject the rewritten program.

In general, applying optimizations that pass the checker’s constraints requires compilers to be aware of the specific restrictions that impact each optimization. The checker has numerous restrictions [5, 6], making it tedious to consider the cross-product of optimizations and safety conditions.

2.3 K2: A Program-Synthesis-Based Compiler

We present K2, a compiler that leverages program synthesis to consider correctness, performance, and safety of programs together rather than piecemeal, to resolve the phase-ordering problem between efficiency and safety in BPF optimization.

Program synthesis is the combinatorial search problem of finding a program that satisfies a given specification. Appendix A overviews program synthesis approaches in the literature. Given a sequence of instructions in the BPF bytecode format, we are interested in synthesizing an alternative sequence of BPF instructions that satisfies the specification that: (i) the synthesized program is equivalent to the source program in its input-output behavior, (ii) the synthesized program is safe, and (iii) the synthesized program is more efficient than the source program. The precise definitions of efficiency and safety will be discussed in §3 and §6.

Fig. 1 presents an overview of K2, which synthesizes programs satisfying the specification above. K2 consumes Clang-compiled BPF bytecode, and implements the stochastic search procedure described in §3. The search process synthesizes proposals, which are candidate rewrites of the bytecode. The proposal is evaluated against a suite of automatically-generated test cases to quickly prune programs which are not equivalent to the source program, or unsafe. If the proposal passes all tests, K2 uses formal equivalence-checking (§4, §5) and formal safety-checking (§6) to determine the value of a cost function over the proposal. The cost combines correctness, safety, and performance characteristics, and is used to guide the search process towards better programs. Formal equivalence-checking and safety-checking may generate counterexamples, i.e., inputs where the proposal’s output differs from that of the original bytecode, or the proposal exhibits unsafe behaviors. These tests are
a candidate rewrite, i.e., a proposal $p_{\text{synth}}$, using one of the rules below, chosen randomly with fixed probabilities $\text{prob}_{(i)}$:

1. **Replace an instruction** ($\text{prob}_{\text{ir}}$): at random, choose an instruction from $p_{\text{curr}}$, and modify both its opcode and operands. For example, change $\text{bpf}_{\text{add}} r1 4$ to $\text{bpf}_{\text{mov}} r4 r2$.

2. **Replace an operand** ($\text{prob}_{\text{op}}$): at random, choose an instruction and replace one of its operands with another value of the same type. For example, change $\text{bpf}_{\text{add}} r1 4$ to $\text{bpf}_{\text{add}} r1 10$.

3. **Replace by NOP** ($\text{prob}_{\text{nop}}$): at random, choose an instruction and replace it with a nop, effectively reducing the number of instructions in the program.

4. **Exchange memory type 1** ($\text{prob}_{\text{mem1}}$): at random, choose an instruction, and if it is a memory-based instruction (i.e., a load or store), sample a new width for the memory operation and a new immediate or register operand. The instruction’s memory address operand (i.e., address base and offset) as well as its type (load vs. store) are unchanged. For example, change $r1 = *(&u16*) (r2 - 4)$ to $r3 = *(&u32*) (r2 - 4)$.

5. **Exchange memory type 2** ($\text{prob}_{\text{mem2}}$): at random, choose an instruction, and if it is a memory-based instruction, sample a new width for the memory operation. All other instruction operands are unchanged. For example, change $r1 = *(&u16*) (r2 - 4)$ to $r1 = *(&u32*) (r2 - 4)$.

6. **Replace contiguous instructions** ($\text{prob}_{\text{cir}}$): at random, choose up to $k$ contiguous instructions (we pick $k = 2$) and replace all of them with new instructions.

These rewrite rules define the transition probabilities of the Markov chain, which we denote by $\text{tr}(p_{\text{curr}} \rightarrow p_{\text{synth}})$. We use the probabilities $\text{prob}_{(i)}$ shown in Table 8 (Appendix F.1). In our experience, any probabilities that allow the Markov chain to move “freely” through the space of programs suffice to find programs better than the input.

**Non-orthogonality of rewrite rules.** The rewrite rules above are not mutually exclusive in the program modifications they affect. For example, replacement by NOP (rule 3) is just a specific version of the more general instruction replacement (rule 1). Given enough time, a small set of general rules is sufficient to explore the space of programs. However, the existence of more specific rules accelerates the convergence of the Markov chain to better programs.

**Domain-specificity of rewrite rules.** STOKE and its variants [55, 127] proposed variants of the rewrite rules (1–3) above. Rules (4), (5), and (6) are domain-specific rules that K2 uses to accelerate the search for better BPF programs. Rules (4) and (5) help identify memory-based code optimizations (§9). Rule (6) captures one-shot replacements of multiple instructions, e.g., replacing a register addition followed by a store into a single memory-add instruction. These domain-specific rules improve both the quality of the resulting programs and the time to find better programs (§8, Appendix F.1).

### 3.2 Cost Function

We compute a cost function over each candidate program. The cost function $f(p)$ contains three components: an error cost, a performance cost, and a safety cost.

**Error cost.** The error cost function $\text{err}(p)$ is 0 if and only if the program $p$ produces the same output as the source program $p_{\text{src}}$
on all inputs. We would like a function that provides a smooth measure of the correctness of program \( p \) with respect to the source program \( p_{src} \), to guide the search towards “increasingly correct” programs. Similar to STOKE, we incorporate test cases as well as formal equivalence checking (§4 & §5) to compute an error cost. Using a set of tests \( T \) and executing \( p_{synth} \) on each test \( t \in T \), we set

\[
\text{err}(p) := c \cdot \sum_{t \in T} \text{diff}(o_{p_{synth}(t)}, o_{p_{src}(t)}) + \text{unequal} \cdot \text{num_tests}
\]

where:
- \( o_{p_{synth}(t)} \) and \( o_{p_{src}(t)} \) are the outputs of the proposal and the source program on test case \( t \),
- \( \text{diff}(x, y) \) is a measure of the distance between two values. We consider two variants: (i) \( \text{diff}_{\text{pop}}(x, y) := \text{popcount}(x \oplus y) \) is the number of bits that differ between \( x \) and \( y \), and (ii) \( \text{diff}_{\text{abs}}(x, y) := \text{abs}(x - y) \), which represents the absolute value of the numerical difference between \( x \) and \( y \). Relative to STOKE, which only considers \( \text{popcount} \) as the semantic distance between values, we also find that many packet-processing programs require numeric correctness (e.g., counters), captured via \( \text{diff}_{\text{abs}}(\cdot) \).
- \( c \) is a normalizing constant denoting the weight of each test case. STOKE adds the full error cost for each test case, setting \( c = c_{\text{full}} = 1 \). We also explore a second variant, \( c_{\text{avg}} = 1/|T| \), where \(|T|\) is the number of test cases, to normalize the contributions of the many test cases we require to prune complex, “almost correct” BPF programs.
- \( \text{unequal} \) is \( 0 \) if the first-order-logic formalization of the two BPF programs (§4) finds that the programs are equivalent, else it is 1. We only run equivalence-checking if all test cases pass, since it is time-consuming. If any test case fails, we set \( \text{unequal} \) to 1.
- \( \text{num_tests} \) includes two variants: (i) the number of test cases on which \( p \) produced incorrect outputs, and (ii) the number of test cases on which \( p \) produced correct outputs. STOKE uses only the first variant. We consider the second variant to distinguish a program that is equivalent to the source program from one that satisfies all the test cases but is not equivalent.
- Considering all variants from equation (1), there are 8 error cost functions. We run MCMC with each cost function in parallel and return the best-performing programs among all of them.

**Performance cost.** We use two kinds of performance costs corresponding to different scenarios, namely optimizing for program size and program performance.

The function \( \text{perf}_{\text{inst}}(p_{synth}) \) (instruction count) is the number of extra instructions in \( p_{synth} \) relative to \( p_{src} \).

The function \( \text{perf}_{\text{lat}}(p_{synth}) \) is an estimate of the additional latency of executing program \( p_{synth} \) relative to \( p_{src} \). Unfortunately, executing a candidate BPF program \( p_{synth} \) to directly measure its latency is unviable, since the kernel checker will reject most candidate programs. Instead, we profile every instruction of the BPF instruction set by executing each opcode millions of times on a lightly loaded system, and determining an average execution time \( \text{exec}(i) \) for each opcode \( i \). The performance cost function is the difference of the sum of all the opcode latencies, i.e., \( \text{perf}_{\text{lat}}(p_{synth}) := \sum_{i \in p_{synth}} \text{exec}(i_{synth}) - \sum_{i \in p_{src}} \text{exec}(i_{src}) \).

**Safety cost.** To our knowledge, K2 is the first synthesizing compiler to incorporate generic safety constraints in first-order logic into synthesis. The safety properties considered by K2 are described in §6. Our approach to dealing with unsafe programs is simple: once a program \( p_{synth} \) is deemed unsafe, we set \( \text{safe}(p_{synth}) \) to a large value \( \text{ERR}_\text{MAX} \), leaving just a small probability for it to be accepted into the Markov chain. We set \( \text{safe}(p_{synth}) = 0 \) for safe programs. We do not simply reject unsafe programs because the path from the current program to a more performant and safe program in the Markov chain may pass through an unsafe program (for some intuition on why, see Fig. 4 in [127]). We leave formulating smooth cost functions to guide the search through progressively “safer” programs to future work.

The final cost function we use is \( \alpha \cdot \text{err}(p_{synth}) + \beta \cdot \text{perf}(p_{synth}) + \gamma \cdot \text{safe}(p_{synth}) \). We run parallel Markov chains with different \((\alpha, \beta, \gamma)\) and return the programs with the least performance costs.

### 3.3 Proposal Acceptance

To determine whether a candidate proposal should be used as the next state of the Markov chain, the cost \( f(p_{synth}) \) is turned into the probability of \( p_{synth} \) in the steady-state distribution, as follows [78]:

\[
\pi(p_{synth}) = \frac{e^{-\beta \cdot f(p_{synth})}}{Z}
\]

where \( Z = \sum_{p} e^{-\beta \cdot f(p)} \). The Metropolis-Hastings algorithm computes an acceptance probability for \( p_{synth} \) as follows:

\[
\alpha = \min \left( 1, \frac{\pi(p_{synth}) \cdot \text{tr}(p_{synth} \rightarrow p_{curr})}{\pi(p_{curr}) \cdot \text{tr}(p_{curr} \rightarrow p_{synth})} \right)
\]

With probability \( \alpha \), the next state of the Markov chain is set to \( p_{synth} \), else the next state is just \( p_{curr} \). Here, the \( \text{tr}() \) are the transition probabilities between programs (§3.1). Intuitively, \( p_{synth} \) is always accepted if its cost is lower than that of \( p_{curr} \). Otherwise, \( p_{synth} \) is accepted with a probability that decreases with the increase in the cost of \( p_{synth} \) relative to \( p_{curr} \).

K2 repeats the process in §3.1, §3.2, and §3.3 from the new state of the Markov chain, looping until a timeout.

### 4 CHECKING THE EQUIVALENCE OF BPF PROGRAMS

K2 synthesizes output programs that are formally shown to be equivalent to the input program. To do this, we first formalize the input-output behavior of the two programs in first-order logic, using the theory of bit vectors [98]. We identify the input and output registers of the two programs based on the kernel hook they attach to [88]. Then, we dispatch the logic query below to a solver:

- inputs to program 1 \( \Rightarrow \) inputs to program 2
- \( \wedge \) input-output behavior of program 1
- \( \wedge \) input-output behavior of program 2
- \( \Rightarrow \) outputs of program 1 \( \neq \) outputs of program 2

If the formula is satisfiable, there is a common program that causes the outputs of the two programs to differ, which is added to the test suite (§3). If the formula is unsatisfiable, the two programs are equivalent in terms of input-output behaviors.

The rest of this section describes how we obtain the input-output behavior of a single program in first-order logic. Our formalization
handles arithmetic and logic instructions (§4.1), memory access instructions (§4.2), and BPF maps and other helper functions (§4.3). We have checked the soundness of our formalization using a test suite that compares the outputs produced by the logic formulas against the result of executing the instructions with given inputs.

Preliminaries. We begin by reordering the instructions in the program so that all control flow only moves forward. This is possible to do when a BPF program does not contain any loops. Then, we convert the entire program into static-single-assignment (SSA) form [39, 63]. The result after SSA conversion is a sequence of BPF bytecode instructions where (i) each assignment to a register uses a fresh label with a version number, e.g., `bpf_mov r0 1; bpf_mov r0 2` is turned into `bpf_mov r0_v1 1; bpf_mov r0_v2 2`, and (ii) each statement is associated with a well-defined path condition [110]. For example, in the instruction sequence,

\[
\begin{align*}
\text{bpf} &\text{eq r1 0 1} \quad // \text{if r1} \neq 0: \\
\text{bpf} &\text{mov r2 1} \quad // \quad r2 = 1
\end{align*}
\]

the second instruction is associated with the path condition `r1 = 0`.

At the highest level, we construct first-order formulas corresponding to each instruction, and conjoin them, i.e., through the logical conjunction operator `∧`, to produce a final formula that represents the input-output relationship of the entire program. Now we discuss how K2 formalizes each kind of instruction.

4.1 Arithmetic And Logic Instructions

To model register-based arithmetic and logic instructions, we represent each version of each register using a 64-bit-wide bit vector data type. The action of each instruction is formalized by representing its impact on all the registers involved. Our formalization handles both 32-bit and 64-bit opcodes, as well as signed and unsigned interpretations of the data.

As an example, consider the 32-bit arithmetic instruction `bpf_add32 dst src` (opcode `0x84`) which has the action of taking the least significant 32 bits of the registers `dst` and `src`, adding them, and writing back a (possibly truncated) 32-bit result into `dst`, zeroing out the most significant 32 bits of `dst` register.

Suppose we are given a single instruction `bpf_add32 dst_x src_y` (after SSA) where `x` and `y` represent the version numbers of `dst` and `src`, respectively. Suppose the result is stored in `dst_z`. This instruction results in the formula

\[
\begin{align*}
\text{tmp} &\equiv (\text{dst}_x.\text{extract}(31, 0) + \\
\text{src}_y.\text{extract}(31, 0)) \quad ∧ \\
\text{dst}_z &\equiv \text{concat}( \\
&\quad \text{bv32}(0), \text{tmp}.\text{extract}(31, 0))
\end{align*}
\]

where `tmp` is a fresh variable to hold the intermediate result of the 32-bit addition of `dst_x` and `src_y`, `extract(a, b)` represents the effect of picking up bits a...b of a given bit vector, `concat(x, y)` represents the bit vector produced by concatenating the two bit vectors `x` and `y`, with `x` occupying the higher bits of significance in the result, and `bv32(0)` is a 32-bit bit vector representing 0.

Similarly, we have constructed semantic representations of all 64-bit and 32-bit arithmetic and logic instructions [4].

4.2 Memory Access Instructions

BPF supports memory load and store instructions of varying sizes [4] using pointers. We encode memory operations directly in the theory of bit vectors to produce an efficient encoding in a single first-order theory. We show how this encoding occurs in three steps.

Step 1: Handling loads without any stores. Suppose a BPF program contains no stores to memory, and only load instructions, i.e., `bpf_ld rX rY`. To keep the descriptions simple, from here on we will use the notation `rX = rY` to represent the instruction above.

The key challenge in encoding loads is handling aliasing, i.e., different pointers `rY` might point to the same memory region, and hence the different `rX` must have the same value.

Suppose the `i`th load instruction encountered in the program reads from memory address `rY_i` and loads into register `rX_i`. Then for the `i`th load, we conjoin the formula

\[
∧ \big( rY_j \neq rY_i \quad ⇒ \quad rX_j \neq rX_i \big)
\]

Formulating this formula requires maintaining all the previous loads in the program that might affect a given load instruction. To achieve this, K2 maintains a memory read table for the program: the source and destination of each load is added to this table in the order of appearance in the post-SSA instruction sequence. K2 handles partial overlaps in loaded addresses by expanding multi-byte loads into multiple single-byte loads.

Step 2: Handling stores and loads in straight-line programs. Stores complicate the formula above due to the fact that a load of the form `rX = rY` must capture the latest write to the memory pointed to by `rY`. For example, in the instruction sequence `rX_1 = rY; rX_2 = rY`, the first and second load from `rY` may return different values to be stored in `rX_1` and `rX_2`.

Suppose the program contains no branches. Then, the latest write to a memory address can be captured by the most recent store instruction (in order of encountered SSA instructions), if any, that writes to the same address. K2 maintains a memory write table, which records the memory address and stored variable corresponding to each store in the program. Suppose `k` stores of the form `rY_i = rX_i` (1 from 1 · · · k) have been encountered in the program before the load instruction `rX_i = rY_i`. Then, the load is encoded by the formula

\[
∧ \big( j < k \quad ∧ \quad rY_{j+1} = rY_{j+1} \quad ∧ \quad rX_{j+1} = rX_{j+1} \big)
\]

The formula `∧ (j < k)` asserts that the address loaded isn’t any of the addresses from stores that are more recent than store `j`. Hence, if `rY_j = rY_j`, the loaded value must come from store `j`.

Informally, the overall formula for a load instruction takes the form: if the address was touched by a prior store, use the value from that store, otherwise use the aliasing clauses from the "loads-only" case in step (1) above. Together, step (1) and step (2) complete K2’s encoding of memory accesses for straight-line programs.

\[\text{It is possible for a load to occur without a prior store e.g., when an instruction reads from input packet memory.}\]
Step 3: Handling control flow. We construct a single formula per instruction including control flow akin to bounded model checking [46]. Our key insight to generalize the encoding from step (2) above is to additionally check whether the path condition of the load instruction is implied by the path condition of the prior store:

\[ \land_{j \in \mathcal{S}_k} \land_{i,j \in \mathcal{S}_k} ! (r_{Y_i} = r_{Y_l} \land pc_i \Rightarrow pc_l) \land (r_{Y_j} = r_{Y_l} \land pc_j \Rightarrow pc_l) \Rightarrow r_{X_j} = r_{X_l} \]

Note that the path conditions \(pc_j\) of each load or store \(j\) are already computed by K2 during the preliminary SSA pass.

4.3 BPF Maps and Helper Functions

BPF helpers (§2) provide special functionality in a program, including stateful operation. Due to space constraints, we only briefly discuss our formalization of BPF maps—the most frequently used helpers—in this subsection. A more detailed treatment of maps and other helpers is available in Appendix B.

Maps. BPF maps are similar to memory, in that they can be read and written using a key (rather than an address). However, two features make BPF maps very different from memory.

First, the inputs to lookup, update, or delete a map entry in BPF’s map API are all pointers to memory holding a key or value. This results in two levels of aliasing: distinct pointers may point to the same location in memory (like regular pointer aliasing); additionally, distinct locations in memory may hold the same key, which must result in the same value upon a map look-up. Intuitively, we handle these two levels by keeping two pairs of tables for read and write operations. The first pair of read/write tables tracks the contents of the addresses corresponding to the key and value pointers, as in §4.2. The second pair of read/write tables tracks the updates to the value pointers corresponding to the map’s actual keys.

Second, keys in a map can be deleted, unlike addresses in memory. Our encoding treats a deletion as a update of the value pointer to \(\theta\) (a null pointer) for the corresponding key, so that any subsequent lookup returns null, mimicking the BPF lookup function semantics.

Other helper functions. Each helper function considered for optimization ideally should be formalized using its symbolic semantics. We have added formalizations for helpers used to obtain random numbers, access the current Unix timestamp, adjust memory headroom in a packet buffer, and get the ID of the processor on which the program is running.

The list of BPF helpers currently numbers in the hundreds and is growing [62, 95]. For most helpers, it is possible to model the function application as a call to an uninterpreted function \(f(.)\) [51]: the only governing condition on the input/output behavior of the function is that calling it with the same inputs will produce the same outputs, i.e., \(x = y \Rightarrow f(x) = f(y)\). (Stateful functions include the state as part of the inputs.) While such modeling is general, it limits the scope of optimization across function calls, since it is impossible to prove equivalence of code involving uninterpreted functions without requiring that the sequence of function calls and the inputs to each function call must be exactly the same in the input and the output programs.

5 FAST EQUIVALENCE CHECKING

The formulas generated in §4.1–§4.3 are in principle sufficient to verify the equivalence of all BPF programs we have tested. However, the corresponding verification task is too slow (§8). Equivalence-checking time grows quickly with the number of branches, the number of memory accesses, and the number of distinct maps looked up in the BPF programs. Equivalence checking is in the inner loop of synthesis (Fig. 1): large verification times render synthesis impractical.

We have developed several optimizations that accelerate equivalence-checking times by 6 orders of magnitude on average over the programs we tested. This section summarizes the key ideas; more details are available in Appendix C. Several optimizations leverage lightweight static analysis that is only feasible due to the restrictions in the BPF instruction set.

The time to solve a logic formula is often reduced significantly by assigning specific values to, i.e., concretizing, formula terms whose value is otherwise unconstrained, i.e., symbolic [46, 50, 57, 97, 119]. Our first three optimizations are of this kind.

I. Memory type concretization. All pointers to memory in BPF programs have well-defined provenance, i.e., it is possible to develop a static analysis to soundly and completely track the type of memory (stack, packet, etc.) that each pointer references. This allows K2 to maintain separate read and write tables (§4.2) for each memory type. Consequently, the size of aliasing-related formulas reduces from \(O((\sum N_i)^2)\) to \(O(\sum N_i^2)\), where \(N_i\) refers to the number of accesses to memory of a specific type \(t\).

II. Map type concretization. Similar to memory-type concretization, a simple static analysis can soundly and completely determine the map that is used for a specific lookup or update instruction. This has the effect of breaking map accesses across several maps in the two-level map tables (§4.3) into separate map-specific two-level tables.

III. Memory offset concretization. Many packet-processing programs perform reads and writes into memory at offsets that can be determined at compile time, for example, specific packet header fields. We developed a “best-effort” static analysis to soundly determine if a pointer holds a reference to a compile-time-known offset into a memory region. If such a constant offset is determined, a formula like \(r_{Y_i} = r_{Y_l}\) (appearing in §4.2) can be simplified to \(\text{constant} = r_{Y_i}\), or even \(\text{constant} = \text{constant} = \text{constant} = \text{constant}\). The latter doesn’t even require a solver to be evaluated, and can result in several cascading clause simplifications. In the limit, if all offsets can be concretely determined, this optimization has the effect of modeling the entire memory as if it is a set of named registers. If we cannot statically determine concrete offsets, we fall back to the symbolic formulas described in §4.2.

IV. Modular verification. K2 scales to large programs by synthesizing and verifying instruction sequences of smaller length within localized “windows” in the program, and then combining the results across the windows. Hence, K2 parses down the verification task to correspond to the size of the window rather than that of the full program. Effectively, this would turn K2 into a peephole optimizer [107]. However, traditional peephole optimizers necessitate that the rewrites must apply in any program context, rejecting many
strong optimizations that could work conditionally within a specific part of the program (e.g., r1 *= r3 may be changed into r1 <= r2 if the value of r3 is known to be 2). To discover strong optimizations but keep equivalence-checking fast, we develop window-based formulas that use stronger preconditions and weaker postconditions than peephole optimizers. K2 leverages variable liveness (as in prior work [40]) as well as concrete values of the live variables, both of which are inferred through static analysis:

\[
\begin{align*}
\text{variables live into window 1} &\Rightarrow \text{variables live into window 2} \\
\land \text{inferred concrete valuations of variables} &\land \text{input-output behavior of window 1} \\
\land \text{input-output behavior of window 2} &\Rightarrow \text{variables live out of window 1} \\
\neq \text{variables live out of window 2}
\end{align*}
\]

V. Caching. We cache the outcomes of equivalence-checking a candidate program to quickly determine if a structurally-similar program was checked earlier. This has the effect of reducing the number of times we call the solver. We canonicalize the program by removing dead code before checking the cache.

6 SAFETY OF BPF PROGRAMS

K2 ensures that the programs returned by the compiler are safe, which requires proving specific control-flow and memory-access safety properties about the output programs, described below.

K2’s safety checks are implemented using static analysis and first-order logic queries over the candidate programs generated at each step of the stochastic search (§3). By considering safety with optimization at each step, K2 resolves the phase-ordering problem (§2.2) that hampers traditional optimizing compilers for BPF.

K2’s safety checks are distinct from those of the kernel checker, though there is a significant overlap between them. We developed safety-checking directly within K2, eschewing the alternative approach of invoking the kernel checker on a candidate program at each step of search, for two reasons. First, in addition to reporting that a program is unsafe, K2’s safety queries also return a safety counterexample, i.e., an input that causes the program to exhibit unsafe behaviors. The counterexample can be added to the test suite (Fig. 1) to prune unsafe programs by executing them in the interpreter, rather than using an expensive kernel checker (system) call. This has the overall effect of speeding up the search loop. Second, the kernel checker is a complex piece of software that is evolving constantly. We believe that, over the long term, a logic-based declarative encoding of the safety intent will make it easier to understand and maintain the compiler’s safety constraints.

K2 guarantees that the output programs returned to the user will pass the kernel checker. K2 achieves this using a post-processing pass: outputs from K2’s search loop which fail the kernel checker are removed before presenting them to the user. As of this writing, all the outputs from K2’s search already pass the kernel checker.

Now we discuss K2-enforced safety properties in detail.

Control flow safety. The structure of BPF jump instructions [4] allows the set of possible jump targets in the program to be determined at compile time. Hence, K2 constructs the complete control flow graph over basic blocks at compile time [39]. Programs synthesized by K2 satisfy the following safety properties:

1. There are no unreachable basic blocks.
2. The program is loop-free (i.e., no “back-edges” in the control flow), and hence, terminates. K2 ensures this during proposal generation (§3.1) by only producing jump offsets taking control flow “forward” in a topologically-sorted list of basic blocks.
3. The program has no out-of-bounds jumps. K2 ensures this by only synthesizing jump targets that are within the program’s valid set of instructions.

The rest of the safety checks below are implemented using first-order logic queries. Logic queries provide safety counterexamples, which also allow K2 to prune an unsafe program using the interpreter rather than an expensive solver query down the road. To our knowledge, K2 is the first to leverage counterexamples for both correctness and safety during synthesis.

Memory accesses within bounds. K2 ensures that programs it synthesizes only access operating system memory within the bounds they are allowed to. The access bounds for each type of memory are known ahead of time. For example, the size of the program stack is fixed to 512 bytes [88]; packet inputs are provided with metadata on the start and end addresses; and BPF map values have a pre-defined fixed size based on the known attributes of the map.

K2 leverages a sound and complete static analysis to determine the type of memory that a load or store instruction uses. Then, K2 formulates a first-order query to determine if there are any program inputs that cause the access to violate the known safe bounds of that memory. K2 considers both the offset and the size of the access, and models the types of pointers returned from BPF kernel helpers functions very precisely. For example, the instruction sequence corresponding to \( r0 = \text{bpf_map_lookup}(...) ; r1 = \text{r0} \) will produce a safety counterexample for the case when the lookup returns a NULL pointer. However, \( r0 = \text{bpf_map_lookup}(...) ; \text{if (r0 != 0)} \{ \text{r1 = *r0} ; \} \) is considered safe, since the path condition ensures a valid value for \( r0 \).

Memory-specific safety considerations. The BPF kernel checker explicitly requires that a stack memory address cannot be read by a BPF program before that address is written to [88]. The same rule applies to registers which are not program inputs. This restriction is distinct from placing safe bounds on an address that is read, since an address that is considered unsafe to read at one moment, i.e., before a write, is considered safe to read after the write. K2 leverages the memory write table (§4.2) to formulate a first-order query that checks for semantically-safe loads from the stack under all program inputs. Further, the stack pointer register \( r10 \) is read-only; K2’s proposal generation avoids sampling \( r10 \) as an instruction operand whenever that operand might be modified by the instruction (§3.1).

Access alignment. The kernel checker enforces that memory loads and stores of a certain size happening to specific memory types (e.g., the stack) must happen to addresses aligned to that size. That is, an address \( a \) with an \( N \)-byte load or store must be such that \( a \mod N == 0 \). For example, the two instructions bpf_stxw and bpf_stxdw will require two different alignments, up to 32 bits and up to 64 bits, respectively.
Somewhat surprisingly, all of the safety properties above can be decided with sound and complete procedures due to the simplicity of the BPF instruction set.

**Modeling checker-specific constraints.** We encode several other specific properties enforced by the kernel checker. These checks can distinguish semantically-equivalent code sequences that meet with different verdicts (accept versus reject) in the Linux kernel checker. We added these checks “on-demand”, as we encountered programs from K2 that failed to load. A selection of kernel-checker-specific safety properties we encoded include:

1. Certain classes of instructions, such as ALU32, NEG64, OR64, etc. are disallowed on pointer memory;
2. Storing an immediate value into a pointer of a specific type (PTR_TO_CTX [6]) is disallowed;
3. Registers r1 · · · r5 are clobbered and unreadable after a helper function call [88];
4. Aliasing pointers with offsets relative to the base address of a (permitted) memory region is considered unsafe.

Our encoding of kernel checker safety properties is incomplete; we believe it will be necessary to keep adding to these checks over time as the kernel checker evolves. A distinct advantage of a synthesis-based compiler is that such checks can be encoded once and considered across all possible optimizations, rather than encoded piecemeal for each optimization as in a rule-based compiler.

7 IMPLEMENTATION

We summarize some key points about the implementation of K2 here. More details are available in Appendix D.

K2 is implemented in 24500 lines of C++ code and C code, including proposal generation, program interpretation, first-order logic formalization, optimizations to equivalence-checking, and safety considerations. K2 consumes BPF bytecode compiled by clang and produces an optimized, drop-in replacement. The interpreter and the verification-condition-generator of K2 can work with multiple BPF hooks [103], fixing the inputs and outputs appropriately for testing and equivalence-checking. K2 uses Z3 [65] as its internal logic solver for discharging equivalence-checking and safety queries.

K2 includes a high-performance BPF interpreter that runs BPF bytecode using an optimized jumtable implementation similar to the kernel’s internal BPF interpreter [2]. We encoded a declarative specification of the semantics of most arithmetic and logic instructions in BPF using C preprocessor directives. This enabled us to auto-generate code for both K2’s interpreter and verification formula generator from the same specification of the BPF instruction set, akin to solver-aided languages [114, 136].

8 EVALUATION

In this section, we answer the following questions:

1. How compact are K2-synthesized programs?
2. How beneficial is K2 to packet latency and throughput?
3. Does K2 produce safe, kernel-checker-acceptable programs?
4. How useful are the optimizations to equivalence checking (§5)?
5. How effective are K2’s search parameters to find good programs?
6. How beneficial are K2’s domain-specific rules (§3.1)?

For questions (1) and (2), we compare K2-synthesized results with the best program produced by clang (across -01/02/03/Os.).

First, we describe how K2 is set up to produce the program we compare against clang. We choose the desired performance goal, which is either to reduce the instruction count or program latency. Then, we set off multiple runs of K2 in parallel, starting from the output of clang -02, and run them until a timeout. Each run uses a different parameter setting for its Markov chain (§3.2). In particular, we explore the 16 parameter settings described in Appendix F.1. Among these parallel runs, we choose the top-k best-performing programs which are safe and equivalent to the source program across all the Markov chains. We set k = 1 for the instruction count performance goal and k = 5 for the latency goal. Since the latency-based cost function used inside K2 is just an estimate of performance (§3.2), we measure the average throughput/latency performance of the top-k programs and return the best program.

We obtain our benchmark programs from diverse sources, including the Linux kernel’s BPF samples, recent academic literature [52], and programs used in production from Cilium and Facebook. We have considered 19 BPF programs in all, which attach to the network device driver (XDP), transport-level sockets, and system calls.

**Program Compactness.** Table 1 reports the number of instructions in K2-optimized programs relative to those of clang -01/-02/-03/Os (-02 and -03 are always identical). We show the compression achieved, the overall compile time, the time to uncover the smallest program for each benchmark, as well as some metrics on the complexity of the program being optimized, such as the number of total basic blocks and the length of the longest code path (measured in basic blocks). In all cases, K2 manages to compress the program beyond the best known clang variant, by a fraction that ranges from 6–26%, with a mean improvement of 13.95%. The average time to find the best program4 is about 22 minutes; often, the best program can be found much sooner.

Notably, K2 can handle programs with more than 100 instructions, something that even state-of-the-art synthesizing compilers find challenging [117]. The time to reach the best program displays significant variability. Programs with more instructions take longer to compress by the same relative amount. However, we do not find any significant relationship between optimization time and the number of basic blocks. Some examples of optimizations are in §9.

**Latency and throughput improvements.** We measure the improvements in packet-processing throughput and latency obtained by optimizing programs with K2. Improvements in the compiler’s estimated performance are presented in Appendix E.

4This average excludes the largest benchmark xdp-balancer, which is an outlier.
We report program throughput as the maximum loss-free forwarding rate (MLFFR [1]) of a single core. This is measured by increasing the offered load from the traffic generator slowly and recording the load beyond which the packet loss rate rises sharply. We measure throughput in millions of packets per second (Mpps) at 64-byte packet size. We use the minimum packet size since network-related CPU usage is proportional to packets per second rather than bytes per second, and XDP programs can easily saturate 100 Gbit/s on a single core with larger packet sizes [83]. Since latency varies with the load offered by the traffic generator, we report the latencies of the program variants at four separate offered loads: (i) low (load smaller than the throughput of the slowest variant), (ii) medium (load equal to the throughput of the fastest variant), (iii) high (load equal to the throughput of the slowest variant), and (iv) saturating (load higher than the throughput of all known variants). We average the results of 3 trials, with each result obtained after waiting 60 seconds or until the numbers stabilize.

K2’s measured improvements in throughput and latency over the best clang-compiled variant of the same program are summarized in Table 2 and Table 3. K2 provides small improvements in throughput ranging from 0.475%, while K2’s latency benefits range from 1.36%–55.03%. These benefits arise from target-specific optimizations with the latency cost function. (Appendix H shows detailed pictures of packet latency at varying loads.) More work remains before fully attaining the potential benefits of synthesis (§11).

### Table 1: K2’s improvements in program compactness across benchmarks from the Linux kernel (1–13), Facebook (14, 19), hXDP [52] (15, 16), and Cilium (17, 18). “DNL” means that the program variant did not load as it was rejected by the kernel checker.

| Benchmark          | Number of basic blocks | Number of instructions | When smallest prog. is found |
|--------------------|------------------------|------------------------|------------------------------|
|                    | All | Longest path | -O1 | -O2/-O3 | -Os | K2 | Compression | Time (sec) | Iterations |
| (1) xdp_exception  | 5   | 5            | 18  | 18      | 16  | 11.11% | 79  | 372,399    |
| (2) xdp_redirect   | 5   | 5            | 19  | 18      | 16  | 11.11% | 10  | 889        |
| (3) xdp_devmap_xmit| 6   | 6            | 36  | 36      | 29  | 19.44% | 1,201| 659,903    |
| (4) xdp_cpumap_kthread | 5 | 5           | 24  | 24      | 24  | 18    | 25.00% | 1,170 | 628,354 |
| (5) xdp_cpumap_enqueue | 4 | 3          | 26  | 26      | 26  | 21    | 19.23% | 1,848 | 300,438 |
| (6) sys_enter_open | 13  | 6           | 24  | 24      | 24  | 24    | 16.67% | 519  | 834,179 |
| (7) socket/0       | 13  | 6           | 32  | 29      | 29  | 27    | 6.90%  | 6    | 914      |
| (8) socket/1       | 20  | 17          | 35  | 32      | 32  | 30    | 6.25%  | 9    | 3,455    |
| (9) xdp_router_ipv4| 5   | 5           | 139 | 111     | 99  | 10.81%| 898  | 354,154   |
| (10) xdp_redrect   | 18  | 16          | 45  | 43      | 35  | 18.60%| 523  | 228,101   |
| (11) xdp1_kern/xdp1| 5   | 4           | 72  | 61      | 56  | 8.20% | 472  | 739,416   |
| (12) xdp2_kern/xdp1| 15  | 11          | 93  | 78      | 78  | 71    | 8.97% | 157  | 100,811  |
| (13) xdp_fwd       | 19  | 14          | 170 | 155     | 155 | 128   | 17.42%| 6,157| 2,851,203|
| (14) xdp_pktnctr   | 4   | 3           | 22  | 22      | 22  | 19    | 13.64%| 288  | 614,569  |
| (15) xdp_fw        | 24  | 22          | 85  | 72      | 72  | 65    | 9.72% | 826  | 342,009  |
| (16) xdp_map_access| 6   | 6           | 30  | 30      | 30  | 26    | 13.33%| 27   | 69,628   |
| (17) from-network  | 21  | 18          | 43  | 39      | 39  | 29    | 25.64%| 6,871| 4,312,839|
| (18) recvmsg4      | 4   | 4           | 98  | 94      | 94  | 81    | 13.83%| 3,350| 904,934  |
| (19) xdp-balancer  | 247 | 96          | DNL | 1,811   | 1,771|1,607 | 9.26% | 167,428 | 10,251,406|
| Avg. of all benchmarks | |             |     | 1,811   | 1,771|1,607 | 9.26% | 167,428 | 10,251,406|

### Table 2: Throughput reported as the maximum loss-free forwarding rate (MLFFR) in millions of packets per second per core (§8).
Safety of synthesized programs. We loaded the XDP program outputs produced by K2 into the kernel. All 38 out of the 38 programs found by K2’s search were successfully accepted by the kernel checker, even without K2’s safety post-processing (§6). Table 5 in Appendix F lists the programs we loaded into the kernel.

Benefits of equivalence-checking optimizations. We show the benefits of the optimizations in (§5) to reducing equivalence-checking time and also the number of calls to the solver. Table 4 shows the benefits of optimizations I–IV (memory type, map type, and memory offset concretization, and modular verification) by starting with all optimizations turned on (I, II, III, IV) as the baseline. We progressively turn off each optimization, and show absolute verification times and slowdown relative to the baseline. We find that, across benchmarks, the collective benefits of the optimizations range between 2–7 orders of magnitude, with a mean improvement of 6 orders of magnitude across programs. The larger the program, the more pronounced the impact of the optimizations. Among all the optimizations we apply, modular verification produces the most consistent and significant gains across programs.

Table 5 in Appendix F shows the impact of reductions in the number of queries to the logic solver by caching canonical versions of programs (optimization V, §5). We find caching to be very effective: 93% or more queries otherwise discharged to the solver can be eliminated by caching the equivalence-checking outcomes of syntactically-similar programs checked earlier.

Impact of parameter choices on stochastic search. K2’s stochastic search proceeds in parallel with 16 different sets of parameters. These parameters correspond to variants of the cost functions, with different coefficients used to combine error and performance, as well as different program rewrite rules (§3.2). The full set of values parameterizing each set is described in Appendix F.1. Across 13 programs, we show the efficacy of each set of parameters in optimizing instruction size. Despite K2’s current use of 16 parameter sets, some of those sets are much more likely to produce optimal results than others. Hence, it is possible to obtain K2’s gains with much less parallelism. More generally, exploring the identification of hyperparameters that provide the best results given a limited compute budget is an interesting problem deserving further exploration [100].

Impact of domain-specific rewrite rules. We evaluate the benefits imparted by K2’s domain-specific program rewrite rules (§3.1) to the quality of generated programs. Table 10 in §8 shows the results from optimizing the instruction count of programs with different settings where we selectively turn the domain-specific rules on or off. Each domain-specific rule is necessary to find the best program for each benchmark. Disabling any one of the rules entirely results



We consider 4 offered loads: low (smaller than the slowest throughput of clang), medium (the slowest throughput among clang and K2), high (the highest throughput among clang and K2), and saturating (higher than the fastest throughput of clang or K2).

Table 3: Average latencies (in microseconds) of the best clang and K2 variants at different offered loads (in millions of packets per second).

| Benchmark | Low clang | K2 | Reduction | Medium clang | K2 | Reduction | High clang | K2 | Reduction |
|-----------|-----------|----|-----------|-------------|----|-----------|------------|----|-----------|
| xdp_fwd | 29.148 | 20.676 | 32.98% | 31.309 | 20.321 | 49.98% | 84.450 | 76.929 | 8.91% |
| xdp_ruter_ipv4 | 63.343 | 59.834 | 5.51% | 1.5 | 1.5 | 84.450 | 76.929 | 8.91% | 1.5 |
| xdp_balancer | 33.272 | 30.358 | 5.93% | 3 | 87.291 | 71.649 | 17.92% | 3 | 87.291 | 71.649 | 17.92% |

9 OPTIMIZATIONS DISCOVERED BY K2

We present two classes of optimizations that K2 discovered while reducing the number of instructions in the program. Several more examples from these classes and others are in Appendix G.

Example 1. Coalescing multiple memory operations. In the program xdp_pktcnt[12] developed by Facebook, K2 transformed

```c
bpf_mov r1 0 // r1 = 0
bpf_stx_32 r10 -4 r1 // *(u32*)(r10-4) = r1
bpf_stx_32 r10 -8 r1 // *(u32*)(r10-8) = r1
```

into the single instruction

```c
bpf_st_imm64 r10 -8 0 // *(u64*)(r10-8) = 0
```

driven by coalescing a register assignment and two 32-bit register stores into a single 64-bit store that writes an immediate value. The original instruction sequence comes from two assignments in the C code: u32 cntl_flag_pos = 0; u32 cntr_pos = 0. This example is one of the simplest of this class of optimizations that K2 found. In a couple of cases, K2 shrunk sequences of 12 instructions containing complex swaps of memory contents into 4–8 instructions.

Example 2. Context-dependent optimizations. K2 discovered rewrites that depend on the specific context (e.g., current register values) of instructions within a program. Example, in the balancer_kern program [11] developed by Facebook, K2 transformed the sequence

```c
bpf_mov64 r0 r2 // r0 = r2
bpf_and64 r0 r3 // r0 = r0 & r3
bpf_rsh64 r0 21 // r0 = r0 >> 21
```

into the sequence

```c
bpf_mov32 r0 r2 // r0 = lower32(r2)
bpf_and32 r0 r3 // r0 = r0 & r3
```

This transformation does not generally hold under all values of r3. K2 used the precondition that the value of r3 prior to this sequence was 0x00000000ffe00000. More generally, we found optimizations where K2 leveraged both preconditions and postconditions on the values and liveness of registers and memory addresses.

We believe that the specificity of the optimizations described above (and in Appendix G) may well be challenging to match with a rule-based optimizing compiler. Beyond the categories described above, K2 derived optimizations using complex opcodes (e.g., bpf_xadd64 rX off rY = *(u64*)(rX + off) += rY) and non-trivial dead code elimination that leverages the liveness of memory addresses. More examples are available in Appendix G.
### 10 RELATED WORK

Data plane code optimization has been a topic of recent interest [72, 75, 109]. Chipmunk [75] generates code for high-speed switches, where programs must fit within the available hardware resources or they won’t run at all. In contrast, K2 starts with a program that can run; the goal is to improve its performance safely. In concurrent work, Morpheus [109] explores dynamic recompilation of data plane code based on workload characteristics, reminiscent of conditionally-correct optimization [129]. K2’s approach is orthogonal: it is purely compile-time and the optimizations are valid across all run-time configurations. Farshin’s thesis [72] suggests, but stops short of applying stochastic search to NFVs, due to performance variability. hXDP [52] executes BPF code on FPGAs; K2’s goal is to optimize BPF over ISA-based processors.

There is a rich literature on synthesizing data plane rules and control plane policies for high-speed routers [43, 44, 68, 69, 124, 133, 134]. K2 must synthesize BPF instructions, which are more expressive than router data plane rules and control policies.

K2 builds significantly on the literature on program synthesis [38, 40, 55, 82, 85, 89, 93, 105, 112, 117, 126, 135, 139] and accelerating formal verification [46, 48, 50, 57, 97, 119]. Below, we summarize key three technical differences from this literature.

First, K2 makes several domain-specific contributions in formalizing BPF programs relative to prior work. Sound BPF JITs [114, 115, 137, 138] assert the equivalence between BPF bytecode instructions and lower-level machine instructions on a per-instruction basis. K2 solves a fundamentally different problem: synthesizing new BPF bytecode and checking the equivalence of synthesized and source BPF bytecode. Unlike sound JIT compilers, K2 requires modeling control flow and pointer aliasing which are not concerns for per-instruction verification tasks. Prevail [77] implements a fast abstract interpretation of BPF programs to prove in-bound memory access safety and control flow safety. In contrast to Prevail, K2 performs synthesis, and considers several additional kernel-checker-specific safety properties to generate kernel-executable BPF bytecode. To our knowledge, none of the prior works formalize BPF maps and helpers in sufficient detail to support equivalence-checking, which requires modeling two levels of aliasing (§4.3). Further, K2 contributes several domain-specific techniques to accelerate equivalence-checking by 6 orders of magnitude.

Second, most prior x86 code synthesizers do not handle program safety considerations [40, 112, 117, 126–129]. To our knowledge, the only prior approach to synthesize safe code is the NaCl loop superoptimizer [55], which considers only access alignment (§6).

Finally, K2 includes several domain-specific program rewrites ($3$) that accelerate convergence to better programs.

### 11 CONCLUSION

We presented K2, a compiler for BPF based on program synthesis technology. K2 can produce safe and optimized drop-in replacements for existing BPF bytecode. K2 naturally leads to several avenues for follow-up research. (1) **Scaling to larger programs:** Currently, K2 cannot optimize large programs (200+ instructions) within a short time (e.g., a minute). Developing techniques to optimize large programs quickly is a direction ripe for further research. (2) **Designing better cost functions:** K2’s latency cost function is a weak predictor of actual latency. The design of high-fidelity cost functions to statically estimate program performance metrics such as tail latency and maximum per-core throughput will help boost the throughput and latency gains available from synthesis. (3) **Addressing engineering challenges:** The active evolution of the BPF ecosystem [26] makes it challenging to keep K2’s safety checks in sync with that of the kernel checker and to develop support for emerging BPF hooks and helper functions.

We hope that the community will build on our compiler and the techniques in this paper. K2’s source code, including all of our experimental scripts, is available at https://k2.cs.rutgers.edu/. *

This work does not raise any ethical issues.

### ACKNOWLEDGMENTS

This work was funded by the National Science Foundation grants CNS-1910796 and CNS-2008048. We thank the SIGCOMM anonymous reviewers, our shepherd Nate Foster, Akshay Narayan, Paul Chaignon, and Vibhaalakshmi Sivaraman for their thoughtful feedback and discussions. We are grateful to Nikolaj Bjørner for improvements to the Z3 solver that helped support K2’s requirements.

### REFERENCES

[1] 1999. RFC 2544: Benchmarking Methodology for Network Interconnect Devices. [Online. Retrieved Jan 27, 2021.] https://tools.ietf.org/html/rfc2544.

[2] 2014. BPF kernel interpreter. [Online. Retrieved Jun 21, 2021.] https://github.com/torvalds/linux/blob/master/kernel/bpf/core.c#L1356.

[3] 2016. Some notes on verifier complexity. [Online. Retrieved Jul 12, 2021.] https://github.com/cilium/cilium/commits/8f7c6767180a9920b10c9646945f2970daad6e6.
Sythesizing Safe and Efficient Kernel Extensions for Packet Processing

David Menendez and Santosh Nagarakte. 2017. Alive-infer: Data-driven pre-condition inference for peephole optimizations inllvm. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 49–63.

Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2021. Dynamic Recompilation of Software Network Services with Morphus. arXiv:2106.08833 [cs.NI].

Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University. http://cs.au.dk/~am011er/spa/.

Michael K. Smith, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation. 395–404.

Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-based Pruning for Speeding up Superoptimization. In ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University. http://cs.au.dk/~am011er/spa/.

Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishna, and Operating Systems Principles. 2013. Jitk: A trustworthy in-kernel interpreter infrastructure. In Proceedings of the 2013 ACM SIGPLAN conference on Programming language design and implementation. 225–242.

Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specifications and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In USENIX Operating Systems Design and Implementation (OSDI). 572–585.

Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishna, Thomas Anderson, and Timothy Roescoe. 2014. Arzakis: The Operating System is the Control Plane. In USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhumjar. 2016. Scaling superoptimization. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. 297–310.

Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hrisy, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. [Online. Retrieved Oct 29, 2020.] http://www.cis.upenn.edu/bcpierce/scf/current/index.html.

SorawePornchareonwae, James Bornholt, and Emina Torlak. 2020. Fixing Code that Explodes Under Symbolic Evaluation. In International Conference on Verification, Model Checking, and Abstract Interpretation. Springer. 44–67.

Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, Anirudh Sivaraman. 2021. The K2 compiler. [Online. Retrieved June 30, 2021.] https://k2.cs.rutgers.edu.

Quentin Monnet. 2020. Optimizing BPF: Smaller Programs for Greater Performance. [Online. Retrieved Jan 20, 2021.] https://www.netronome.com/blog/optimizing-bpf-smaller-programs-greater-performance/.

Luigi Rizzi. 2012. netmap: A Novel Framework for Fast Packet I/O. In 2012 USENIX Annual Technical Conference (USENIX ATC 12).

Hugo Sadok, Zhipeng Zhao, Valerie Chong, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. 2021. We need kernel interposition over the network dataplane. In Proceedings of the Workshop on Hot Topics in Operating Systems. 152–158.

Shambwadiya Saha, Santhosh Prabhu, and P Madhusudan. 2015. NetGEn: Synthesizing data-plane configurations for network policies. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research. 1–6.

Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. 2001. Beyond Softnet. In Annual Linux Showcase & Conference, Vol. 5. 18–18.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. CoRR abs/1711.04422 (2017). arXiv:1711.04422 http://arxiv.org/abs/1711.04422

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. ACM SIGARCH Computer Architecture News 41, 1 (2013), 305–316.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. 391–406.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally correct superoptimization. ACM SIGPLAN Notices 50, 10 (2015), 147–162.

Nikita V. Shirokov. 2018. XDP: 1.5 years in production. Evolution and lessons learned. http://vger.kernel.org/lpc_net2018_talks/LPC_XDP_Shirokov_v2.pdf. In Linux Plumbers Conference.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Elcioglu. 2005. Programming by sketching for bit-streaming programs. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. 281–294.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. 404–415.

Kausik Subramanian, Loris D’Antoni, and Aditya Akella. 2017. Genetic Synthesizing forwarding tables in multi-tenant networks. In Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages. 572–585.

Kausik Subramanian, Loris D’Antoni, and Aditya Akella. 2018. Synthesis of Fault-Tolerant Distributed Router Configurations. Proc. ACM Meas. Anal. Comput. Syst. 2, 1 (April 2018), 26 pages. https://doi.org/10.1145/3179425

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 264–276.

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In Proceedings of the 2013 ACM international symposium on New ideas, paradigms, and reflections on programming & software. 135–152.

Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In International Conference on Computer Aided Verification. Springer. 564–586.

Xi Wang, David Latzar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A trustworthy in-kernel interpreter infrastructure. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 33–47.

Yichen Yang, Phitchaya Mangpo Phothilimthana, Yusu Remy Wang, Max Willers, Sudip Roy, and Jacques Piemar. 2021. Equality Saturation for Tensor Graph Superoptimization. arXiv:2101.01352 [cs.AI].
Appendices are supporting material that has not been peer-reviewed.

A APPROACHES TO PROGRAM SYNTHESIS

In this paper, we are given a sequence of instructions in a fixed instruction set, i.e., a BPF bytecode source program. We are interested in generating an alternative sequence of instructions, i.e., a synthesized program, that satisfies the specification that (i) the synthesized program is equivalent to the source program in its input-output behaviors, (ii) the synthesized program is safe, and (iii) the synthesized program is more efficient than the source program. The precise meanings of efficiency and safety in the BPF context are described in §2.3 and §6.

To simplify the following discussion, suppose the program specification is simply (i) above, i.e., $\text{spec} \models p_{\text{synth}}(x) \iff p_{\text{src}}(x)$ for source program $p_{\text{src}}$ and synthesized program $p_{\text{synth}}$ for all program inputs $x$. At a high level, the program synthesis problem we are interested in can be formulated as the logical query

$$\exists p. \forall x. p(x) \iff p_{\text{src}}(x) \quad (4)$$

where $p$ is any program composed of instructions from the BPF instruction set. As written down, this problem is a quantified boolean formula (QBF) with alternating quantifiers, which does not permit efficient decision procedures for problems that arise in synthesis [131]. Hence, program synthesizers take the approach of counterexample guided inductive synthesis (CEGIS) [131, 132]. First, a candidate program $p_{\text{cand}}$ is determined through a search procedure. Then, the synthesizer checks whether the candidate satisfies the specification, by asking

$$\exists x. p_{\text{cand}}(x) \not\models p_{\text{src}}(x) \quad (5)$$

for the fixed program $p_{\text{cand}}$.

Typically, synthesis algorithms use test cases to quickly prune candidates that do not satisfy the specification. If test cases do not eliminate $p_{\text{cand}}$ as a candidate satisfying the specification, the query above can usually be formulated in a first-order logic theory [51] which permits efficient decision procedures. The query above is an example of equivalence-checking, which determines whether two programs produce the same output on all inputs. If the query (5) is satisfiable, we get a counterexample, which can be added to the set of test cases to prune the same or similar programs in the future without discharging computationally-expensive logic queries. If (5) is unsatisfiable, we have found a program $p_{\text{cand}}$ that produces the same output as $p_{\text{src}}$ on all inputs, and hence meets the specification.

The synthesis approaches in the literature differ mainly in the search procedures they use to propose candidate programs $p$. There are broadly four search approaches. Enumerative search (e.g., [40, 105, 117]) searches the space in order from smallest to largest programs, terminating the search with the smallest program that satisfies the specification. Rule-based search [93, 135, 139] uses targeted rewrite rules to transform the source program into another program that satisfies the specification yet is more optimal. Deductive search [38, 82, 85, 89, 112, 126] encodes the search into a deductive query discharged to a solver whose solution implies a program within a specified grammar. Stochastic search [55, 127] searches the space guided by cost functions that enable sampling new random programs. Further, there are search algorithms that combine the best of multiple approaches [84, 117].

Among the synthesis approaches above, in our view, stochastic synthesis is the easiest to generalize to new and diverse contexts, due to its ability to support very expressive cost functions and constraints. We adopt and adapt stochastic synthesis in this paper.

B VERIFICATION CONDITIONS FOR BPF MAPS AND HELPER FUNCTIONS

BPF maps are a special kind of memory. Like memory, a BPF program could read a map with a specific key (look-up) and write a value corresponding to an existing key (update). However, three aspects make BPF maps very different from memory from the formalization perspective. First, the BPF map API [95] requires input keys in the form of pointers to memory, and further, returns values that are themselves pointers to memory. Second, keys can be deleted in a BPF map, unlike addresses in memory. Third, the keys and values in BPF maps are persistent structures that exist before and after program execution. Together, these aspects of BPF maps prevent us from directly applying the same formalization methods as memory access (§4.2) or other existing axiomatizations (e.g., partial maps [118]) to model BPF maps in first-order logic. None of the prior works on formalizing BPF [77, 114, 115, 137] handle any aspect of BPF maps.

In the rest of this subsection, we show how our compiler handles pointer inputs/outputs as well as deletions. We show how we address the last aspect (map persistence) while performing program equivalence checking (§4).

B.1 Handling pointer access to map memory

Supplying an input key to a map operation (e.g., look-up) as a pointer creates two levels of aliasing behaviors: First, two distinct registers may point to the same memory address (same as regular pointer aliasing). Second, a map needs to return the same value given the same (value of) key, even if the keys are supplied from distinct memory addresses.

We formalize map access in bit vector theory by decomposing the two levels of aliasing as follows. First, we create a new symbolic variable for each key supplied to the map API and each pointer value it returns. We call these variables the valuations of the (input) key pointer and the (output) value pointer. We apply the formulas in §4.2 to the key pointer, where the data that is read is the key’s valuation.

To handle aliasing among the valuations of the keys themselves, we write down formulas analogous to memory aliasing (§4.2) over the valuations of the key and the value. This entails maintaining map write and read tables analogous to memory read and write tables.

Addressing aliasing among key valuations has the added benefit of encoding the partial map axioms [118], e.g., a look-up of a key following an update of that same key returns the value from that update. No guarantees are made over the returned value pointer itself (this is a reference to the kernel’s memory) except that it is non-zero.

Handling map manipulations through memory-access instructions. BPF map values can be manipulated directly through memory-access instructions, e.g., to increment a counter stored as a map
value using a `bpf_xadd` instruction, which has the impact of performing `*ptr = *ptr + reg`. K2 uses the static analysis described in §5 to determine the memory region corresponding to the pointer being loaded or stored. The memory read and write tables (§4.2) ensure that K2 puts down the right formulas for subsequent operations over these map value memory regions.

B.2 Handling map deletions
Keys in a BPF map can be deleted, unlike memory locations. If a deleted key is subsequently looked up, the result is a null pointer. We model this by the simple trick of setting the value address corresponding to the valuation of a deleted key to 0 in the map write table. When another key with the same valuation is looked up, the returned value address is 0, indicating to the program that the key does not exist in the map. Luckily, this return value coincides with BPF’s semantics for the return value of map lookup, which is 0 whenever the key does not exist in the map. We also handle setting the right return value for the BPF map delete API call, which provides distinct return values depending on whether the key currently exists in the map.

B.3 Handling multiple maps
The discussion so far assumes the existence of a single map with all operations over this map. BPF programs may look-up several maps in one program by loading map descriptors. However, the set of all possible maps accessed by the program is known at compile time. Hence, we can handle the general case by prefixing the precondition `map_id == K` as the head of an implication, with the body of the implication being the formula generated using the methods described above.

B.4 Limitation of map model
The current modeling of maps ignores concurrency among threads (kernel or user) accessing the same map. K2 does not make formal guarantees about the equivalence of its output programs to input programs when running in multi-threaded contexts where data races are permissible. However, BPF provides mutual exclusion for map contents through BPF spinlocks [61], and for code that enforces mutual exclusion to map access, the semantics of K2’s output programs are indeed equivalent to those of the input program under concurrent access.

B.5 Other kernel helper functions
In addition to map helper functions, we also modeled a number of other BPF kernel helper functions, including functions to get random numbers, obtain a Unix timestamp, adjust the headroom in a memory buffer containing a packet, get the processor ID on which the program is executing, among others. The list of all BPF helper functions numbers over a hundred (and growing [95]). Out of engineering expediency, we only formalized the actions of helper calls as we needed them, rather than support all of them.

C DETAILS ON OPTIMIZATIONS TO EQUIVALENCE-CHECKING
The techniques described in §4 are sufficient to produce a working version of a BPF bytecode compiler. However, we found that the time to compile a single program of even modest size (~100 BPF instructions) is intractably large. The underlying reason is the significant time required to perform equivalence checking, which runs to several hours on even just a single pair of BPF programs on a modern server (§8).

This section presents domain-specific optimizations that bring down this verification time by several orders of magnitude to a few milliseconds. We show the impact of each optimization in §8. Several features of the BPF instruction set and packet-processing programs contribute to increased equivalence checking time: aliasing in memory and map access, multiple types of memories (stack, packet, metadata, and so on), usage of multiple maps, encoding the semantics of helper functions, and the existence of control flow within programs. K2 incorporates two broad approaches to reduce verification time: concretization of symbolic terms in the formula (§C.1) and modular verification of smaller parts of the program (§C.2) to verify the larger formula from §4.

C.1 Concretizations of Symbolic Terms
It is well known that an explosion in the size and solving difficulty of a formula under symbolic evaluation can be mitigated by assigning specific values to, i.e., concretizing, terms in the formula [46, 50, 57, 97, 119].

We leverage the unique characteristics and simplicity of packet-processing BPF programs, as well as the feasibility of constructing simple static analyses (e.g., sound inference of types) within the K2 compiler, to infer concrete values of several terms within the equivalence-checking formulas dispatched to the solver. These concretizations are “best effort” in the sense that we do not need the analysis within K2 to be complete: we apply the simplifications where possible, falling back to the general versions in §4 where they are not applicable. We eschewed complicated alias relationship mining techniques [55, 101] in favor of simpler domain-specific ones.

To simplify the discussion, we describe our techniques for straight-line programs (no branches) first, and then show how they generalize to loop-free programs with control flow.

I. Memory type concretization. BPF programs use multiple memory regions: the stack, packet, maps, and various kernel data structures like sockets. The handling of pointer-based aliasing of memory access discussed in §4.2 uses a single write table and read table for all of memory.

Instead, K2 leverages multiple read/write tables, one corresponding to each distinct memory region. This is feasible to do since any reference to a specific memory originates from pre-defined inputs to the BPF program [88], such as the R10 register (stack), an input register like R1 (packet or packet metadata), or return values from specific helper calls (map memory, kernel socket structure, etc.). The type of each pointer in a program can then be easily inferred through a simple information-flow static analysis. For example, given a code sequence `bpf_mov r1 r10; bpf_mov r2 r1`, it is straightforward to infer that `r2` is pointing into the stack memory. K2 tracks types for all registers across each instruction in this manner.

The benefit of maintaining separate tables for separate memories is that the size of aliasing formulas in a memory read reduces from
\(O(N^2_{\text{all mem types}})\) to \(\sum_{i \in \text{mem types}} O(N^2_i)\), where \(N_{(i)}\) refers to the number of accesses to memory of a specific type (or all types).

**II. Memory offset concretization.** The exact memory address contained in a register during a BPF program’s execution is in general hard to determine at compile time, since the program stack or the packet contents may be located anywhere within the kernel’s memory. However, in many packet-processing programs, the offsets relative to the base address of the memory region, e.g., such as the offset (from the beginning of the packet) of a header field, are known at program compile time. Further, BPF stack locations can only be read after they are written, and it is feasible to track which concrete offsets are being written to in the first place. K2 attempts to maintain a concrete offset into the memory region for each register known to be a pointer. Each subsequent operation on the register is associated with the corresponding operation on the concrete offset. For example, the instruction `bpf_mov r1 r10`; `bpf_sub r1 2 /* r1 := r10 - 2 */` associates the offset -2 with the register r1. A subsequent operation `bpf_mov r2 r1`; `bpf_sub r2 4` would also update the corresponding offset of r2 to -6. (Both registers have a STACK memory type.)

Offset concretization has the effect of turning a clause like `addr_i == addr_j` (§4.2) into a clause like `offset_i == offset_j`, where `offset_i` and `offset_j` are concrete values, e.g., `-2 == -6`. When a load instruction results in comparing a register offset to other offsets stored in the write table, any comparison of a concrete pair of offsets can be evaluated at compile time without even a solver call. If all pairs of offsets compared are concrete, this can simplify an entire memory read into a single clause of the form `value_i == value_j`. Even if only one of the offsets being compared is known concretely, it simplifies the clause overall.

In the limit, this optimization has the effect of turning the entire memory into nameable locations akin to registers. Similar ideas have been applied in prior efforts to scale verification time [40, 127]. K2 applies this idea in a best-effort manner, falling back to general aliasing when concrete offsets cannot be soundly inferred.

**III. Map concretization.** BPF programs access several maps and frequently perform operations such as lookups, which are among the most expensive BPF operations in terms of verification. Map lookups generate large verification conditions because of two levels of aliasing (§4.3). Accessing multiple maps in a single BPF program further explodes the formula sizes, since one set of clauses is generated for each possible map the program may have accessed. Similar to the idea of concretizing memory types for each access, K2 statically determines the map to which a given lookup or update occurs. This is feasible due to a property of BPF’s map-related instructions: there is a unique opcode `LD_MAP_ID` which loads a unique and concrete `map_id` (obtained from a file descriptor), which is then supplied as a parameter to each BPF map call.

We did not generalize our approach to concretize the keys that are looked up or values that are presented to the map update function call. The BPF map API (§2) mandates map function parameters and return values to be pointers to keys and values, rather than registers containing the keys and values themselves. Map keys are pointers to the stack, and the keys on the stack are often from input memory regions, such as packet fields. As noted above (cf. memory offset concretization), we concretize memory addresses that are accessed, but not the bits stored at concrete addresses in memory. If the keys are symbolic, then the values are symbolic, too. We leave leveraging opportunities to concretize map keys and values to future work.

**Incorporating control flow into optimizations.** So far, the discussion of the static analysis in K2 to determine concrete terms has focused on straight-line code. In the presence of branches, this analysis is generalized as follows. First, statically known information (e.g., pointer type, offset) is annotated with the basic block and path condition under which it is determined. Next, for a subsequent read, to determine the concrete information (e.g., offset) to apply, we use the following procedure iteratively for each prior write to the pointer, starting from the latest to the earliest in order of program execution:

- If the last write is from a basic block that dominates [63] the basic block of the reading instruction, we use the corresponding entry’s concrete information and stop. SSA dominance analysis is performed just once per program.
- If the last write is from a basic block from which control never reaches the basic block of the reading instruction, we skip the corresponding entry’s concrete information and move to the next entry. Reachability analysis is performed just once per program.
- If neither of the above is true, there is a path through the basic block of the write to the current instruction, but there are also other paths. K2 conjoints the clause path condition of the entry \(\Rightarrow\) `read_offset == entry offset`. The process continues to the next entry.

For example, after processing the instruction sequence

\[
\text{jneq } r5 r6 1 /\* \text{ if } r5 == r6 /\* \\
b1: \text{bpf_mov } r1 r10 -2 /\* \text{ } r1 = r10 - 2 /\* \\
b2: \text{bpf_mov } r1 r10 -4 /\* \text{ else } r1 = r10 - 4 /\* \\
b3: \text{bpf_load_32 } r3 r1 /\* \text{ } r3 = (\text{uint32}) r1 /\* \\
\]

the compiler stores the following information:

\[
\begin{align*}
\text{block b1, cond } r5 & = r6, r1 \text{ STACK, offset } -2 \\
\text{block b2, cond } (r5 & = r6), r1 \text{ STACK, offset } -4
\end{align*}
\]

To determine the offset read by block b3 (\(r3 = r1\)), the compiler determines that neither b1 nor b2 dominates b3, but they can both reach b3, hence producing the verification conditions

\[
(r5 == r6) \Rightarrow \text{read_offset } == -2 \land \\
!(r5 == r6) \Rightarrow \text{read_offset } == -4
\]

It is not always possible to make the `read_offset` concrete, as seen in this example.

**IV. Caching** We also cache the verification outcomes of canonicalized versions of a given program to quickly determine if a structurally similar program was equivalence-checked earlier. We canonicalize the program by removing dead code. Then, the canonicalized program is hashed into a key that is used to look-up a program cache of verification outcomes. Equivalence-checking is only performed when there is a cache miss; the outcome of the equivalence checking is inserted into the cache along with the program that was checked.
C.2 Modular Verification

Synthesis and optimization typically scale to large programs by operating over a sequence of smaller windows, where a window is a contiguous sequence of instructions as they appear in the text of the program [102, 108, 117, 129]. Small windows (rather than the full programs) lead to smaller verification conditions, which are much faster to solve.

Verifying the correctness of synthesized windows be thought of in one of two ways. A peephole optimizer [40, 107] would only produce instruction sequences that exhibit equivalent behavior under any values that the window consumes (precondition) and ensure any values it produces are equal to that produced by the original program (postcondition). This is similar to the verification conditions from §4, written just for a window in the program.

An alternative possibility is to consider stronger preconditions and weaker postconditions to verify the window [117, 129]. This enables stronger optimizations: for example, the instruction bpf_mul r2 r1 can be optimized into bpf_ishift r2 2 provided a compiler can infer that r1 = 4 during the multiplication, but not in general. Similarly, other optimizations become possible with weaker postconditions.

We call the part of the program before the window the prefix and the part after the window the postfix. Automatically inferring strong preconditions (that hold after the prefix) and weak postconditions (that must hold before the postfix) is a challenging problem [102, 108, 129]. Recent work [117] takes the approach of using the strongest possible precondition and weakest possible postcondition, by constructing the entire first-order logical representation of the prefix and postfix programs. This approach, while enabling strong optimizations, reduces to using the verification conditions for the full program.

Instead, we build on an earlier approach [40] that uses the live variables [110] to generate stronger preconditions and weaker postconditions than peephole optimizers. Additionally, K2 also infers sets of concrete values for variables, if they exist, and pose those as preconditions for the optimization. K2 chooses windows among basic blocks of a certain maximum instruction count and uses the following window-based verification condition:

variables live into window 1

=⇒ variables live into window 2
∧ inferred concrete valuations of variables
∧ input-output behavior of window 1
∧ input-output behavior of window 2
⇒ variables live out of window 1
≠ variables live out of window 2

Here, variable set “live into” the window contains all variables written in the prefix that are readable in the window, and the set “live out of” the window contains all variables that are written inside the window and read by the postfix. The concrete valuations of variables are inferred through static analysis that determines the set of concrete values that a register might take on, along with the corresponding path conditions. For example, in the code sequence

```c
if r0 > 4:
    r1 = 6
else if r2 < 6:
    r1 = 10
```

else:
    r1 = 8

the clause of the precondition with inferred values is

\[ p1 \Rightarrow r1 = 6 \]
\[ \land p2 \Rightarrow r1 = 10 \]
\[ \land p3 \Rightarrow r1 = 8 \]
\[ \land \text{exactly one of } p1, p2, p3 \text{ is true} \]

where \( p1, p2, \) and \( p3 \) are fresh boolean variables corresponding to the path conditions where \( r1 \) takes on distinct concrete values.

If the formula is satisfiable, the solver produces an input state at the beginning of the window that results in the two windows producing different output states at the end of the two window executions. If not, the two windows are conditionally-equivalent under the specific precondition and postcondition above.

We specialize the liveness analysis to the BPF context by handling BPF registers as well as BPF memory (stack, packet, map values). We build on optimizations (I–III) in §C.1 and track liveness information along with concrete types and offsets. In particular, with map values, we treat each map value as a distinct type of memory, with its own concretizations of each access within the window, since we do not have concrete offsets into the kernel’s (value) memory.

Window-based verification condition generation and equivalence checking are particularly useful in the BPF context for many reasons. First, it becomes unnecessary to track aliasing for map access, since each window typically just contains one live map value pointer, and we only care about aliasing within the value memory itself. In fact, we don’t even use any information about the key in the verification condition. Second, it becomes possible to handle several new helpers in a much more expedient way with window verification, since we can optimize “around” the function by simply tracking the inputs and outputs of the helper, rather than “across” the function, which requires capturing its input-output semantics. Third, window verification allows us to guide the stochastic search better for programs that output a small number of output bits—programs known to be hard to optimize with typical error functions [127]. This is because K2’s window postcondition ensures that the window obtains other intermediate values in the program correctly, rather than (just) the output.

The fundamental disadvantage of window-based verification relative to full-program verification is its inability to discover some strong optimizations that would be found with equivalence-checking full programs [117]. Unlike window-based equivalence checking, full program checking can detect aliasing conditions over the entire program, optimize across helper functions (with full semantics), and can optimize instructions under the strongest possible preconditions and weakest possible postconditions.

D MORE DETAILS ON THE IMPLEMENTATION OF K2

Please see §7 for an overview of K2’s implementation.

K2 uses the same internal BPF instruction structure as the kernel, so that K2 can consume BPF instructions directly from the binary
Table 5: We loaded 38 K2-optimized program variants (known to be equivalent to the corresponding source programs) into the kernel. All programs were successfully accepted by the kernel checker.

| Benchmark         | # Variants produced | # accepted by kernel checker | Cause(s) of checker failure |
|-------------------|---------------------|-----------------------------|-----------------------------|
| xdp1              | 5                   | 5                           | -                           |
| xdp2              | 5                   | 5                           | -                           |
| xdp_redirect      | 5                   | 5                           | -                           |
| xdp_map_access    | 5                   | 5                           | -                           |
| xdp_router_ipv4   | 5                   | 5                           | -                           |
| xdp_pktcnnr       | 3                   | 3                           | -                           |
| xdp_fwi           | 5                   | 5                           | -                           |
| xdp_fw            | 5                   | 5                           | -                           |

Table 6: The benefit of caching (§3) in reducing the number of solver calls. The benchmark numbers correspond to those in Table 1.

| Benchmark | # progs. hit cache | total # calls to the cache | Hit rate | # iters. |
|-----------|---------------------|-----------------------------|----------|----------|
| (1)       | 13,479              | 14,470                      | 93%      | 1,000,000|
| (2)       | 32,706              | 35,072                      | 93%      | 1,500,000|
| (3)       | 30,557              | 31,833                      | 96%      | 4,000,000|
| (4)       | 68,828              | 72,220                      | 95%      | 4,000,000|
| (14)      | 15,827              | 16,552                      | 96%      | 2,000,000|
| (17)      | 134,505             | 140,686                     | 96%      | 5,000,000|
| (18)      | 100,641             | 109,046                     | 92%      | 5,000,000|

ELF format of the instructions (i.e., the .o files) without explicitly decoding from and encoding into another format. Binary encode/decode is known to be a significant source of compiler bugs [111].

The compiler consumes inputs from pre-compiled BPF bytecode object files as well as instruction sequences encoded through the kernel bpf_insn data structure. The extraction of the inputs from ELF binaries uses libbpf, the kernel's standard library to work with BPF bytecode. A tricky aspect of BPF bytecode is that the text section of the ELF file is not directly executable. The text section must be processed using relocations [99] during the loading process [113] to ensure that run-time references to map file descriptors and other symbols are updated before execution (the text section is independent of these run-time parameters). K2 consumes instructions from relocated ELF.

The output of the compiler can take on two forms: a sequence of binary instructions, which is useful to test simple programs quickly, or a patched ELF object file that contains the sections of the original ELF input with the optimized instruction sequence patched in lieu of the original instructions in the text section. K2 uses pyelftools to perform the patching.

K2 uses relocation metadata, specifically the set of instructions that are touched by the libbpf loader, to ensure that the linkages between the text section and the remaining ELF sections are unmodified. Hence, outputs from K2 can serve as drop-in replacements to existing BPF object files.

K2 includes a high-performance BPF interpreter that runs BPF bytecode instructions using an optimized jumptable implementation akin to the kernel’s internal BPF interpreter [2]. Using C preprocessor directives, we share code for most instructions between the interpreter and the equivalence checker. Before adopting this design, we found that the mismatch between the behaviors of the interpreter and the semantics of the first-order logic formulation of the same program was a significant and vexing source of bugs. We found that using a shared semantic description of each instruction that is common to the interpreter and the first-order logic verification condition generator was helpful in ensuring that the input-output behaviors of the interpreter and the program’s formalization were compatible.

We construct first-order logic representations using Z3 [65] and also use it as our solver. To reduce the impact of large memory-bound queries on the overall compile time, we use two separate Z3 solver processes, with the query discharged to the solvers using a serialized smtlib2 format. We pick the solver that returns a response first and respawn server processes after a fixed number of discharged queries.

K2’s interpreter and the equivalence-checking query formulation can distinguish different BPF program types and fix the inputs and outputs appropriately.

E ESTIMATED PERFORMANCE FROM K2
In this section, we show the performance estimated by K2 with the latency goal. We perform latency and throughput measurements of the top-k programs from these experiments in §8.

F ADDITIONAL EVALUATION RESULTS
F.1 Parameters of stochastic search
Table 8 shows the best-performing parameter settings of K2. Table 9 shows K2’s instruction count reductions from those parameter settings. Table 10 shows the benefits of K2’s domain-specific rules in stochastic synthesis.

F.2 Efficacy of safety checks
The results of loading the outputs of K2 using the kernel checker are shown in Table 5.

F.3 Caching efficacy
The efficacy of caching in reducing the number of solver queries to perform equivalence checking is illustrated in Table 6.

F.4 Benefits of domain-specific rewrite rules
In Table 10, we show the benefits of including domain-specific program rewrite rules (§3.1) while generating proposals within the stochastic optimization loop.

G MORE OPTIMIZATIONS DISCOVERED BY K2
Table 11 lists several optimization case studies across benchmarks from the Linux kernel, hXDP, and Cilium.
### Table 7: Improvements in estimated performance of programs according to K2.

| Benchmark          | Program Runtime (sec) | When lowest perf cost prog. is found | Time (sec) | Iterations |
|--------------------|-----------------------|--------------------------------------|------------|------------|
|                    | -01                   | -02/-03                              | K2         | Gain       |
| (9) xdp_router_ipv4| 57.45                 | 50.34                                | 47.21      | 6.22%      | 1,455      | 450,502   |
| (10) xdp_redirect  | 17.15                 | 16.08                                | 14.52      | 9.70%      | 472        | 203,332   |
| (11) xdp1_kern/xdp1| 31.33                 | 25.57                                | 24.55      | 3.99%      | 155        | 240,116   |
| (12) xdp2_kern/xdp1| 34.79                 | 30.88                                | 28.86      | 6.54%      | 145        | 107,089   |
| (13) xdp_fwd       | 57.02                 | 51.56                                | 43.73      | 15.19%     | 4,316      | 2,850,176 |
| (14) xdp_pktcntnr  | 12.32                 | 12.32                                | 11.85      | 3.81%      | 48         | 201,057   |
| (15) xdp_fw        | 65.02                 | 46.61                                | 45.01      | 3.43%      | 858        | 345,320   |
| (16) xdp_map_access| 27.96                 | 27.96                                | 27.28      | 2.43%      | 20         | 50,814    |
| (17) from-network  | 32.27                 | 30.96                                | 29.17      | 5.78%      | 713        | 401,730   |
| (18) recvmag4      | 68.12                 | 68.12                                | 63.83      | 6.30%      | 3,444      | 907,834   |
| (19) xdp-balancer  | DNL                   | 760.12                               | 724.73     | 4.66%      | 170,154    | 7,502,210 |
| Avg. of all benchmarks |                   |                                       |            |            | 6.19%      | 16,525    | 1,205,471 |

Table 9: Performance of K2 under different parameter settings.
Table 10: Improvements in program compactness under different proposal generation settings (§3.1). We consider turning the following proposal-generation rewrite-rules on or off: $MEM_1$ implements a type 1 memory exchange, sampling to replace all non-pointer operands, $MEM_2$ implements a type-2 memory exchange, sampling to replace only the memory operation width, and $CON T$ replaces $k = 2$ contiguous instructions. Instruction counts with the * mark indicate that they are the minimal found among all the proposal generation settings tested.

H PROFILES OF PROGRAM LATENCY VS. OFFERED LOAD

These results supplement the numbers in §8 regarding the latency, throughput, and drop rates of various XDP programs as offered load from the traffic generator increases.
Table 11: A catalog of optimizations found by K2.

| Benchmark | BPF program before optimization | BPF program after optimization | C code | Note |
|-----------|---------------------------------|--------------------------------|--------|------|
| xdp2_kern/xdp1 | `bpf_load_16 r1 r7 0` | `bpf_load_32 r1 r7 0` | `src[0] = p[0];` | This instruction sequence swaps three higher bytes and three lower bytes through six 8-bit loads and stores. K2 coalesced six loads and stores into two 16-bit loads and stores and one 8-bit load and store. |
|          | `bpf_load_16 r2 r7 6` | `bpf_load_32 r2 r7 6` | `dst[0] = p[0];` |
|          | `bpf_load_16 r3 r7 2` | `bpf_load_32 r3 r7 2` | `dst[0] = p[0];` |
|          | `bpf_stx_16 r7 0 r2` | `bpf_stx_32 r7 0 r2` | `dst[0] = p[0];` |
|          | `bpf_load_16 r2 r7 10` | `bpf_load_32 r2 r7 10` | `dst[0] = p[0];` |
|          | `bpf_load_16 r3 r7 4` | `bpf_load_32 r3 r7 4` | `dst[0] = p[0];` |
|          | `bpf_stx_16 r7 8 r3` | `bpf_stx_32 r7 8 r3` | `dst[0] = p[0];` |
|          | `bpf_load_16 r2 r7 6` | `bpf_load_32 r2 r7 6` | `dst[0] = p[0];` |
|          | `bpf_load_16 r3 r7 2` | `bpf_load_32 r3 r7 2` | `dst[0] = p[0];` |
| xdp_fwd | `bpf_load_16 r1 r10 -2` | `bpf_load_32 r1 r10 -2` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_stx_8 r7 4 r1` | `bpf_stx_32 r7 4 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_rsh64 r1 8` | `bpf_rsh64 r1 8` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_stx_8 r7 2 r1` | `bpf_stx_32 r7 2 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_load_16 r1 r10 -6` | `bpf_load_32 r1 r10 -6` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_stx_16 r7 0 r1` | `bpf_stx_32 r7 0 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
| sys_enter_open | `bpf_load_32 r1 r0 24` | `bpf_load_32 r1 r0 24` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_add64 1 1` | `bpf_xadd_32 r0 0` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_stx_32 r10 -16 r1` | `bpf_stx_32 r10 -16 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_load_32 r10 -26 r7` | `bpf_load_32 r10 -26 r7` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_load_32 r10 -16` | `bpf_load_32 r10 -16` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_stx_32 r10 -28 r1` | `bpf_stx_32 r10 -28 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_load_32 r10 -26` | `bpf_load_32 r10 -26` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_move4 r1 1` | `bpf_xadd_32 r0 0` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_xadd_32 r1 0` | `bpf_xadd_32 r1 0` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_xadd_32 r0 0` | `bpf_xadd_32 r0 0` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_xadd_32 r10 -10 r1` | `bpf_xadd_32 r10 -10 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_xadd_32 r10 -20 r1` | `bpf_xadd_32 r10 -20 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |
|          | `bpf_xadd_32 r10 -30 r1` | `bpf_xadd_32 r10 -30 r1` | `memcpy(eth->h_dest, fib_params.dmac, ETH_ALEN);` |

Figure 2: Throughput, average latency and drop rate under different offered loads. We measured the top-\(k\) K2 versions (\(k = 9\) for xdp-balancer and \(k = 5\) for other benchmarks), and two clang versions: -O1 and -O2. Note that -O2 and -O3 are identical for these benchmarks.