Automatically optimizing BPF programs using program synthesis

Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, Anirudh Sivaraman
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
BPF can safely and efficiently extend kernel functionality

• A general kernel extension mechanism
  • Networking
  • Observability
  • Security
• A virtual machine with RISC instruction set
  • Eleven 64-bit registers
  • Stack (512 bytes)
  • Key-value maps
  • Helper functions
Workflow of BPF developers

The kernel checker ensures **safety**. Unprivileged BPF programs shouldn’t crash the kernel or leak privileged data!
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Motivation: It’s hard to develop high-quality BPF programs

- Low latency
- High throughput
- Safe
- Compact
(1) Size

• The kernel checker must verify program safety quickly
• Modern kernels examine 1 million instructions across all code paths
In practice, programs with even a few thousand instructions may be rejected by the kernel checker.

Disable some features or refactor the code.

https://github.com/cilium/cilium/issues/15249
(2) Performance

• Even small optimizations matter at high line rates
• Option 1: Developers manually optimize code
  • Strong expertise
  • Painstaking for long programs
• Option 2: Compiler optimization support
  • clang-9 -O2/O3 produced identical code for benchmarks
The Problem: Can we automatically produce compact, more performant programs?

The Challenge: Tension between Performance and Safety
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Kernel checker is stricter than necessary

• A program “unsafe” to the kernel checker may in fact be safe

Example:

*(u16*)(r10 - 511) = 0xFFFF

// store a value on the stack
Kernel checker is stricter than necessary

- A program “unsafe” to the kernel checker may in fact be safe

Example:

```c
*(u16*)(r10 - 511) = 0xFFFF
// store a value on the stack
```
Kernel checker is stricter than necessary

- A program “unsafe” to the kernel checker may in fact be safe

Example:

```c
*(u16*)(r10 - 511) = 0xFFFF
// store a value on the stack
```

Constraint

**stack access alignment**

(stack access address - stack start) mod 2 == 0

“Unsafe”: (r10 - 511) - (r10 - 512) mod 2 = 1
Kernel checker is stricter than necessary

• A program “unsafe” to the kernel checker may in fact be safe

Example:

*(u16*)(r10 - 511) = 0xFFFF
// store a value on the stack

Constraint

stack access alignment
(stack access address - stack start) mod 2 == 0

“Unsafe”: (r10 - 511) - (r10 - 512) mod 2 = 1
Pattern-based optimizations

• Traditional compilers match patterns & rewrite small regions of code

Example:

*(u8*)(rX + off) = 0
*(u8*)(rX + off + 1) = 0

can be optimized as

*(u16*)(rX + off) = 0
Pattern-based optimizations

• Traditional compilers match patterns & rewrite small regions of code

Example:

*(u8*)(rX + off) = 0  
*(u8*)(rX + off + 1) = 0

can be optimized as

*(u16*)(rX + off) = 0
Pattern-based optimizations

• Traditional compilers match patterns & rewrite small regions of code

Example:

*(u8*)(rX + off) = 0
*(u8*)(rX + off + 1) = 0

can be optimized as

*(u16*)(rX + off) = 0

```
 portals
      0x0
      0x0
Memory
```
Pattern-based optimizations

• Traditional compilers match patterns & rewrite small regions of code

Example:

\[
*(u8\ast)(rX + \text{off}) = 0
\]
\[
*(u8\ast)(rX + \text{off} + 1) = 0
\]
can be optimized as

\[
*(u16\ast)(rX + \text{off}) = 0
\]
Optimizations can violate safety!

- Many pattern-matching optimizations are incompatible with the safety constraints enforced by the checker!

Example:

\[(u8\ast)(rX + \text{off}) = 0\]
\[(u8\ast)(rX + \text{off} + 1) = 0\]

Can be optimized as

\[(u16\ast)(rX + \text{off}) = 0\]

Constraint: stack access alignment

\[(\text{stack access address} - \text{stack start}) \mod 2 = 0\]

“Unsafe”: \[(r10 - 511) - (r10 - 512) \mod 2 = 1\]

This optimization will be rejected
Every potential optimization must also consider safety.

We call this the phase-ordering problem of BPF compilation.
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
K2, an optimizing compiler for BPF

K2 achieves

✓ 6–26% compression
✓ 1.36–55.03% lower average latency
✓ 0–4.75% higher throughput relative to best clang-compiled program among the -O2/Os options
K2’s Contributions

• K2 leverages stochastic program synthesis to optimize programs

• K2 provides formal correctness and safety guarantees
  • BPF instruction set in first-order logic
    • BPF arithmetic & logic, pointer aliasing, control flow, BPF maps, helper functions
  • Fast equivalence-checking techniques: 6 orders of magnitude gain
Stochastic Program Synthesis

A search procedure that automatically generates programs satisfying a specification:

- Correctness (semantic equivalence)
- Safety
- High performance

Consider these aspects together: address the phase-ordering problem!
A randomized method\textsuperscript{1} to explore the space of programs, guided by a general cost function.

Fast and generalizes easily to BPF optimization:
- enumerative
- constraint-based
- cooperative
- stochastic

Handles complex costs with complex constraints:
- (performance)
- (safety)

\textsuperscript{1} Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. ASPLOS 2013.
Stochastic search in K2

Iteration 1

Current program (input program)

Perf: 10
Error: 0
Safe: 0
Total: 10

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 1

Proposal:
Perf: 10
Error: 100
Safe: MAX
Total: MAX

Current program (input program):
Perf: 10
Error: 0
Safe: 0
Total: 10

Max acceptable cost increment: 0.5
MAX > 10 + 0.5

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 1

Current program (input program)
- Perf: 10
- Error: 0
- Safe: 0
- Total: 10

Proposal
- Perf: 10
- Error: 0
- Safe: MAX
- Total: MAX

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe

Max acceptable cost increment: 0.5
MAX > 10 + 0.5

Rejected
Stochastic search in K2

Iteration 2

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe

Max acceptable cost increment: 0.5
10.1 < 10 + 0.5
Stochastic search in K2

Iteration 2

Helps the search find the global optimal

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe

Max acceptable cost increment: 0.5
10.1 < 10+0.5
Stochastic search in K2

Iteration 2

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe

Current program
Perf: 10.1
Error: 0
Safe: 0
Total: 10.1
Stochastic search in K2

Iteration 3

Proposal
Perf: 8
Error: 1
Safe: 0
Total: 9

Current program
Perf: 10.1
Error: 0
Safe: 0
Total: 10.1

Max acceptable cost increment: 0.5
9 < 10.1 + 0.5

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 3

Current program
Perf: 10.1
Error: 0
Safe: 0
Total: 10.1

Proposal
Perf: 8
Error: 1
Safe: 0
Total: 9

Max acceptable cost increment: 0.5
9 < 10.1+0.5

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 3

Current program

Perf: 8
Error: 1
Safe: 0
Total: 9

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 4

Current program
Perf: 8
Error: 1
Safe: 0
Total: 9

Proposal
Perf: 8
Error: 0
Safe: 0
Total: 8

Max acceptable cost increment: 0.5
8 < 9 + 0.5

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 4

Current program
Perf: 8
Error: 1
Safe: 0
Total: 9

Proposal
Perf: 8
Error: 0
Safe: 0
Total: 8

Max acceptable cost increment: 0.5
8 < 9 + 0.5

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
Stochastic search in K2

Iteration 4

With more iterations, the best programs have lower cost!

Current program

Perf: 8
Error: 0
Safe: 0
Total: 8

Cost contour line: the darker line, the lower cost
Total cost = Perf + Error + Safe
K2 Overview

Input program → BPF bytecode

Optimized program → BPF bytecode
K2 Overview

1. **Input program**
   - BPF bytecode

2. **Proposal**

3. **Current program**

4. **Cost computation**

5. **Decide next program**

6. **Stochastic search loop**

7. **Next Prog.**

8. **Optimized program**
   - BPF bytecode
K2 Overview

Input program

Proposal

Current program

Cost computation

Decide next program

Optimized program

Stochastic search loop

BPF bytecode

Programs safe + equivalent

Next Prog. accepted

Proposal

Current program

Cost computation

Decide next program

Optimized program

BPF bytecode
K2 Overview

Input program

Proposal

Current program

Cost computation

Decide next program

Perf. cost ranking

Top-k programs

Post processing

Optimized program

BPF bytecode

Stochastic search loop

Hit max iteration

Next Program accepted

Next Program

Programs safe + equivalent
**K2 Overview**

- **Input program**

  - Proposal
  - Current program
    - Cost computation
    - Decide next program

  - Stochastic search loop

- **Perf. cost ranking**
  - Top-k programs
  - Post processing
    - Optimized program

- **Correctness + Safety + Performance**

- **Programs safe + equivalent**

- **Hit max iteration**

**BPF bytecode**
Computing cost

- Performance cost:
  - instruction count for reducing program size
  - program estimated running time for improving throughput/latency
Computing cost

- Pruning unequal or unsafe proposals by interpreting them with test cases
- Speed up the cost computation
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Equivalence check

• Logically assert that, for all inputs, given the same input to the two programs, the outputs of the programs must be the same

input: r1, r10

program 1
r0 = r1
exit

program 2
r0 = r10
exit

output: r0

outputs not equal if r1 != r10

• How to do this in general? Proving program equivalence requires formalizing the programs’ behaviors in logic

BPF programs in this example are fake, only used to illustrate the equivalence check
Equivalence check

- First-order logic with the theory of 64-bit-wide bit vectors
- If unequivalent, solvers also return a counterexample (one input) where the two programs generate different outputs

\[
\begin{align*}
\text{inputs to program 1} & \equiv \text{inputs to program 2} \\
\land \text{ input-output behavior of program 1} & \\
\land \text{ input-output behavior of program 2} & \\
\Rightarrow & \text{ outputs of program 1} \neq \text{ outputs of program 2}
\end{align*}
\]
Equivalence check

• First-order logic with the theory of 64-bit-wide bit vectors

input: r1, r10
output: r0

inputs to program 1 == inputs to program 2
\land \ input-output behavior of program 1
\land \ input-output behavior of program 2
\Rightarrow \ outputs of program 1 \neq \ outputs of program 2

program 1
- r0 = r1
- exit

r1_{p1} == r1_{p2}
\land \ r10_{p1} == r10_{p2}
\land \ r0_{p1} == r1_{p1}
\Rightarrow \ r0_{p1} \neq r0_{p2}

program 2
- r0 = r10
- exit

r0_{p1} == r10_{p2}
\land \ r0_{p2} == r10_{p2}
\Rightarrow \ r0_{p1} \neq r0_{p2}
Equivalence check

• First-order logic with the theory of 64-bit-wide bit vectors

input: $r_1, r_{10}$
output: $r_0$

program 1
---
$r_0 = r_1$
exit

program 2
---
$r_0 = r_{10}$
exit

inputs to program 1 == inputs to program 2
$\Lambda$ input-output behavior of program 1
$\Lambda$ input-output behavior of program 2
$\Rightarrow$ outputs of program 1 != outputs of program 2

$r_0 = r_{10}$
exit

SMT solver

SAT: program1 != program 2
model:
$r_{1\_p1} = 1, r_{10\_p1} = 2$  
$r_{1\_p2} = 1, r_{10\_p2} = 2$  
(same inputs)
$r_{0\_p1} = 1$  
$r_{0\_p2} = 2$  
(different outputs)
Formalization in first-order logic

• Characterizing input-output behavior of programs requires formalizing each BPF instruction opcode in first-order logic

• Tedious, but straightforward for arithmetic and logic instructions

• BPF programs are loop free

• Challenge: memory load/store, branching, BPF helper calls (in the paper)
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Fast equivalence check

• Equivalence checking is an expensive operation
• The first-order formula solving time grows quickly with instructions, branches, memory operations, map operations, etc.
• Cilium recvmsg4 (94 instructions): eq. check time > 24 hours!
• Equivalence checking is in K2’s inner (stochastic search) loop

Can we reduce equivalence checking time?
Fast equivalence check

• Simplify the first-order logic formula → reduce solving time
• Cilium recvmsg4 (94 instructions)
Fast equivalence check: Modular verification

• Suppose programs only differ in a small window of instructions

program 1

\[
\begin{align*}
\text{r1 = 0} \\
\text{r2 = 0} \\
\text{r3 = r2} \\
\text{r0 = r3} \\
\text{exit}
\end{align*}
\]

program 2

\[
\begin{align*}
\text{r1 = 0} \\
\text{r2 = 0} \\
\text{r3 = r1} \\
\text{r0 = r3} \\
\text{exit}
\end{align*}
\]

prefix

\[
\begin{align*}
\text{r3 = r2} \\
\text{r2 = 0}
\end{align*}
\]

simplify

\[
\begin{align*}
\text{r3 = r2} \\
\text{r1 = 0}
\end{align*}
\]

postfix

\[
\begin{align*}
\text{r3 = r1} \\
\text{r0 = r3}
\end{align*}
\]
Fast equivalence check: Modular verification

• Suppose programs only differ in a small window of instructions

program 1

\[
\begin{align*}
    r_1 &= 0 \\
    r_2 &= 0 \\
    r_3 &= r_2 \\
    r_0 &= r_3 \\
    \text{exit}
\end{align*}
\]

program 2

\[
\begin{align*}
    r_1 &= 0 \\
    r_2 &= 0 \\
    r_3 &= r_1 \\
    r_0 &= r_3 \\
    \text{exit}
\end{align*}
\]

prefix

\[
\begin{align*}
    r_3 &= r_1 \\
    r_0 &= r_3 \\
    \text{exit}
\end{align*}
\]

postfix

\[
\begin{align*}
    r_3 &= r_2 \\
    r_3 &= r_1 \\
\end{align*}
\]

win 1

win 2

What are output variables to be compared?

Live variables out of the window
(infer from the postfix program)
Fast equivalence check: Modular verification

- Suppose programs only differ in a small window of instructions

```
program 1
r1 = 0
r2 = 0
r3 = r2
r0 = r3
exit
```

```
program 2
r1 = 0
r2 = 0
r3 = r1
r0 = r3
exit
```

- Suppose programs only differ in a small window of instructions

```
prefix

r1 = 0
r2 = 0
```

```
simplify

r3 = r1
r0 = r3
exit
```

```
postfix

win 1
r3 = r1

win 2
r3 = r2
```

What are output variables to be compared?

Live variables out of the window
(infer from the postfix program)
Fast equivalence check: Modular verification

• Suppose programs only differ in a small window of instructions

```
program 1
r1 = 0
r2 = 0
r3 = r2
r0 = r3
exit

program 2
r1 = 0
r2 = 0
r3 = r1
r0 = r3
exit
```

output: r3
Fast equivalence check: Modular verification

• Suppose programs only differ in a small window of instructions

program 1

\[
\begin{align*}
  r1 &= 0 \\
  r2 &= 0 \\
  r3 &= r2 \\
  r0 &= r3 \\
  \text{exit}
\end{align*}
\]

program 2

\[
\begin{align*}
  r1 &= 0 \\
  r2 &= 0 \\
  r3 &= r1 \\
  r0 &= r3 \\
  \text{exit}
\end{align*}
\]

\[
\begin{align*}
  r3 &= r2 \\
  \text{win 1}
\end{align*}
\]

\[
\begin{align*}
  r3 &= r1 \\
  \text{win 2}
\end{align*}
\]

\[
\begin{align*}
  \text{output: } r3
\end{align*}
\]

Infer input variables and preconditions from the prefix program

\[==\]

\[!=\]
Fast equivalence check: Modular verification

• Suppose programs only differ in a small window of instructions

program 1

\[
\begin{align*}
\text{r1} &= 0 \\
\text{r2} &= 0 \\
\text{r3} &= \text{r2} \\
\text{r0} &= \text{r3} \\
\text{exit}
\end{align*}
\]

program 2

\[
\begin{align*}
\text{r1} &= 0 \\
\text{r2} &= 0 \\
\text{r3} &= \text{r1} \\
\text{r0} &= \text{r3} \\
\text{exit}
\end{align*}
\]

input: \( r1 = 0, r2 = 0 \)

win 1

\[
\begin{align*}
\text{r3} &= \text{r2}
\end{align*}
\]

win 2

\[
\begin{align*}
\text{r3} &= \text{r1}
\end{align*}
\]

output: \( r3 \)

Infer input variables and preconditions from the prefix program
Fast equivalence check: Modular verification

• Suppose programs only differ in a small window of instructions
• Equivalence check over two windows instead of two programs
• Infer:
  • Input variables and preconditions from the prefix program
  • Output variables from the postfix program

\[
\begin{align*}
\text{win input variables preconditions} & \quad \text{inferred from the prefix program} \\
\land \text{variables live into win 1} & \quad = \quad \text{variables live into win 2} \\
\land \text{input-output behavior of win 1} & \\
\land \text{input-output behavior of win 2} & \\
\Rightarrow \text{variables live out of win 1} & \quad \neq \quad \text{variables live out of win 2}
\end{align*}
\]
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Safety check

• Safety checks
  • Control flow
  • Memory accesses within bounds
  • Access alignment
  • Checker-specific constraints

• Techniques
  • Static analysis
  • First-order logic formula
    • Use safety counterexample inputs to prune unsafe programs
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Evaluation:
How well does it work?

Program compactness
(number of instructions)

Program performance
(Latency & Throughput)
How compact are K2-synthesized programs?

- 19 benchmarks
  - Cilium, Facebook, hXDP, kernel samples
  - Instruction count: 18-1771
- Compression: 6-26%
  - Mean: 13.95%
- Compiling time
  - Mean: 22 minutes (excluding Facebook’s Katran xdp-balancer)

### Benchmark Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of instructions</th>
<th>Compiling time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clang(^2)</td>
<td>K2</td>
</tr>
<tr>
<td>xdp_router_ipv4</td>
<td>111</td>
<td>99</td>
</tr>
<tr>
<td>xdp_map_access</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>xdp_redirect</td>
<td>43</td>
<td>35</td>
</tr>
<tr>
<td>from-network</td>
<td>39</td>
<td>29</td>
</tr>
<tr>
<td>xdp_pktcntr</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>xdp-balancer</td>
<td>1771</td>
<td>1607</td>
</tr>
</tbody>
</table>

\(^1\) More benchmark results are in the paper
\(^2\) The smallest program across clang -O1/O2/O3/Os
How beneficial is K2 to packet throughput and latency?

- BPF programs attached to the DUT’s network device driver
- Measure packet-processing throughput and the average roundtrip latency
How beneficial is K2 to packet throughput and latency?

- Throughput: the maximum loss-free forwarding rate (MLFFR) in Mpps (millions of packets per second) per core
How beneficial is K2 to packet throughput and latency?

- Throughput: the maximum loss-free forwarding rate (MLFFR) in Mpps (millions of packets per second) per core
- Avg. throughput improvement across 6 benchmarks: 0–4.75%

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>-O1</th>
<th>-O2/O3</th>
<th>K2</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>xdp2</td>
<td>8.855</td>
<td>9.547</td>
<td>9.748</td>
<td>2.11%</td>
</tr>
<tr>
<td>xdp_router_ipv4</td>
<td>1.496</td>
<td>1.496</td>
<td>1.496</td>
<td>0.00%</td>
</tr>
<tr>
<td>xdp_fwd</td>
<td>4.886</td>
<td>4.984</td>
<td>5.072</td>
<td>1.77%</td>
</tr>
<tr>
<td>xdp1</td>
<td>16.837</td>
<td>16.85</td>
<td>17.65</td>
<td>4.75%</td>
</tr>
<tr>
<td>xdp_map_access</td>
<td>14.679</td>
<td>14.678</td>
<td>15.074</td>
<td>2.70%</td>
</tr>
<tr>
<td>xdp-balancer</td>
<td>DNL*</td>
<td>3.292</td>
<td>3.389</td>
<td>2.94%</td>
</tr>
</tbody>
</table>

* Not able load into the kernel as the program was rejected by the kernel checker
How beneficial is K2 to packet throughput and latency?

• Average roundtrip latency in 4 different packet sending rates in Mpps (millions of packets per second) per core

![Graph showing latency vs TX rate]

Smaller latency is better

• TX rate: low (smaller than the lowest throughput of the worst clang or K2)
How beneficial is K2 to packet throughput and latency?

- Average roundtrip latency in 4 different packet sending rates in Mpps (millions of packets per second) per core

![Graph showing average latency vs TX rate](image)

Smaller latency is better

- TX rate: medium (the lower throughput between the best clang and K2)
How beneficial is K2 to packet throughput and latency?

- Average roundtrip latency in 4 different packet sending rates in Mpps (millions of packets per second) per core

![Graph showing latency vs TX rate]

Smaller latency is better

- TX rate: high (the higher throughput between the best clang and K2)
How beneficial is K2 to packet throughput and latency?

- Average roundtrip latency in 4 different packet sending rates in Mpps (millions of packets per second) per core

Smaller latency is better

- TX rate: **saturating** (higher than the highest throughput of the best clang or K2)
How beneficial is K2 to packet throughput and latency?

- Average roundtrip latency in 4 different packet sending rates in Mpps (millions of packets per second) per core
- 4 benchmarks: reduction 1.36–55.03%
Optimizations discovered by K2

• Performance goal: reduce instruction count
• Example 1: coalescing multiple memory operations (from Facebook’s xdp_pktcntnr)

```
r1 = 0
*(u32*)(r10-4) = r1
*(u32*)(r10-8) = r1
*(u64*)(r10-8) = 0
```

two 32-bit writes  
one 64-bit write
Optimizations discovered by K2

- Performance goal: reduce instruction count
- Example 1: coalescing multiple memory operations (from Facebook’s xdp_pktcntr)

\[
\begin{align*}
  r1 &= 0 \\
  *(u32*)(r10-4) &= r1 \\
  *(u32*)(r10-8) &= r1 \\
  *(u64*)(r10-8) &= 0
\end{align*}
\]

- Example 2: context-dependent optimizations (from Facebook’s xdp-balancer)
  - Window input: \( r3 = 0x00000000ffe00000 \)

\[
\begin{align*}
  r0 &= r2 \\
  r0 &= r0 \& r3 \\
  r0 &= r0 >> 21 \\
  r0 &= lower32(r2) \\
  r0 &= r0 >> 21
\end{align*}
\]
Outline

• Background
• Motivation
• Challenge
• Our solution (program synthesis)
• Main techniques
  • Equivalence check
  • Equivalence check acceleration
  • Safety check
• Evaluation
• Conclusion and future work
Conclusion

• K2: compiler for safe, compact, performance-optimized BPF programs
  • Up to 26% size and 55% latency reductions

• Domain-specific techniques in synthesis and verification
  • Reduce equivalence checking time by 6 orders of magnitude

Synthesis is a viable approach to optimize BPF programs
Future work

• Scale up optimization to larger BPF programs in a short time

• Explore generating safe optimized code for other infrastructures such as the Windows OS and programmable NICs

• Repair unsafe BPF programs
Future work

• Scale up optimization to larger BPF programs in a short time

• Explore generating safe optimized code for other infrastructures such as the Windows OS and programmable NICs

• Repair unsafe BPF programs
Future work

• Scale up optimization to larger BPF programs in a short time

• Explore generating safe optimized code for other infrastructures such as the Windows OS and programmable NICs

• Repair unsafe BPF programs
Discussion

• Benchmarks (program size, performance)

• What’s a good time budget for an optimizing compiler in your context?

• How can K2 deal with the evolution of the kernel checker?

• Feedback on K2 and future work
Thank you!

k2_compiler@email.rutgers.edu

https://k2.cs.rutgers.edu