Applying program synthesis to optimize eBPF

Qiongwen Xu

Rutgers University

qx51@cs.rutgers.edu

1 Problem statement

Extended Berkeley Packet Filter (eBPF) [8, 1] has emerged as a powerful method to extend packet-processing functionality in the Linux operating system. Users can write an eBPF program and attach it in the kernel at specific hooks (e.g., network device driver [5]) to process packets. To ensure safe execution (e.g., crash-free) of a user-developed eBPF program in kernel context, Linux uses an in-kernel eBPF verifier. An eBPF program is allowed to execute only if it is proved safe by the eBPF verifier.

However, developing high-performance eBPF programs is not easy because every optimization must respect the eBPF verifier's intricate safety rules. Even small performance optimizations to eBPF code must be meticulously hand-crafted by expert developers. The optimization support in compilers is inadequate. For all the benchmarks tested in our experiments, Clang-9 produced the identical eBPF bytecode under optimization flags `-O2` and `-O3`.

Our goal is that for a given loop-free eBPF program, synthesize a semantically equivalent eBPF program that can be accepted by the eBPF verifier and has better performance in terms of program length, throughput, and latency. An eBPF program has multiple inputs: a register $r1$ pointing to packet data; a register $r10$ pointing to a scratch memory, referred to as the BPF stack; and memory shared between user space and kernel space, in the form of key-value maps. An eBPF program has multiple outputs: a register $r0$ containing the return value, packet, and key-value maps.

In this report, firstly we introduce and discuss several program synthesis works (§2). Then we present our work of applying program synthesis to optimize eBPF programs with several domain-specific techniques (§3).

2 Related work

2.1 Overview

In this report, program synthesis is to synthesize a program satisfying the specification that (a) the synthesized program is semantically equivalent to the source program and (b) other requirements such as low latency. Program synthesis problem can be formulated in Eq. (1).

$$\exists p. \forall x. p(x) == p_{src}(x) \wedge f(p) \tag{1}$$

where p is the program to be synthesized, p_{src} is the source program, x is the program input, and $f(p)$ is the logic that the synthesized program satisfies other requirements.

To simplify the discussion, we will first discuss how to solve the partial problem of Eq. (1) (*i.e.*, the equivalence requirement) using Eq. (2).

$$\exists p. \forall x. p(x) == p_{src}(x) \tag{2}$$

This formula is a Quantified Boolean Formula (QBF) satisfiability problem with an existential quantifier and a universal quantifier, which is not efficient to be solved [14]. A technique

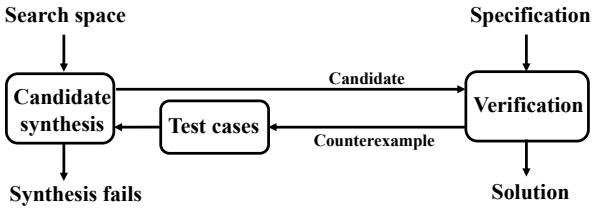

Figure 1: Counterexample-guided inductive synthesis. Candidate synthesis searches for a candidate passing all test cases. Verification proves the equivalence of the candidate and produces a counterexample if the proof fails.

called counterexample-guided inductive synthesis (CEGIS) [14] (Fig. 1) was proposed to improve efficiency. CEGIS simplifies the formula into two parts: candidate synthesis and verification (Fig. 1). Candidate synthesis is driven by the test cases (*i.e.*, program inputs). The initial test cases are provided by users or randomly generated. Candidate synthesis tries to synthesize a candidate which passes all existing test cases (Eq. (3)).

$$\exists p. \forall x \in tc. p(x) == p_{src}(x) \tag{3}$$

where tc is test cases. Then, the verification does an equivalence check of the candidate by checking whether there is an input (*i.e., counterexample*) that makes the candidate and the source program produce different outputs (Eq. (4)).

$$\exists x. p(x) ! = p_{src}(x) \tag{4}$$

If Eq. (4) is unsatisfiable, the candidate is the solution to the program synthesis problem. Otherwise, a counterexample will be generated and added in the test cases. The counterexample can help the candidate synthesizer avoids synthesizing a program that has a similar behavior to the current candidate. In the candidate synthesis, if time is exhausted or no more candidates can be generated, the synthesis fails.

Generally, verification (*i.e.*, Eq. (4)) is formulated as a query which can be solved by the off-the-shelf SAT/SMT solver. The solver will give a counterexample if the query is satisfiable. There are several different approaches to synthesizing candidates. In this report, we will describe (a) stochastic synthesis which utilizes Markov Chain Monte Carlo (MCMC) method to do a stochastic search (§2.2), (b) enumerative synthesis with pruning algorithms (§2.3), (c) constraint-based synthesis which leverages off-the-shelf SAT/SMT solvers to search for a candidate (§2.4), and (d) others (§2.5). Most of these approaches utilize test cases to quickly prune unequal programs which cannot be candidates, reducing the number of time-consuming queries in the verification.

2.2 Stochastic synthesis

Stochastic synthesis was proposed in [12], which uses a stochastic search technique known as MCMC method to synthesize programs. In §2.2.1, we provide an overview of the basic approach, and in §2.2.2, we show how to extend this to loops.

2.2.1 STOKE

STOKE [12] is a framework which uses stochastic search for loop-free x86 instructions superoptimization [7]. The optimized program is semantically equivalent to the source program and has lower latency.

The stochastic search in STOKE is guided by a cost function $c(.)$ which reflects the correctness and the estimated latency, *i.e.*,

$$c = w_e * c_{err} + w_p * c_{perf} \tag{5}$$

where w_e is the weight of error cost c_{err}, w_p is the weight of performance (*i.e.*, latency) cost c_{perf}. Thus, the program optimization problem can be converted to searching for a program with the minimal cost.

In general, the cost function is complex and highly irregular. STOKE utilizes MCMC to find out a program with a low cost (near-optimal) in a reasonable time. One property of MCMC is that in the limit of sampling time, the frequency of the same point sampled by the MCMC method is in direct proportion to a probability density function value. Hence, programs with lower costs are more likely to

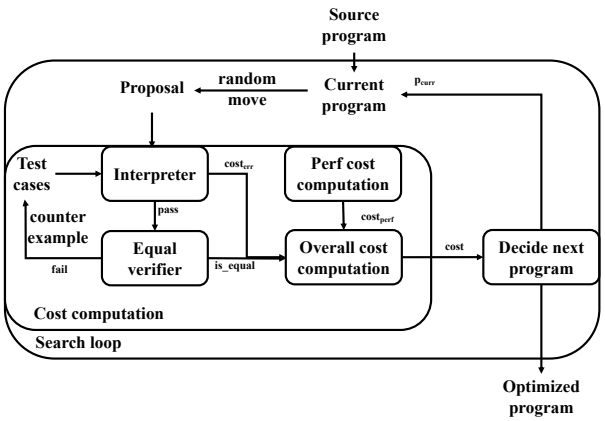

Figure 2: An overview of STOKE

be sampled by designing a probability density function that is inversely related to the cost function. The optimized program is the program with zero error cost and minimal performance cost among all sampled programs.

Fig. 2 is an overview of STOKE using MCMC. The search loop starts from the source program, performs the stochastic search for a configurable time, and produces an optimized program. Each search loop iteration includes the following steps, noting that the first current program is the source program.

1. Assuming the current program is P, perform a random move to get a new proposal P^*. A random move is a random modification to the current program, such as modifying an instruction `move r1 0` to `move r1 2`.
2. Calculate the acceptance criteria α of moving from the current program P to the proposal P^* using Eq. (6), where $c(.)$ is the cost function, S is the source program. β is a configurable constant. Usually, it is a small number like 2.

$$\alpha(P \to P^*; S) = min(1, \frac{e^{-\beta \cdot c(P^*;S)}}{e^{-\beta \cdot c(P;S)}}) \quad (6)$$

3. Perform a uniform(0,1) sample to get a random value v.
4. Decide the next program by comparing v with α. If v is smaller than α, the new proposal P^* will be accepted as the next current program. Otherwise, the next current program is P. According to Eq. (6), if the proposal has a smaller cost than the current program, the proposal will be accepted. Otherwise, it may be rejected.

The cost function is a combination of the correctness cost and the performance cost. Intuitively, the correctness cost is either 0 if the proposal is equal to the source program or 1 if the proposal is unequal. However, the 0/1 output makes the search space uneven. Instead, STOKE utilizes test cases to measure the distance (*e.g.*, the Hamming distance) between outputs of the proposal and the source program. In addition, the test cases also reduce the equivalence verification time by reducing the solver queries (CEGIS discussion in §2.1). The performance cost is estimated by the sum of the average latencies of each instruction, *i.e.*,

$$\begin{aligned} c_{perf}(P;S) = &\sum_{inst \in P} LATENCY(inst) \\ &- \sum_{inst \in S} LATENCY(inst) \end{aligned} \quad (7)$$

There are three key aspects in MCMC sampling to increase the probability of finding a better program: the starting program, moves, and the cost function. In practice, we run the search for a reasonable time. The search can not sample all possible programs (*i.e.*, it's an incomplete search). Hence there is no guarantee that the search can find the optimal program. If the starting program is "close" to the optimal program, it is more likely that the search can find the best program. The transforming path from the starting program to the optimal program depends on the move and whether to accept this move based on the cost function. This work mentions that it's more likely to get the best results when the moves contain both minor and major changes. Minor changes such as modifying an operand can help find out the local optimal, while major changes like swapping two random instructions can help the search jump from the local optimal area to the global optimal region.

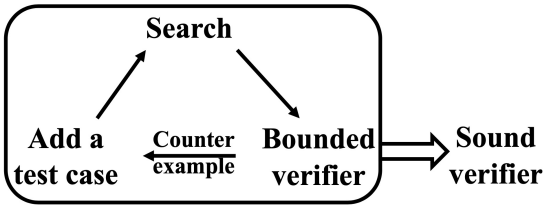

Figure 3: Search with two-stage verification using bounded verifier and sound verifier.

2.2.2 Synthesizing programs with loops

Churchill et al [2] proposed a work on how to synthesize loops based on STOKE.

To support loop synthesis, we need a way to automatically prove the equivalence of two loops. In general, we need to manually construct loop invariants to prove the equivalence of two programs with loops [16]. A previous work [13] proposed data-driven equivalence checking (DDEC) to do the equivalence check automatically. However, DDEC cannot generate counterexamples from failed proofs, making it uneasy to use test cases in STOKE to quickly detect most unequal rewrites. Manually generating test cases is an option. However, as a consequence, the efficiency of the stochastic search heavily relies on the quality of human-provided test cases. If people miss some crucial corner cases, DDEC which involves time-consuming queries will be invoked more frequently.

The solution proposed in this work is to utilize a two-stage verification consisting of a bounded verifier and a sound verifier (Fig. 3).

In the first stage, a bounded verifier partially proves equivalence by checking when there are at most *k* iterations for each loop, whether the proposal is semantically equivalent to the source program. The bound parameter *k* is configured by users. If the proof fails, a counterexample will be generated. Otherwise, this proposal will be sent to the second stage and checked the equivalence for all inputs by the sound verifier.

The key idea of the partial proof in the bounded verifier is to reconstruct loop-free programs by unrolling loops based on *k*, and then to perform the equivalence check of programs without loops. Firstly, decompose the program into a set of basic blocks. Secondly, based on the bound parameter *k*, construct paths (i.e., programs with unrolled loops) where basic blocks in each loop repeat for at most *k* times, and corresponding path conditions. Thirdly, formulate program logic on each path: $output_i = \bigwedge_j inst_j(s_j)$, where $inst_j$ is the j^{th} instruction on the $path_i$ and s_j is the initial state for $path_i$. By combining each path formula for $path_i$ and the corresponding path condition pc_i, we can get the program logic p when iterations are at most *k* times: $p = \bigwedge_i pc_i \rightarrow (output == output_i)$, where $output$ is program's output. Finally, we can prove whether the program logic of proposal P^* is equivalent to the source program S by leveraging an SMT solver to check whether the query $p_{P^*} \wedge p_S \wedge output_{P^*} != output_S$ is unsatisfiable.

The sound verifier is based on DDEC [13]. First, DDEC utilizes test cases to guess a simulation relation which consists of cutpoints and invariants between the proposal and the source program. A cutpoint is a pair of program points where one is from the proposal, the other is from the source program. Each point is associated with an invariant, *i.e.*, the state relationship between two programs. The cutpoints cut each program into some loop-free fragments. Second, DDEC utilizes test cases to infer a set of corresponding code paths where two programs start from the same cutpoint and end at the same cutpoint. Third, DDEC formulates a relationship between the input and output states for each corresponding code path. Finally, DDEC constructs a query to be solved by an SMT solver: for every corresponding code path, if two program starts from the same states, two programs will end at the same states. The sound verifier extends DDEC to improve efficiency and adds support to memory instructions for precision.

2.2.3 Discussion

We use STOKE framework to synthesize eBPF programs. There are two reasons. First, STOKE can easily generalize to the eBPF domain by providing the cost function of the specification: safety, performance, and correctness. Second, eBPF has a

large and high dimensional program search space, since eBPF supports more than 100 opcodes, 64-bit operands, and 16-bit memory and jump offsets. STOKE can quickly explore a large and irregular search space by sampling programs with lower costs more frequently.

eBPF supports bounded loops. The program length (hence, the loop length and the maximum loop iterations) is bounded by the number of instructions on each path in a program. In the modern kernel, the maximum number of all path instructions is one million. To support the bounded loops of eBPF synthesis in future work, we can refer to the two-stage verification model.

2.3 Enumerative synthesis

Enumerative algorithms search for a semantically equivalent program satisfying the specification by enumerating all possible programs from the smallest programs to larger programs. Intuitively, this method can find the optimal program (*e.g.*, with the smallest program length). However, the search space containing all possible programs grows exponentially with an increasing number of opcodes, operands, instructions, and so on. Hence, it is crucial to design a pruning algorithm which can dramatically reduce the search space.

2.3.1 LENS

To speed up superoptimization, Phothilimthana et al [10] proposed a pruning algorithm LENS to accelerate synthesizing loop-free and branch-free programs. LENS solves the inefficiencies in the prior equivalence-class-based enumerative synthesis by selectively refining the search space where programs pass all previous test cases (*i.e.*, programs have the potential to be optimal).

LENS is based on equivalence class pruning technology. Equivalence-class-based work groups programs in the search space according to their behaviors on the test cases. We will use Fig. 4 as an example. Assume pre is the program prefix from the initial state s to the intermediate state u. The search algorithm looks for a program postfix $post$ that can convert u to t, where t is the final state of the source program for the test case(s). If it is proved that there is no such $post$, all program prefixes in the same equivalence class (*i.e.*, the prefixes that convert the initial state to the same intermediate state for the test case(s)) can be pruned away. For example, in Fig. 4, we figure out that program prefix p_1 ($\langle inst1, inst2\rangle$) cannot reach target state t_1. After proving that p_2 ($\langle inst2, inst3\rangle$) reaches u_1, we can prune away programs with prefix p_2, because p_2 and p_1 have the same intermediate state u_1 (*i.e.*, they are in the same equivalence class).

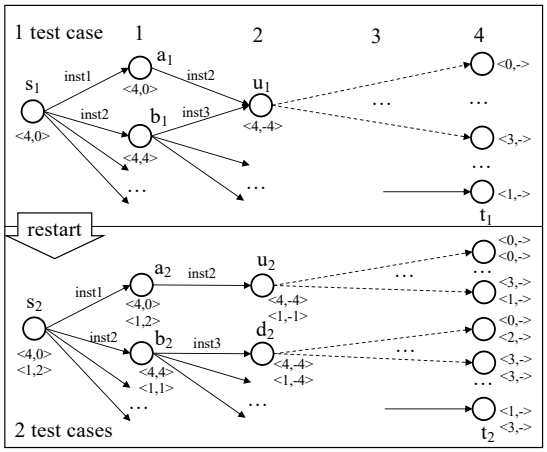

Figure 4: The existing equivalence class pruning strategy.

There are two inefficiencies in this pruning algorithm. First, if an unequal program passes all existing test cases, a counterexample will be produced from an SMT solver. To handle this new test case, the search restarts but it does not utilize the information that some programs are pruned away according to the previous test cases. In Fig. 4, the search revisits p_1 and p_2 on the second test case. Second, the algorithm uses more test cases than necessary. In Fig. 4, p_1 and p_2 behave the same on the first test case but differ on the second one. If p_1 fails the first test case, p_2 can be pruned away. However, since p_1 and p_2 have the different behaviors on the second test case (hence, in different equivalence classes), p_2 cannot be pruned away. The paths from d_2 to the finial states t_2 are visited.

LENS was designed to address these two inefficiencies. When the search processes a new test case, LENS further refines the selective search space where the programs pass all previous test cases. This method rapidly prunes away all programs that fail any previous test cases. For example, p_1 and p_2 will be removed from the search space before starting the search with two test cases.

Fig. 5 shows an example of LENS algorithm. In step 1, LENS checks whether the leaf node s_1 can reach the final node t_1 within one instruction. If it is infeasible, LENS expands the graph by adding one instruction to each leaf node (*i.e.*, s_1), and check whether any new leaf node can reach t_1 with one instruction (step 2). With the expansion in step 2, all leaf nodes cannot reach t_1 in one instruction, LENS further expands the leaf nodes (step 3). After this expansion, b_2 is able to reach t_1 by adding one more instruction. This means that programs $progs_{t_1}$ with the prefix reaching state b_2 and postfix $inst1$ can pass the first test case. Then we need to further check whether $progs_{t_1}$ can pass the second test case. Before checking, LENS refines the search space by only keeping $progs_{t_1}$ (step 4). Then in step 5, LENS builds the new graph by interpreting $progs_{t_1}$ with the second test case and further refines the search space for the next test case in step 6. Once there is a program passing all the existing test cases, it will be sent to the verifier to do the formal equivalence proof. The verifier will produce a counterexample if the proof fails.

2.3.2 Dataflow-based pruning

Souper [11] synthesizes peephole optimization rules (*e.g.*, optimizing $x * 4$ to $x \ll 2$) by enumerating a large number of candidates. These candidates may contain symbolic constants or Holes. A Hole represents an arbitrary DAG of not-yet-enumerated instructions. The search space includes both concrete values and partially symbolic candidates (from Holes). Souper starts with a Hole, expands the Hole in a tree structure until the solution is found or reaches a certain depth.

Recent work by Mukherjee et al [9] shows how to improve Souper by leveraging dataflow to refine

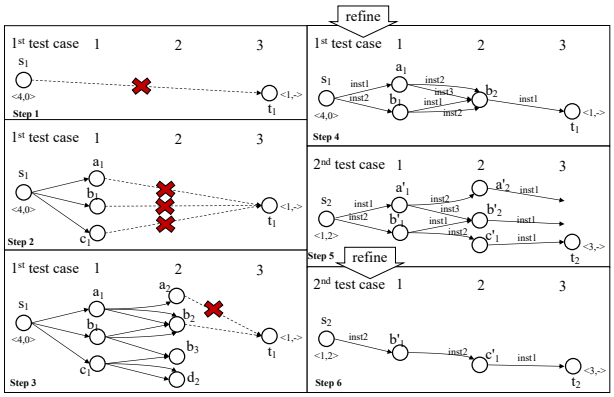

Figure 5: Selective pruning strategy.

search space. If a partially symbolic candidate is proved to be infeasible, the sub-tree rooted in this candidate can be pruned.

The key idea of proving a candidate is infeasible is to check whether there is a conflict between the specification and the candidate. This work uses dataflow analysis to search for conflicts. One approach is to assign the input a concrete value and infer known bits in the output. A known bit is 0, 1, or unknown, which can be inferred by a forward dataflow analysis. For example, if the specification is $4x + 1$ and the candidate is $H \ll 1$, where H is a Hole, and the input is specified as 1, we can figure out one conflict is in the least significant bit between the specification output (*i.e.*, 1) and the candidate (*i.e.*, 0). There are other approaches in the paper, such as specifying the input and inferring the output range to find out a conflict.

2.3.3 Discussion

LENS is an efficient pruning algorithm for enumerative synthesis. However, it is not obvious how to generalize LENS to support programs with branches and utilize the safety guarantee required by eBPF programs to prune the search space.

Using dataflow to prune the space of programs is more suitable to the search space stored in a tree structure, while an eBPF program is an instruction sequence. It is an open question how to use dataflow

to refine the search space in eBPF synthesis.

2.4 Constraint-based synthesis

Constraint-based synthesis is to encode the candidate synthesis (Eq. (3)) into constraints and then utilize an off-the-shelf SAT/SMT solver to solve the constraints. SKETCH (§2.4.1) and Brahma (§2.4.2) are constraint-based synthesizers utilizing bit-vector theory.

2.4.1 SKETCH

SKETCH [14] is a synthesizer that takes the specification and a sketch (*i.e.*, a partial program with some holes) as inputs. Users can express their high-level ideas in this partial program, leaving the tricky parts (*e.g.*, uncertain constants) as holes. SKETCH then solves the holes or shows the sketch is buggy if there is no solution.

SKETCH converts the synthesis of a program, which translates m bits (input) to n bits(output), into a Quantified Boolean Formula satisfiability problem (QBF). Logically, the sketch S can be completed if there exists a control c (*i.e.*, the concrete values for holes) to make the sketch semantically equivalent to the specification (*i.e.*, if Eq. (8) is satisfiable).

$$\exists c \in \{0,1\}^k . \forall x \in \{0,1\}^m . P(x) == S(x,c) \quad (8)$$

where x is the input. We will use the following specification and the sketch as an example.

```
int16 spec(int16 x) {
  return (x & 0xFFF0) << 4;
}
int16 p (int16 x) implements spec() {
  return x << ??; // ?? is a hole
}
```

According to Eq. (8), the formula is

$\exists c \in \{0,1\}^{16} . \forall x \in \{0,1\}^{16} .$
`(x & 0xFFF0) << 4 == x << c`

A model called counterexample-guided inductive synthesis (CEGIS) (§2.1) was proposed in SKETCH to reduce the complexity of (Eq. (8)). In the candidate synthesis stage, the synthesis solver is utilized to figure out a control using the following query (Eq. (9)).

$$\exists c \in \{0,1\}^k . \forall x \in tc . P(x) == S(x,c) \quad (9)$$

where tc is the test cases. For the above example, if the test cases contain $0x1234$ and $0x1111$, then the formula is

$\exists c \in \{0,1\}^{16} .$ `(x == 0x1234 ∨ x == 0x1111) ∧`
`(x & 0xFFF0) << 4 == x << c`

2.4.2 Brahma

Brahma [4] uses constraint-based synthesis to solve component-based problem: synthesize loop-free programs using components from a library for a specific functionality. Users specify the functionalities of the desired program and the library components through logic relations between inputs and outputs. Any component in the library can only be used at most once. The main contribution of Brahma is a novel algorithm of generating the constraint in first-order logic and solving the constraint.

In the constraint generation phase, Brahma utilizes variable location connections to connect the components from the library:

$$\exists L . \forall \vec{I}, O, T .$$
$$c_{lib} \wedge c_{conn}(\vec{I}, O, T, L) \wedge c_{wfp}(L) \implies c_{spec}(\vec{I}, L) \quad (10)$$

1. $\vec{I}$ is the program input, O is the program output.
2. T is the temporary variables in the programs.
3. L contains the location variables (program input or line number) of inputs and output of each component.
4. c_{lib} is the constraint of the relations of each component's inputs and output variables (Eq. (11)).

$$c_{lib} = \bigwedge_i c_i(\vec{I_i}, O_i) \quad (11)$$

where c_i is the logic relation between component inputs $\vec{I_i}$ and output O_i.

5. c_{conn} is the constraint of the order of each components in the program and the line location of each component input (Eq. (12)).

$$c_{conn} = \bigwedge_{x,y \in \{I,O\} \cup T} (l_x == l_y \implies x == y) \tag{12}$$

6. c_{wfp} is the constraint of a well-formed program. It contains:
   a. location range constraint: assume a program has N_{in} input variables and N_{inst} instructions, component input location should be in range $[0, M-1]$, output location in range $[N_{in}, M-1]$, where $M = N_{in} + N_{inst}$, noting that every program input is regarded as a (pseudo) component output.
   b. consistency constraint: every program instruction contains at most one component.
   c. acyclicity constraint: every variable should be initialized before use.
7. c_{spec} is the desired relationship between program input $\vec{I}$ and output O.

The above synthesis constraint (Eq. (10)) is a $\exists\forall$ formula, which is hard to be solved in a reasonable time. To solve this, Brahma relies on CEGIS (§2.1) to utilize two SMT solvers (synthesis solver and verification solver). In the candidate synthesis stage, Brahma tries to find a solution for L that works for all test cases by an SMT solver.

2.4.3 Discussion

SKETCH and Brahma are two constraint-based synthesizers. They are efficient in solving nontrivial programs. There are four reasons why we don't use them to optimize eBPF programs.

1. Performance: neither of them can directly reason about the performance of the program to be synthesized.
2. Development efficiency: besides the specification, both SKETCH and Brahma require users to provide a partial program or a library.
3. Scalability of opcodes: the size of the SKETCH constraint could potentially grow exponentially in the number of opcodes, while eBPF supports more than 100 opcodes.
4. Opcode usage bounded: the library in Brahma bounds the number of times the same opcode can be used. It can easily be bounded for hardware with resource limitations, while the same opcode can be used as many times as users want in an eBPF program. We can provide N_{inst} (the maximum number of instructions in a program) opcode copies in the library, but it will make the synthesis space much larger.

2.5 Others

2.5.1 Equality saturation

Traditional compilers use peephole optimization rules (*e.g.*, optimizing $x * 0$ to 0) to optimize programs. The optimizations are applied in sequence. For example, in Fig. 6, a compiler produces a transformation path $a \to b \to c$. However, the order of optimizations to run can affect the qualify of generated programs. The fixed order may prevent compilers from producing a better program (*e.g.*, d in Fig. 6).

Equality saturation [15] was designed to solve this problem. It is an optimization technique which intends to find the best program over a space of programs semantically equivalent to the source program (*i.e.*, all programs in Fig. 6). The idea is that the transformation from one program to another program using an optimization rule does not change the program input-output behavior, *i.e.*, two programs are equal. If we can use optimization rules to find out all equal programs of the source program, the best program must be a program among these programs.

Equality saturation first builds a space of equal program by repeatedly running the following steps: (a) stores the source program in the equal space, (b) applies optimization rules to the program fragments from the equal space, and (c) stores the new equal versions in the existing equal space until it is saturated (*i.e.*, no more unexplored programs). Then, equality saturation selects the best program among these versions.

To make the optimization efficient and effective, equality saturation utilizes an intermediate repre-

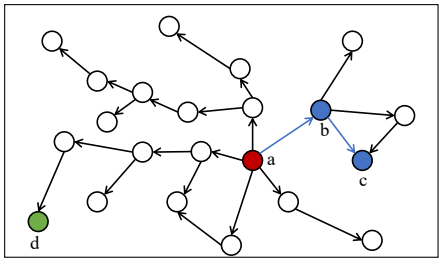

Figure 6: A space of equal programs. Each node is a program, and each arrow is a transformation. a is the source program. c is the program generated by a traditional compiler. d is the best program.

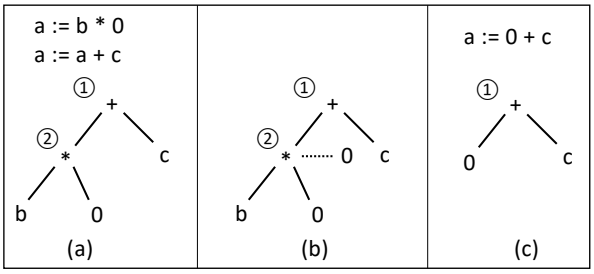

Figure 7: An example of PEGs: (a) the source code and its PEG, (b) the E-PEG, and (c) the optimized code and its PEG, which results by choosing node 1 from (b).

sentation, Equivalence Program Expression Graph (E-PEG), to encode *multiple* equivalent programs into a single data structure. Fig. 7 shows an example of how to utilize E-PEG to optimize a program. Fig. 7(a) is the source program presented by Program Expression Graph (PEG) where each node is an operator, and the edges associated with a node represents where the arguments come from for the node's operator. For this source program, we can apply an optimization rule, *i.e.*, x * 0 = 0, and then the operator * and its affiliated arguments (b and 0) can be replaced by a single '0'. This optimization is encoded with the source program as shown in Fig. 7(b). Equality saturation uses a dashed edge to connect * with 0, which represents the subprogram leading by "*" can be replaced by "0" (*i.e.*, replacing b*0 by 0). Finally, based on the E-PEG, equality saturation can use some searching algorithms to calculate the optimal program, *i.e.*, Fig. 7(c).

This approach eliminates the effect of applying optimizations in a different order, addressing the phase-ordering problem. In addition, it allows designing global profitability heuristics algorithm without worrying about determining whether to accept an optimization which could affect future optimizations.

2.5.2 STNG

STNG [6] utilizes program synthesis to lift stencil computations from low-level Fortran code with nested loops to a summary in high-level predicate language. Stencil computations are a class of algorithms that update each element in a multidimensional grid based on the values of the element's nearest-neighbors using a function (*e.g.*, Fig. 8(b)). The summary in STNG is a postcondition (Fig. 8(a)), *i.e.*, a predicate that will be true after executing the stencil code. To ensure correctness, we need to prove the Fortran code and the summary are semantically equivalent, *i.e.*, the stencil code is valid with respect to the precondition and postcondition.

Since the stencil code has loops, one approach for equivalence check requires using loop invariants which in general is manually constructed to construct a verification condition [16]. If the verification condition (Eq. (13)) is true, it implies the stencil code is equivalent to the postcondition.

$$VC(s, post, invariant) = \forall s.vc_1 \wedge vc_2 \wedge vc_3 \tag{13}$$

where s is the state of inputs, $invariant$ is the loop invariants, $post$ is the postcondition to be verified. vc_1(Eq. (14)) is that if the precondtion is true, then the loop invariant is also true.

$$pre(s) \implies invariant(s) \tag{14}$$

vc_2 (Eq. (15)) is that if the loop invariants is true and the loop condition is true, then after updating state by executing the loop body once, the loop invariants is hold for the updated states.

$$invariant(s) \wedge cond(s) \implies invariant(body(s)) \tag{15}$$

```
procedure sten(imin,imax,jmin,jmax,a,b)
  real (kind=8), dimension(imin:imax,jmin:jmax) :: a
  real (kind=8), dimension(imin:imax,jmin:jmax) :: b
  do j=jmin, jmax
    t = b(imin, j)
    do i=imin+1, imax
      q = b(i,j)
      a(i,j) = q + t
      t = q
    enddo
  enddo
end procedure
```

(a)

```
post(a) ≡ ∀imin+1 ≤ i ≤ imax, jmin ≤ j ≤ jmax.
    a(i,j) = b(i-1,j) + b(i,j)
```

(b)

```
invariant(a, b, j) ≡ j ≤ jmax + 1 ∧
    ∀imin+1 ≤ i ≤ imax, jmin ≤ j'< j.
    a(i,j') = b(i-1,j') + b(i,j')
```

(c)

Figure 8: An example of (a) Fortran code, (b) the corresponding summary (*i.e.*, postcondition) and (c) loop invariants

vc_3(Eq. (16)) is that if the loop invarian is true while the loop condition is false, then the postcondition is true.

$$invariant(s) \wedge \neg cond(s) \implies post(s) \quad (16)$$

Fig. 9 is an example of how to construct VC. An SMT solver is used for checking the satisfiability of the verification condition formula VC. If it is unsatisfiable, the postcondition is equivalent to the stencil code. Otherwise, it is not equivalent.

The main challenge to prove the equivalence of loops is how to get the loop invariants and postcondition automatically, *i.e.*, find out loop invariants and postcondition such that the verification condition holds (Eq. (17)). (18)).

$$\exists post, invariant. \forall s. vc_1 \wedge vc_2 \wedge vc_3. \quad (17)$$

STNG utilizes synthesis with CEGIS (§2.1) model to search for the loop invariants and postconditions. In the synthesis stage, STNG synthesizes the loop invariants and postcondition candidate passing test cases, *i.e.*, state of loop inputs (Eq. (18)).

$$\exists post, invariant. \forall s \in tc. vc_1 \wedge vc_2 \wedge vc_3 \quad (18)$$

```
x = c;
y = 0;
while (x > 0) {
   x--;
   y++;
}
```

(a)

precondition	x = c ∧ y = 0
loop invariant	∀ x ≥ 0. x + y = c
loop condition	x > 0
postcondition	y = c

(b)

Figure 9: An example of how to construct the verification condition: (a) the loop code; (b) the corresponding precondition, loop invariant, loop condition, and postcondition.

where tc is the test cases. In the verification stage, STNG uses an SMT solver to verify whether the loop invariants and postcondition candidate satisfies VC or not. If not, a counterexample will be added into the test cases.

To efficiently synthesize the postcondition and loop invariants, STNG utilizes several techniques to narrow down the search space. We will introduce inductive template generation and quantifier elimination with partial Skolemization.

Inductive template generation STNG analyses the stencil code to infer the overall structure of the stencil code and uses this structure as a constraint on the postcondition and the loop invariants. This approach has two steps.

1. Symbolic Execution: STNG sets loop bounds and array sizes to small, random concrete values, and sets all other inputs such as array elements to symbolic values. Then it interprets the stencil code using symbolic execution. Finally, we will get output arrays where the value of each element is a formula consisting of concrete and symbolic values. Taking Fig. 8 as an example, the value of output $a[2,2]$ is a symbolic formula $b[1,2] + b[2,2]$, and the value of $a[3,4]$ is $b[2,4] + b[3,4]$.
2. Template Generation: Since the template should capture all input-output behaviors, STNG infers the template by searching for the intersection (*i.e.*, the common part) of all output expressions. We denote each expression in the symbolic representation of each output array value by e, where if e is a non-terminal, it is an operator op which consumes input ex-

pressions $e_1, e_2, \cdots e_n$. The following is the way to compute the intersection of two expressions.

$$\sqcap(e_1, e_2) := \begin{cases} e_1 & e_1 = e_2, leaf(e_1) \\ (op\{\sqcap(e_{1i}, e_{2i})\})_i & e_1 = (op\{e_{1i}\}_i) \\ & e_2 = (op\{e_{2i}\}_i) \\ MakeHole(e_1, e_2) & otherwise \end{cases}$$

where the expression $leaf(e_1)$ means e_1 is a leaf in the expression tree. For example, we can compute the intersection of $b[1,2]+b[2,2]$ and $b[2,4]+b[3,4]$ as follows:

$$\begin{aligned} &\sqcap\,(b[1,2]+b[2,2], b[2,4]+b[3,4]) \\ &\quad (apply\ \sqcap\,(e_1,e_2) = (op\{\sqcap(e_{1i},e_{2i})\})_i) \\ &= b[\sqcap(e_{11}, e_{21})] + b[\sqcap(e_{12}, e_{22})] \\ &\quad (apply\ \sqcap\,(e_1,e_2) = MakeHole(e_1,e_2)) \\ &= b[h_1()] + b[h_2()] \end{aligned}$$

where $e_{11} = 1, 2$, $e_{21} = 2, 4$, $e_{12} = 2, 2$, $e_{22} = 3, 4$, and $h_1()$, $h_2()$ are two holes, which need to be synthesized. This structure $b[h_1()]+b[h_2()]$ encodes that the each element in the output array a is the sum of two distinct elements in array b.

Quantifier elimination with partial Skolemization This technique is used in the verification stage to efficiently detect an incorrect candidate postcondition and loop invariants and produces a counterexample. As we discussed earlier, in the verification stage, we need to check whether the candidate postcondition and loop invariants satisfies the verification condition for all possible input states. To simplify the discussion here, we only consider the third verification condition (*i.e.*, vc_3 (Eq. (16))). If we can find out an input state (*i.e.*, counterexample) such that vc_3 is invalid, we prove the candidate postcondition and loop invariants is incorrect, *i.e.*, $\exists s, \neg(invariant(s) \wedge \neg cond(s) \implies post(s))$, which can be simplified to Eq. (19).

$$\exists s, invariant(s) \vee \neg cond(s) \vee \neg post(s) \quad (19)$$

However, the presence of the additional universal quantifier in the loop invariants makes the verification formula (Eq. (19)) hard to be solved by an SMT solver. For example, in Fig. 8(c), inside the loop invariants, $invariant(a,b,j) = \forall i \in [imin+1, imax].j' \in [jmin, j).P(i, j', j, jmax, a, b)$, where P is the remaining part in the $invariant(a,b,j)$. The verification formula is Eq. (21). If the candidate is incorrect, the formula is satisfiable.

$$\begin{aligned} &\exists a, b, j, imin, imax, jmin, jmax. \\ &\forall i \in [imin+1, imax], j' \in [jmin, j). \\ &P(i, j', j, jmax, a, b) \end{aligned} \quad (20)$$

Since there is a $\forall$, it is hard to be solved by an SMT solver. One approach to address this issue is utilizing Skolemization [17]: figure out a function $f_i(imin, imax)$ and $f_{j'}(jmin, j)$ to replace i and j'. Then Eq. (21) becomes

$$\begin{aligned} &\exists a, b, j, imin, imax, jmin, jmax. \\ &P(f_i, f_{j'}, j, jmax, a, b) \end{aligned} \quad (21)$$

However, in general, it is hard to find out f_i and $f_{j'}$ that works for *all* possible values of i and j'. STNG searches for a partial Skolem function $f_S(jmin, j)$ working for *some* values of i and j with some concrete inputs to capture some incorrect loop invariants and postconditions. For example, if we only use two inputs, the first input is $imin = 0, imax = 1, jmin = 2, j = 3$, then $i = 1, j' = 2$, the second input is $imin = 1, imax = 2, jmin = 3, j = 4$, then $i = 2$, $j' = 3$. Eq. (21) can be simplified as $\exists a, b, jmax.P(1, 2, 3, jmax, a, b) \vee P(2, 3, 4, jmax, a, b)$.

3 Our work: K2

We design and implement K2 [18], a program-synthesis-based compiler that automatically optimizes BPF bytecode with formal correctness and safety guarantees. K2 leverages STOKE framework (§2.2.1) to satisfy the complex specification. The cost function in K2 is a combination of correctness, safety, and performance.

Our experiments show that 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 production systems.

K2 incorporates several domain-specific techniques to make synthesis practical by accelerating equivalence-checking of eBPF programs by 6 orders of magnitude. In addition, we propose a way to perform a safety check.

3.1 Fast equivalence check

To do the equivalence check, K2 encodes the program's behavior in first order logic and uses an off-the-shelf SMT solver Z3 [3] to solve the following query.

```
inputs to program 1 == inputs to program 2
 ∧  input-output behavior of program 1
 ∧  input-output behavior of program 2
 ⇒   outputs of program 1
      != outputs of program 2
```

The program behavior formalization includes arithmetic and logic instructions, memory access, eBPF maps, and helper functions.

The equivalence checking time is crucial since the check is in the search loop. We develop two domain-specific techniques to reduce the equivalence checking time by 6 orders of magnitude to make K2 capable of synthesizing programs from real-world systems. The key idea of the techniques is to simplify the equivalence-check formula.

The first method is to concretize the symbolic variables. This can be achieved by a static analysis throughout the eBPF program. For example, K2 can infer (a) the memory type, which can be stack, packet, or map memory; (b) map type, *i.e.*, which map the instruction accesses; and (c) memory offset, *e.g.*, the offset to the first element in the memory such as stack[0].

Our second method, modular verification, does an equivalence check of two *windows* (*i.e.*, a small sequence of instructions in the program) instead of two entire programs. This method is efficient in synthesizing large programs. K2 selects and optimizes a window in the program at a time. K2 repeats the window optimization until we run out of the compiler's time budget, or we find that there is no improvement in each window from the last search throughout the program. We will use Fig. 10 as an example to illustrate modular verification. In Fig. 10, two programs are semantically equivalent. Thus, the modular verification is supposed to prove that the two windows are equal. To achieve this, K2 first infers the variables two windows need to agree on. These variables are variables live out of windows (*i.e.*, $r3$) instead of all variables in the program (*i.e.*, $r0 - r3$). However, we discover if $r2 \neq r1$, two windows will produce different values for $r3$. K2 addresses this by doing a static analysis to infer the constraints of live variables into windows. According to the prefix, we can get the constraint that $r1$ and $r2$ are assigned the same value 0. In summary, K2 uses the following modular equivalence check query to verify whether two windows are semantically equivalent.

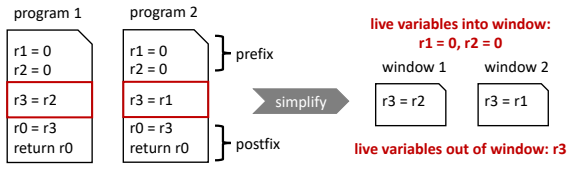

Figure 10: An example of modular verification. We can prove two windows are equivalent by inferring $r1$ and $r2$ are assigned 0 from the prefix and two windows only need to agree on $r3$ from the postfix.

```
live variables into window 1
== live variables into window 2
 ∧  input-output behavior of window 1
 ∧  input-output behavior of window 2
 ⇒   live variables out of window 1
      != live variables out of window 2
```

3.2 Safety check of eBPF programs

The safety of an eBPF program is checked by static analysis, program interpreter, and first-order logic. First, K2 does a static analysis to check whether a program contains any unsafe instructions by inferring the variable states. Second, K2 uses test cases to interpret the program passing the static analysis, to check whether there is any unsafe instruction. Finally, for the program passing the test cases,

K2 does a formal safety check by encoding the program behavior and the safety check in a first-order logic formula and leveraging Z3 solver to check whether the formula is unsatisfiable. If it is satisfiable, K2 will add the counterexample to the test cases. Currently, K2 supports control flow, memory accesses within bounds, memory access alignment, and checker-specific constraints (*e.g.*, $r5 - r9$ are unreadable after a help function call).

4 Conclusion

In this report, we first explained the program synthesis problem and introduced a typical synthesis model CEGIS. Then we described and discussed the previous program synthesis work: stochastic synthesis, enumerative synthesis, constraint-based synthesis, and other program synthesis works. Finally, we presented K2, a compiler for optimizing eBPF programs using program synthesis. Based on previous work CEGIS and the STOKE framework, we designed several domain-specific techniques to efficiently synthesize eBPF programs satisfying complex specification. K2 can produce safe and optimized drop-in replacements for existing eBPF programs.

References

[1] ebpf. `https://www.kernel.org/doc/Documentation/networking/filter.txt`.

[2] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound loop superoptimization for google native client. *ACM SIGPLAN Notices*, 52(4):313–326, 2017.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

[5] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[6] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726, 2016.

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

[8] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[9] Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. Dataflow-based pruning for speeding up superoptimization. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–24, 2020.

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

[11] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A

synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

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

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

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

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

[16] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[17] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

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