Automatically optimizing BPF programs using program synthesis

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

RUTGERS UNIVERSITY | NEW BRUNSWICK

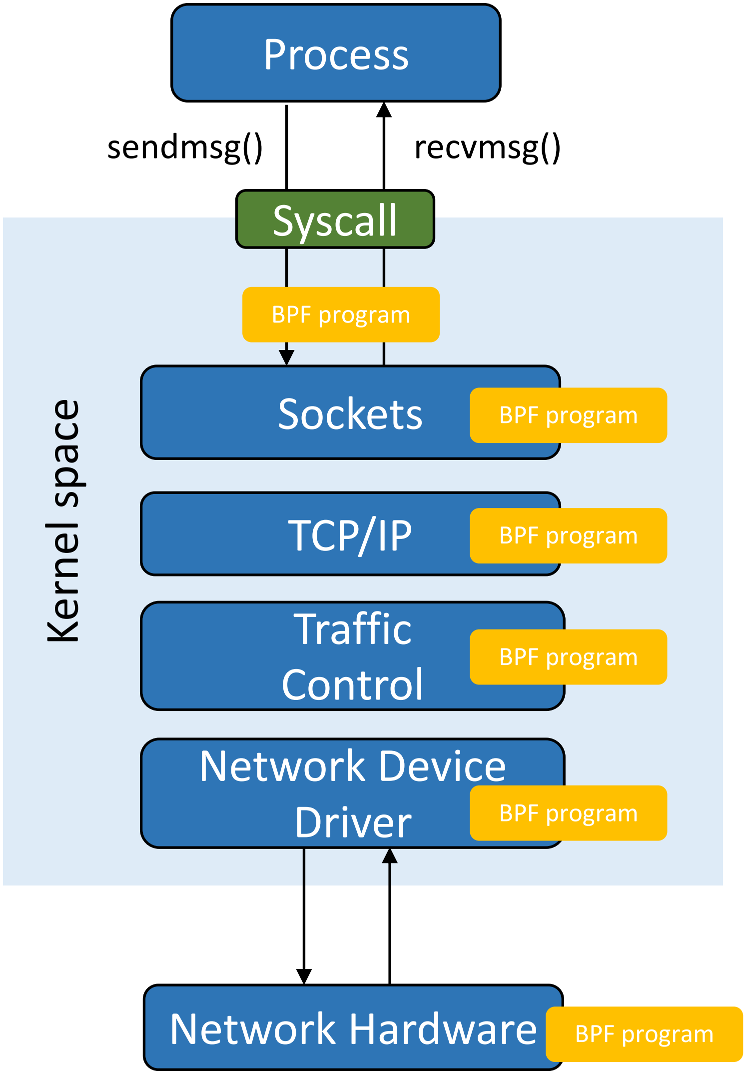
NEW YORK UNIVERSITY

Outline

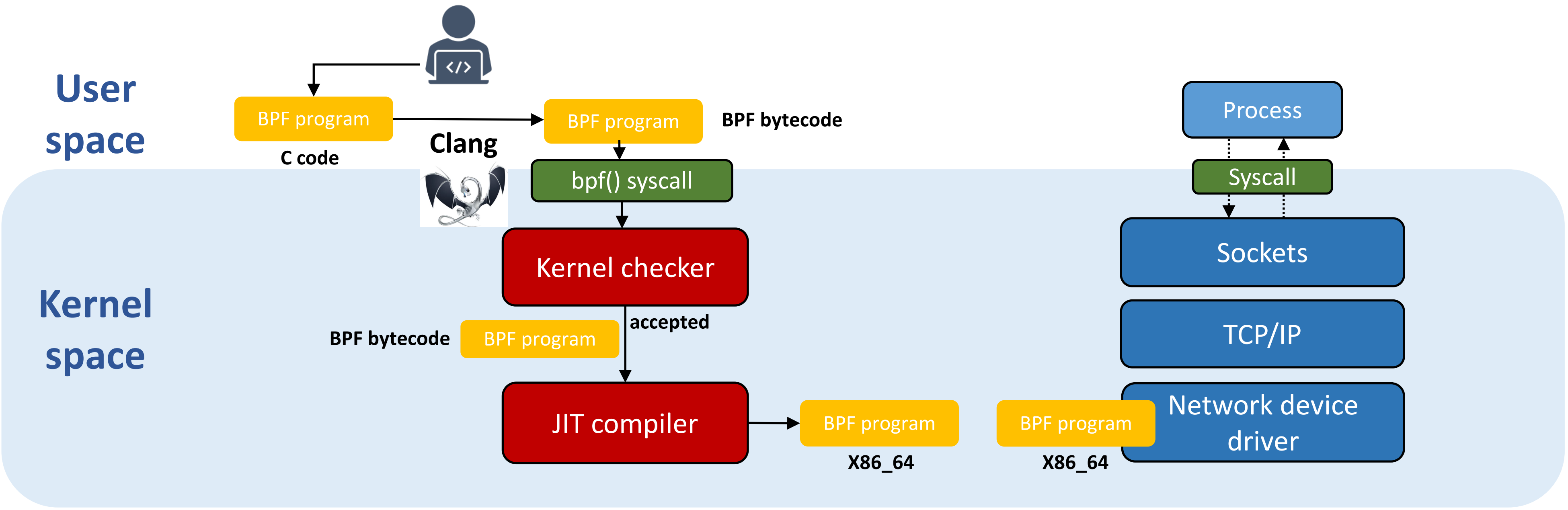
- Background
- Motivation
- Challenge
- Our solution (program synthesis)
- Main techniques
  - Equivalence check
  - Equivalence check acceleration
  - Safety check
- Evaluation
- Conclusion and future work

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

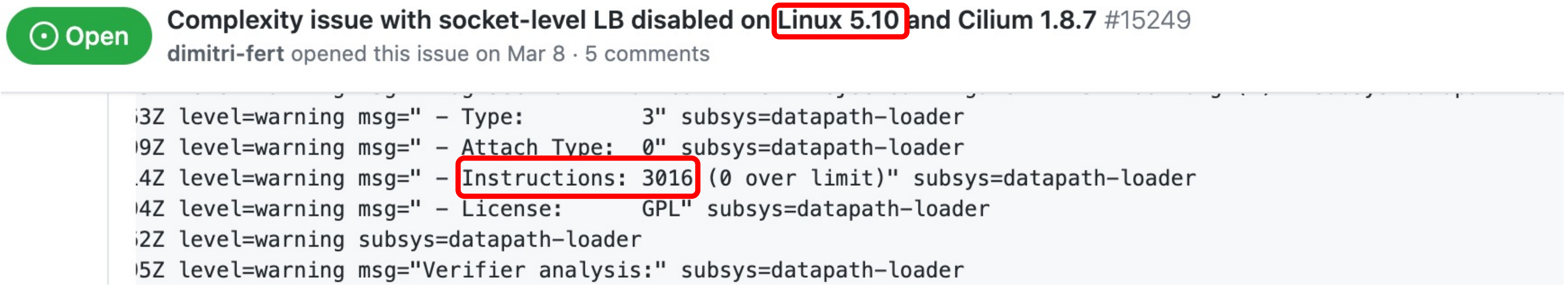
Compact

(1) Size

- The kernel checker must verify program safety quickly
- Modern kernels examine 1 million instructions across all code paths

(1) Size

- In practice, programs with even a few thousand instructions may be rejected by the kernel checker



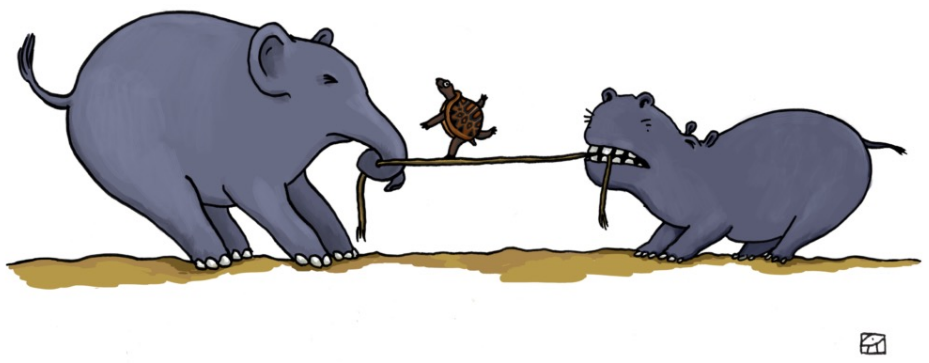
☹ Disable some features or refactor the code

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

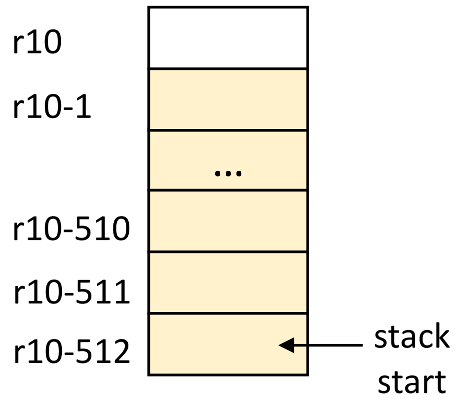
- Background
- Motivation
- Challenge
- Our solution (program synthesis)
- Main techniques
  - Equivalence check
  - Equivalence check acceleration
  - Safety check
- Evaluation
- Conclusion and future work

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

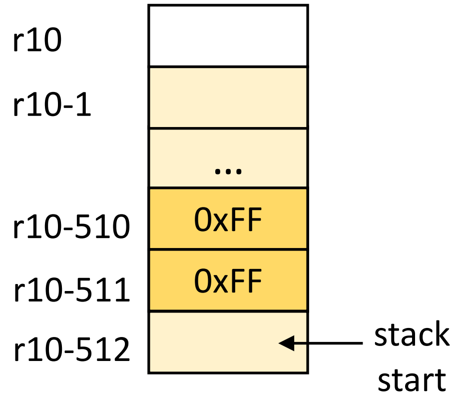


Kernel checker is stricter than necessary

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

Example:

```
*(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:

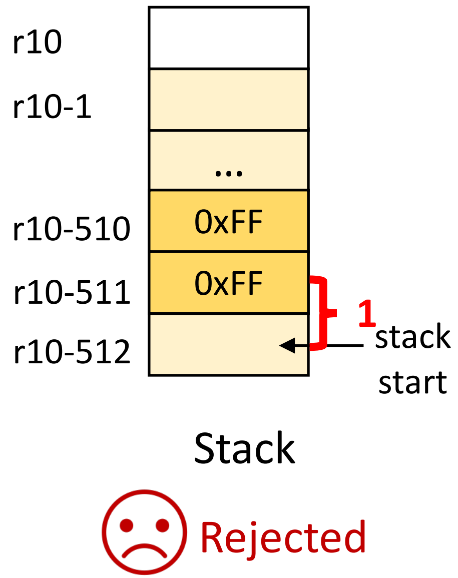
(u16)(r10 - 511) = 0xFFFF
// store a value on the stack

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:

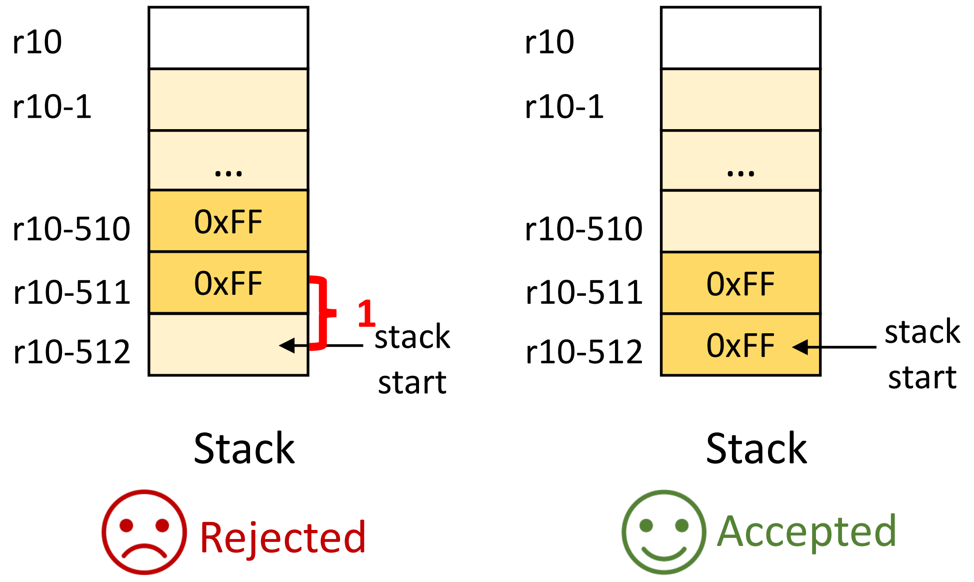
(u16)(r10 - 511) = 0xFFFF
// store a value on the stack

Constraint

stack access alignment

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

"Unsafe": (r10 - 511) - (r10 - 512) mod 2 = 1



Pattern-based optimizations

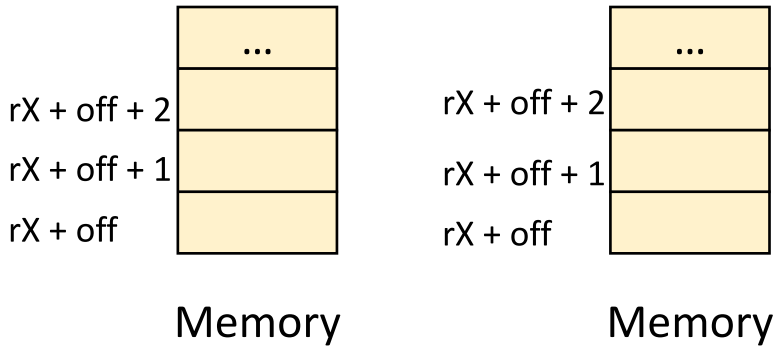
- Traditional compilers match patterns & rewrite small regions of code

Example:

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

can be optimized as

(u16)(rX + off) = 0



Pattern-based optimizations

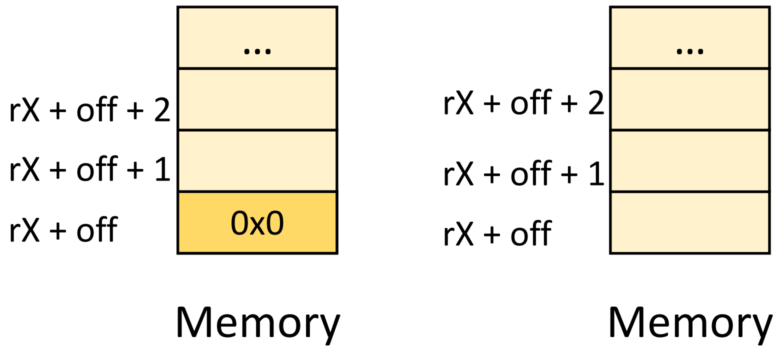
- Traditional compilers match patterns & rewrite small regions of code

Example:

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

can be optimized as

(u16)(rX + off) = 0



Pattern-based optimizations

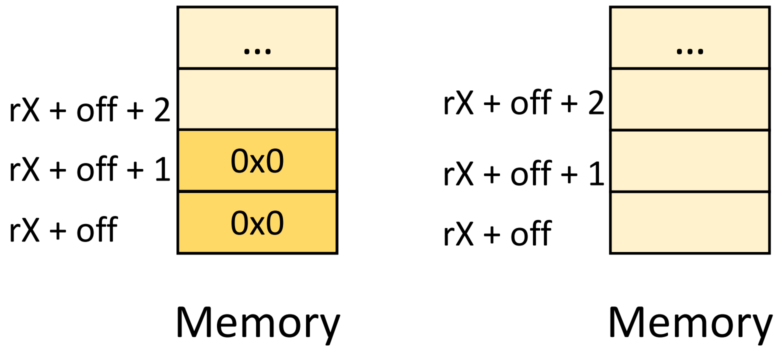
- Traditional compilers match patterns & rewrite small regions of code

Example:

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

can be optimized as

(u16)(rX + off) = 0



Pattern-based optimizations

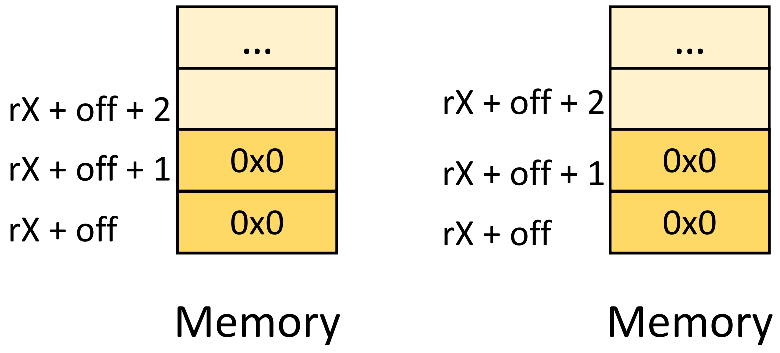
- Traditional compilers match patterns & rewrite small regions of code

Example:

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

can be optimized as

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



Optimizations can violate safety!

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

Example:

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

can be optimized as

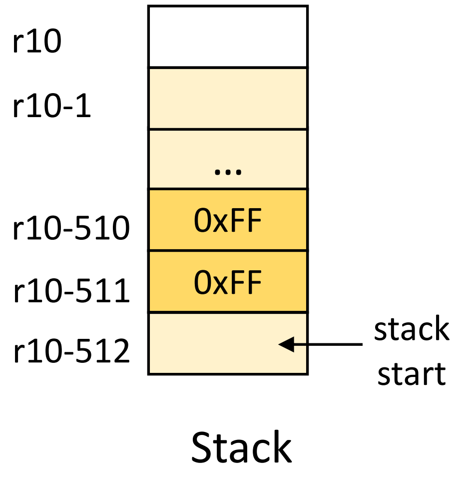
(u16)(rX + off) = 0

Constraint: stack access alignment

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

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

This optimization will be rejected



Rejected

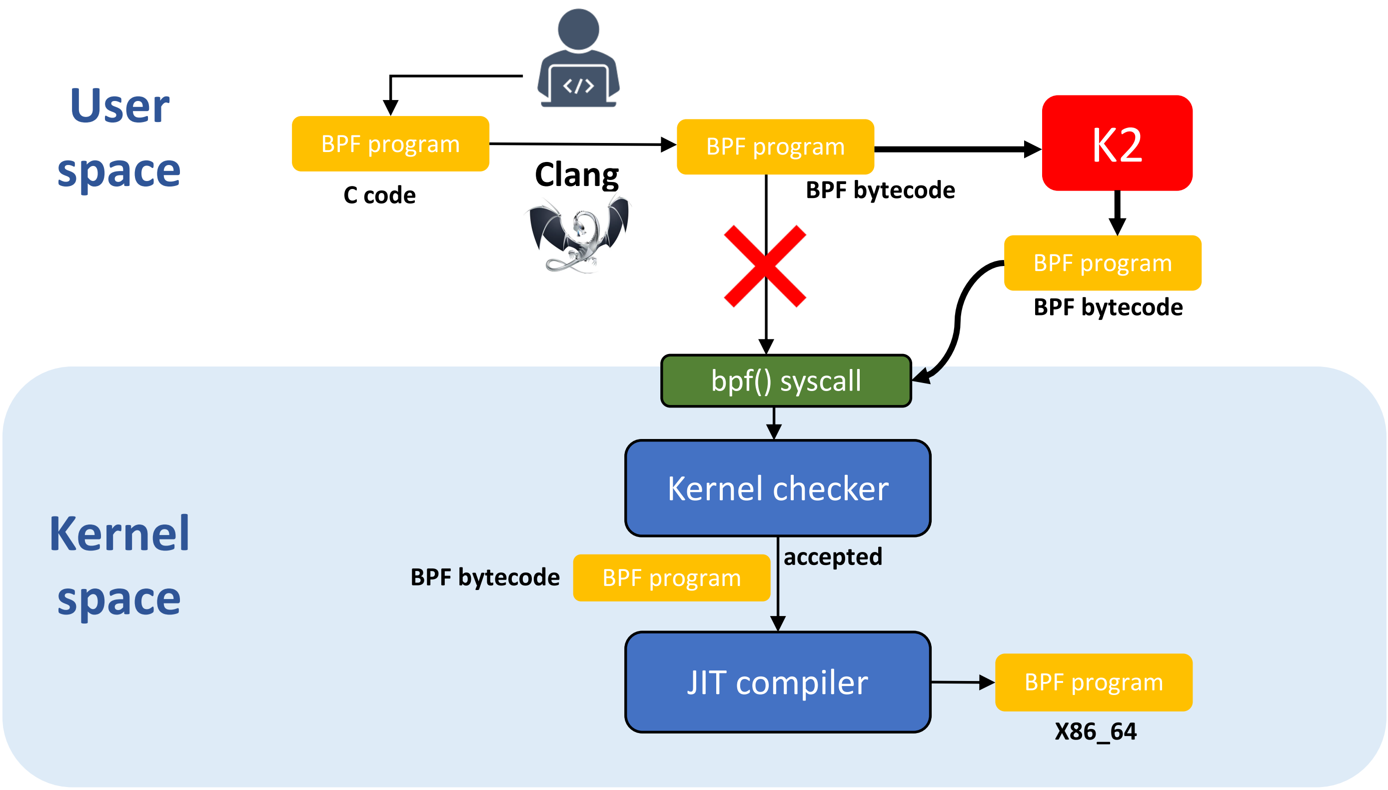
Every potential optimization
must also consider safety.

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!

Stochastic Program Synthesis

A randomized method[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

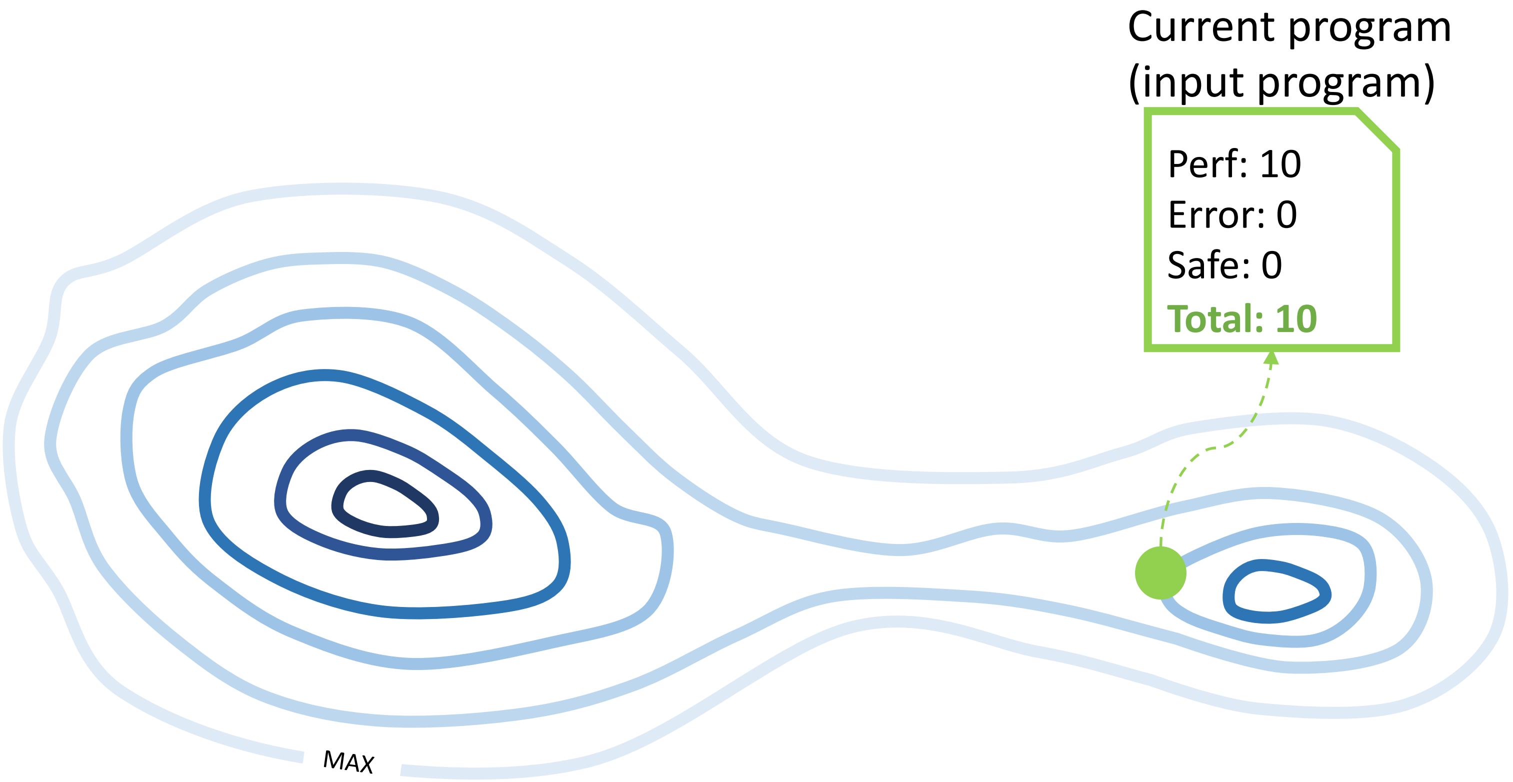
Handles complex costs with complex constraints

(performance) (safety)

[1] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. ASPLOS 2013.

Stochastic search in K2

Iteration 1

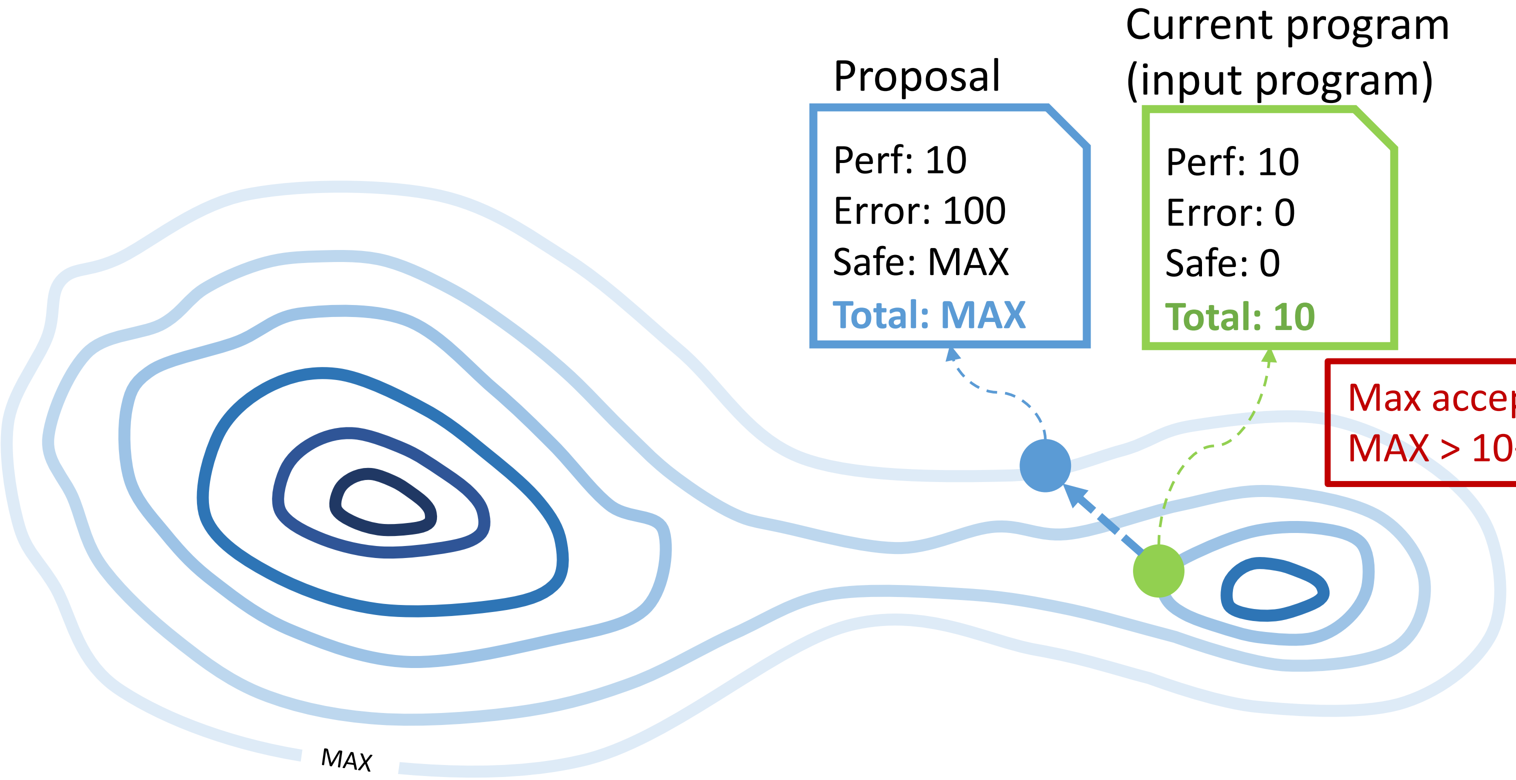


Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 1



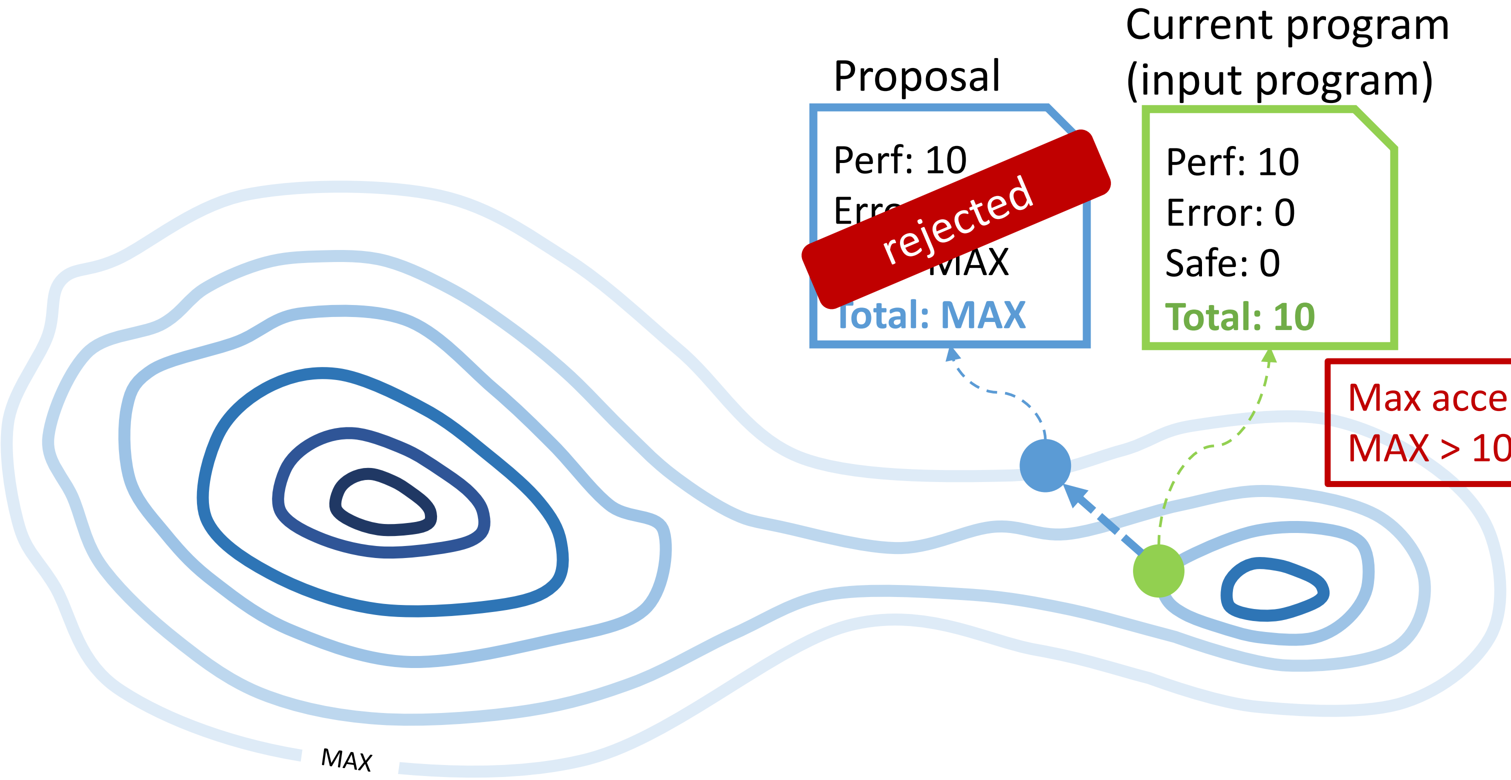
Max acceptable cost increment: 0.5
MAX > 10+0.5

Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 1



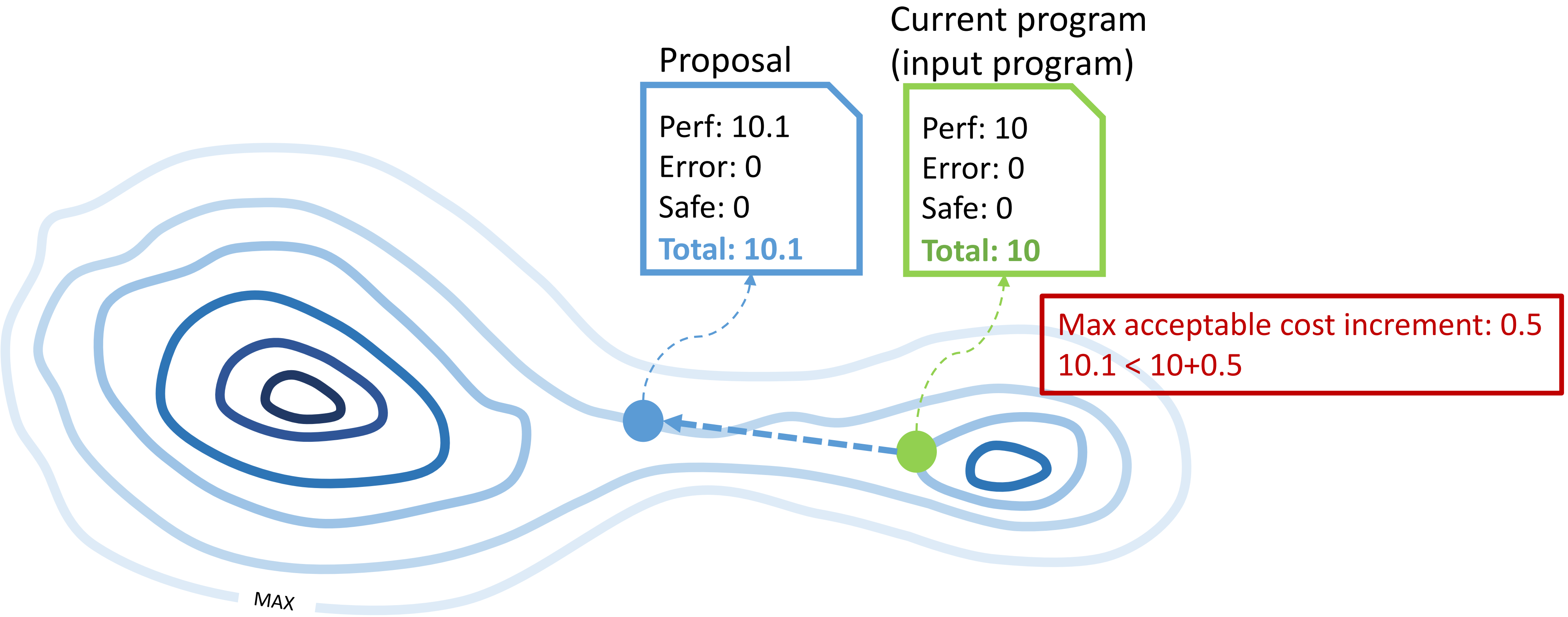
Max acceptable cost increment: 0.5
MAX > 10+0.5

Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 2



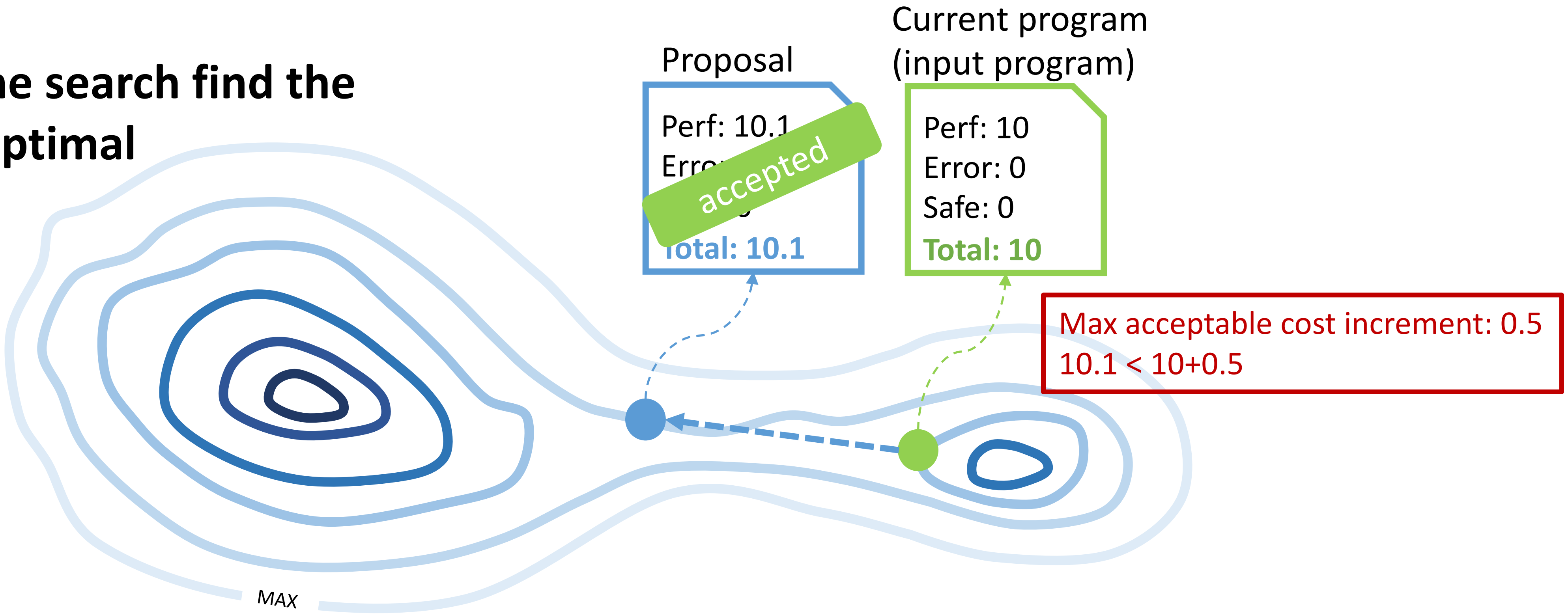
Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 2

Helps the search find the global optimal

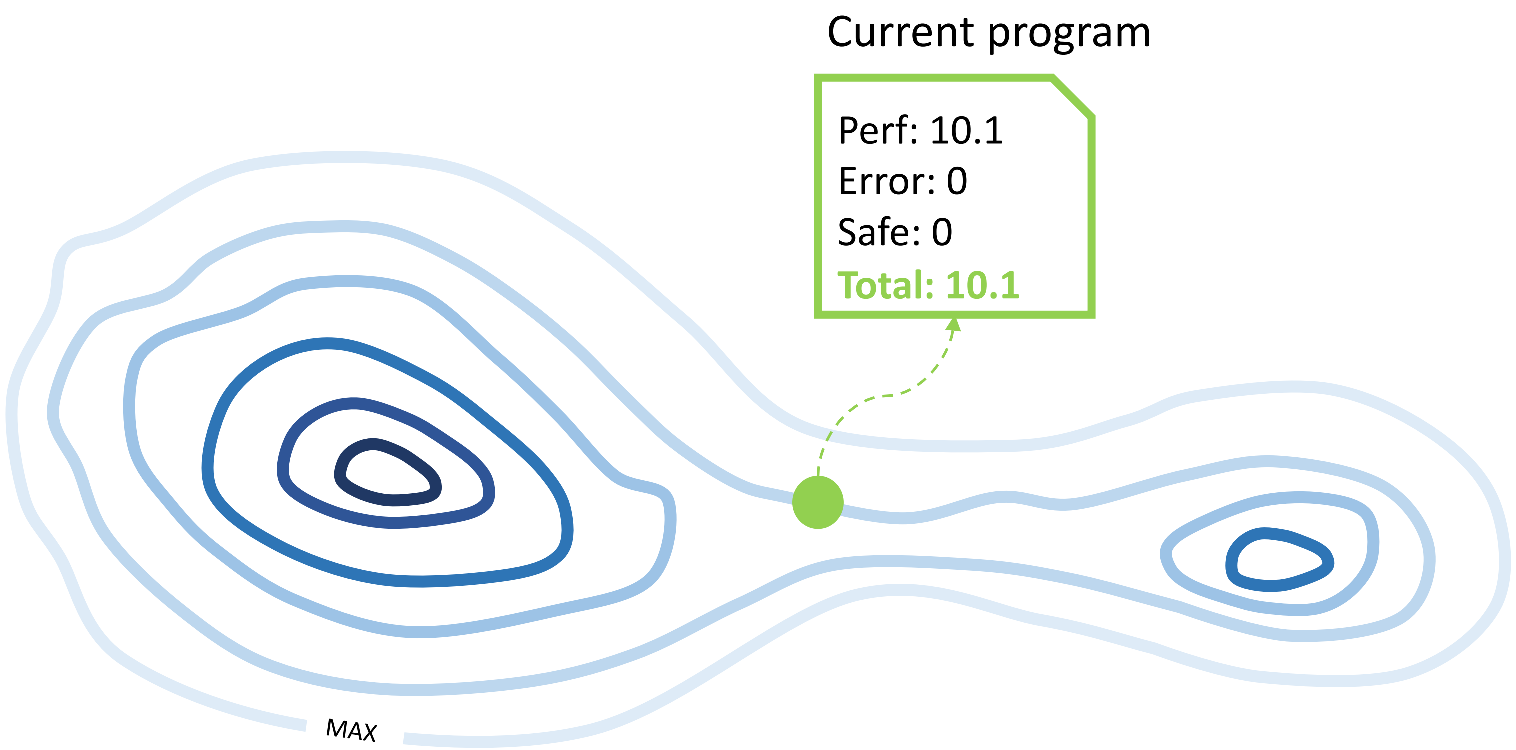


Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 2

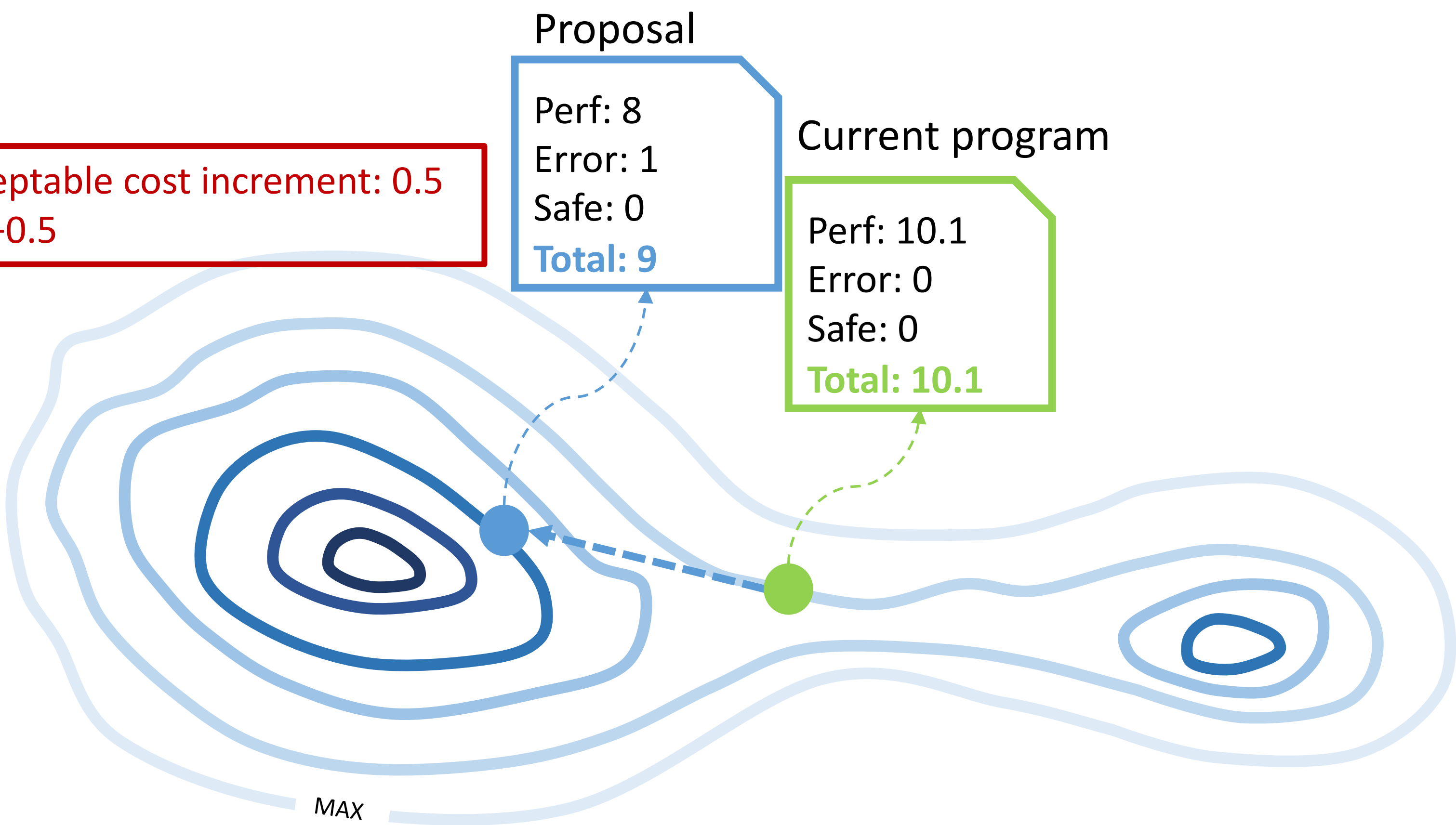


Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 3



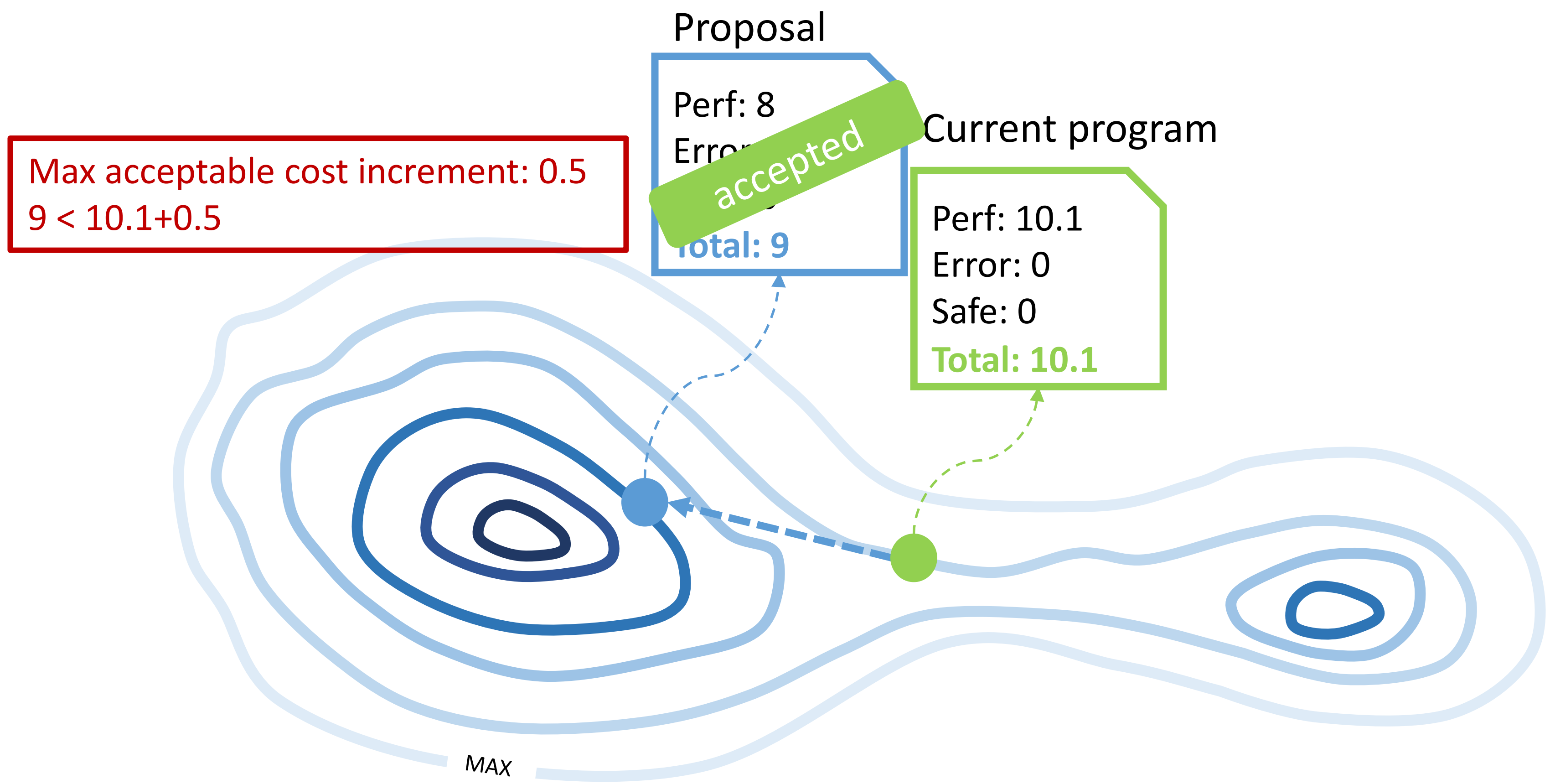
Max acceptable cost increment: 0.5
9 < 10.1+0.5

Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 3

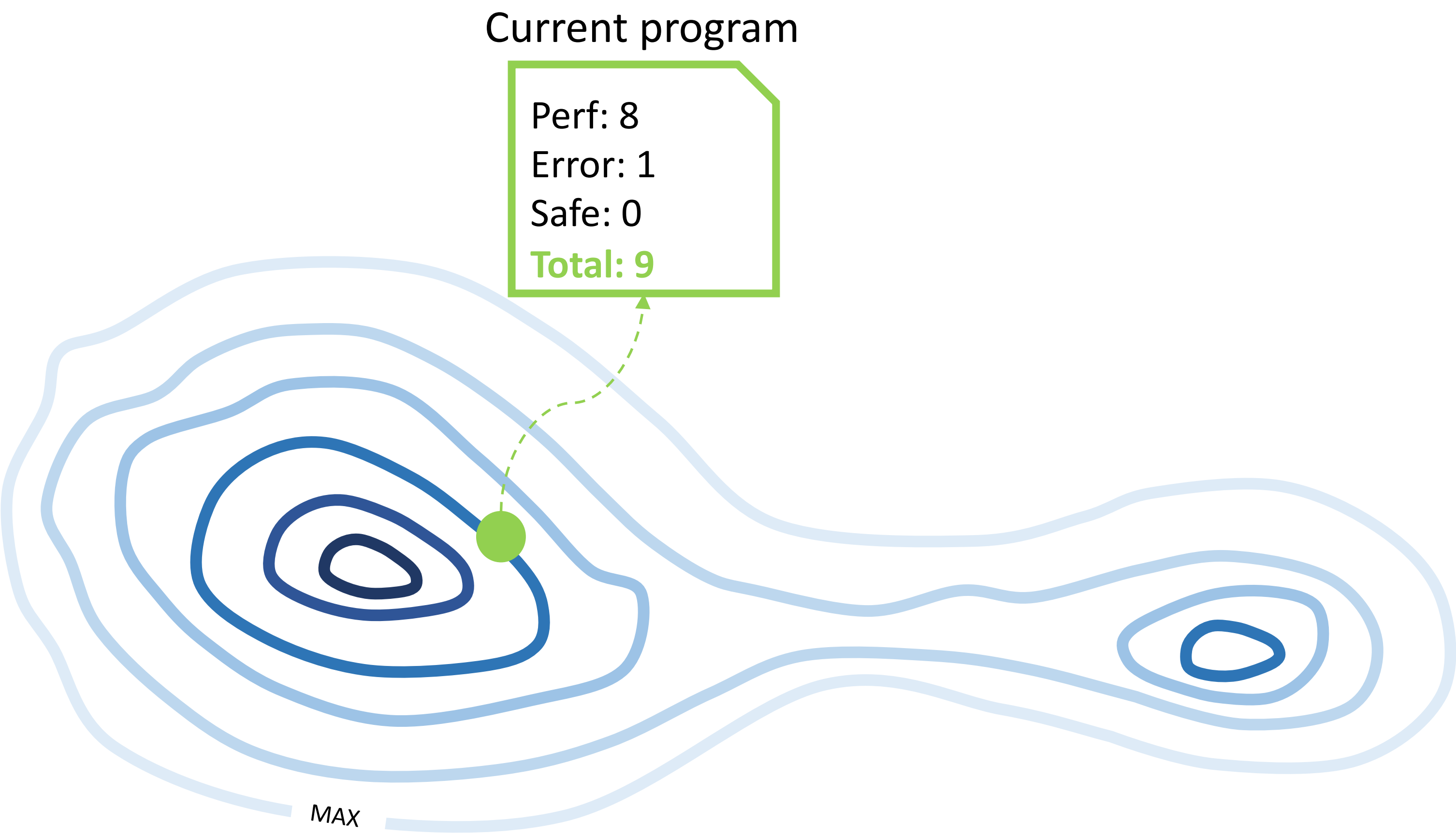


Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 3

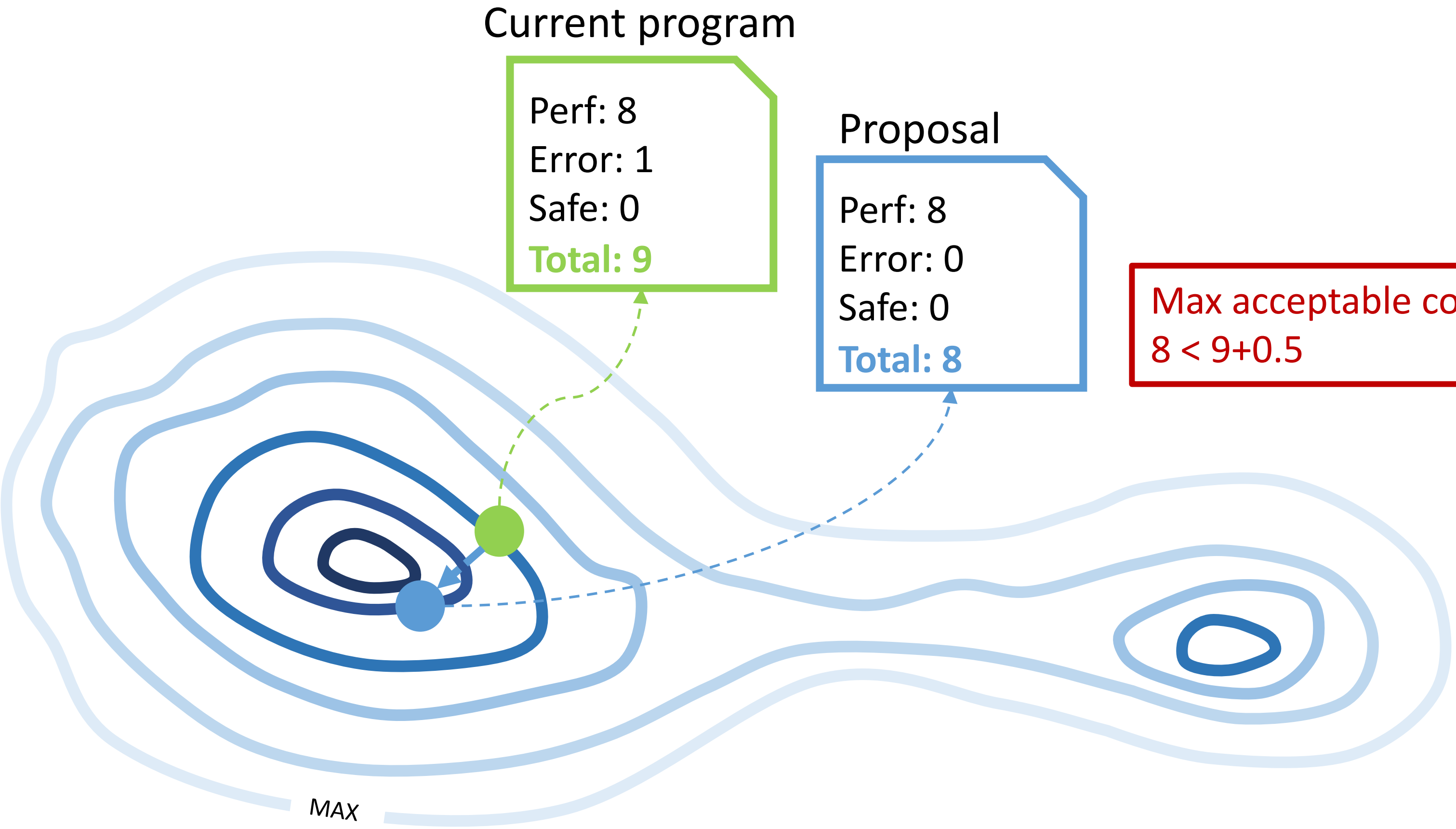


Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

Stochastic search in K2

Iteration 4



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

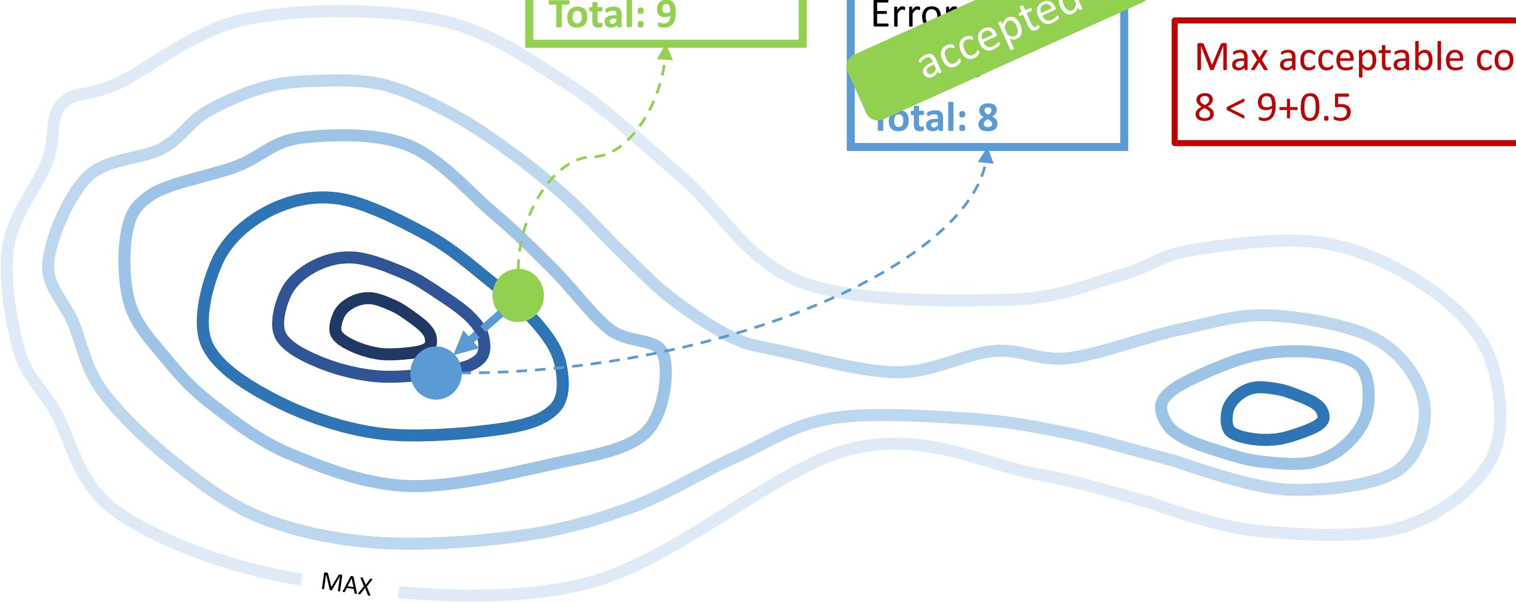
Perf: 8
Error: 1
Safe: 0
Total: 9

Proposal

Perf: 8
Erro
Total: 8

accepted

Max acceptable cost increment: 0.5
8 < 9+0.5



Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

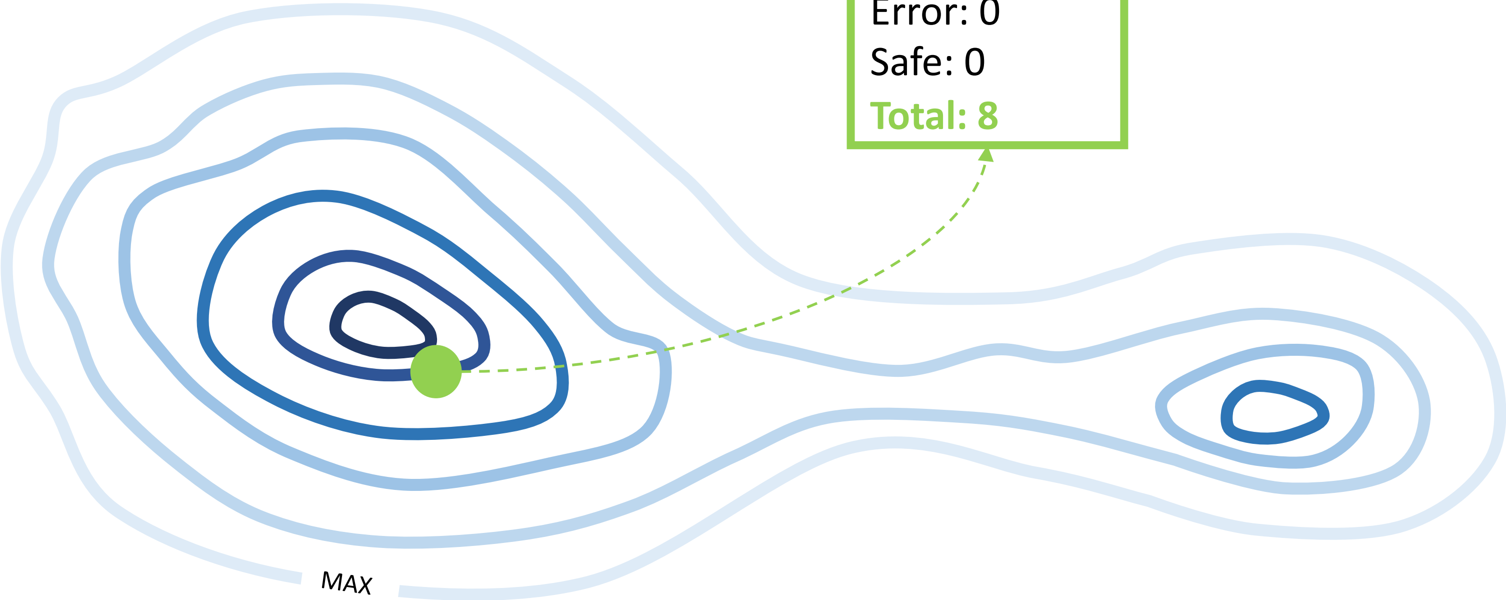
Stochastic search in K2

Iteration 4

With more iterations, the best programs have lower cost!

Current program

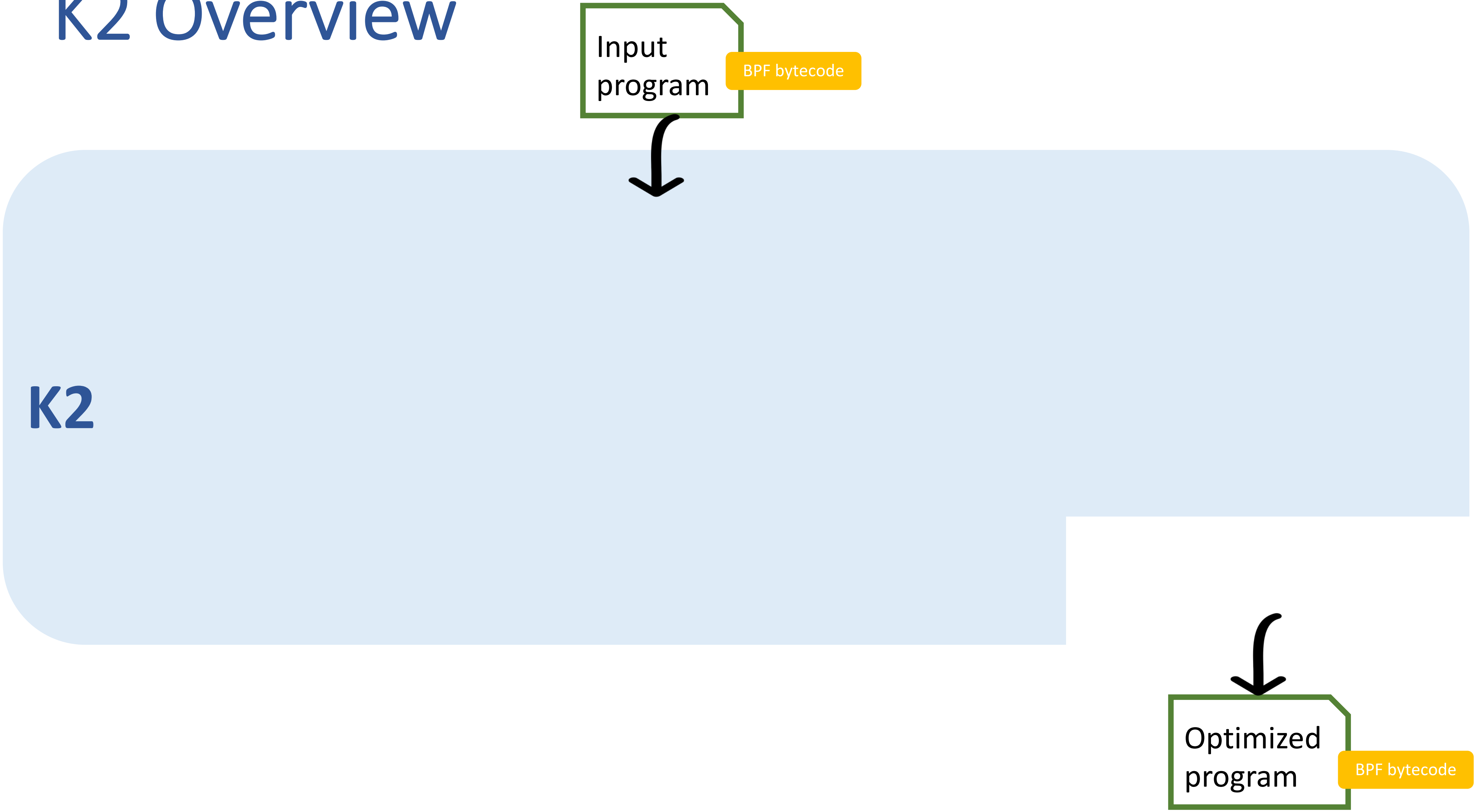
Perf: 8
Error: 0
Safe: 0
Total: 8



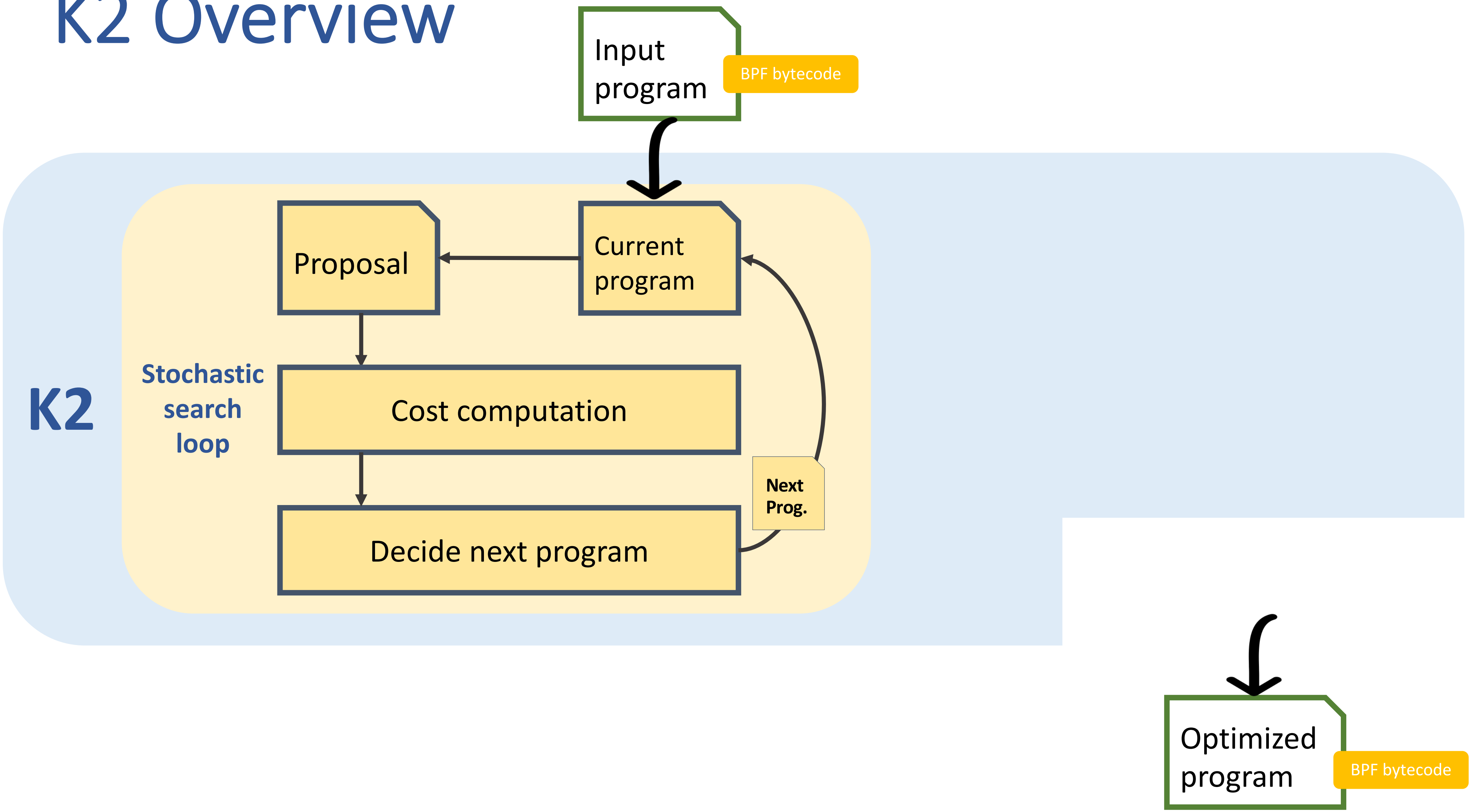
Cost contour line: the darker line, the lower cost

Total cost = Perf + Error + Safe

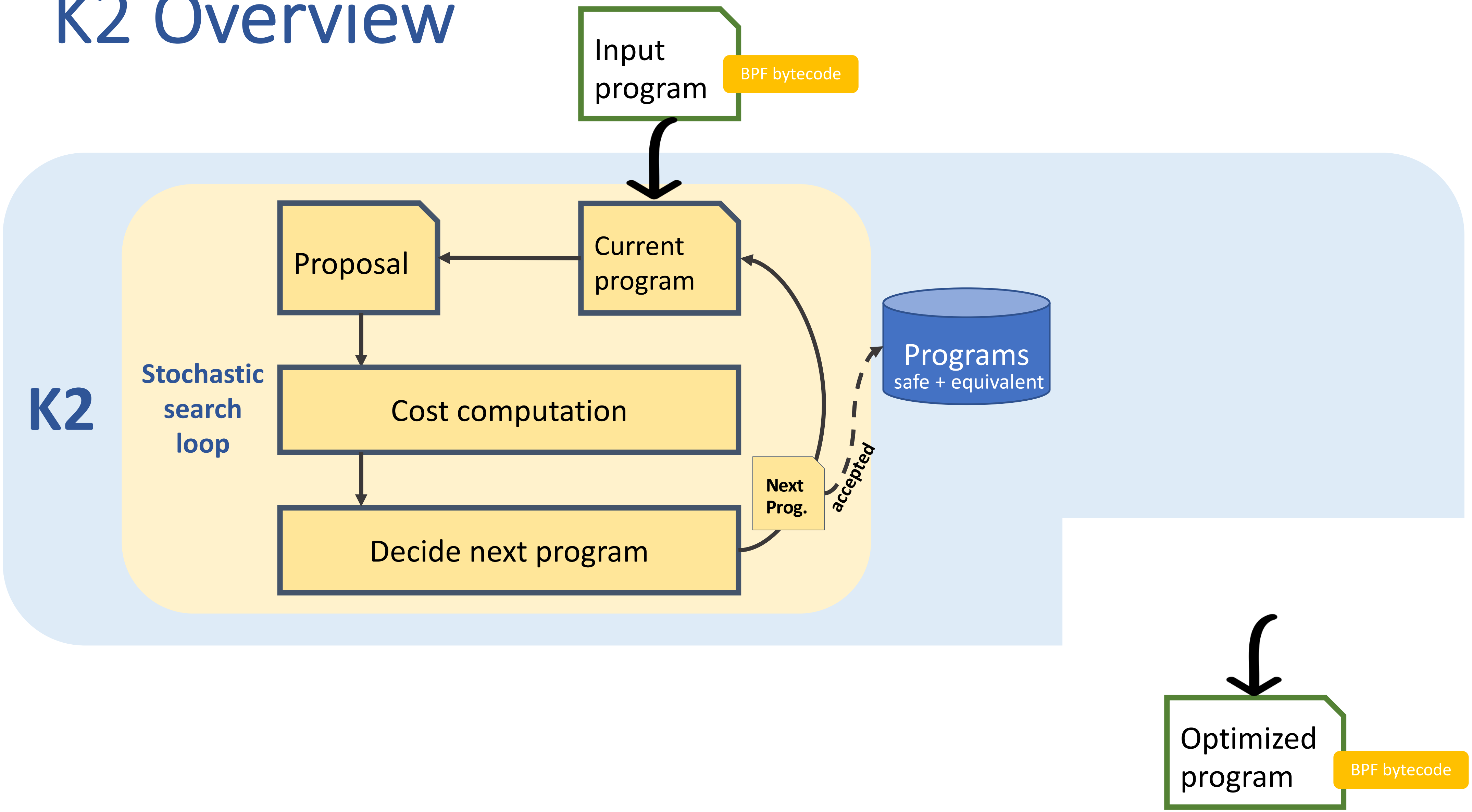
K2 Overview


Input
program
BPF bytecode
K2
Optimized
program
BPF bytecode

K2 Overview

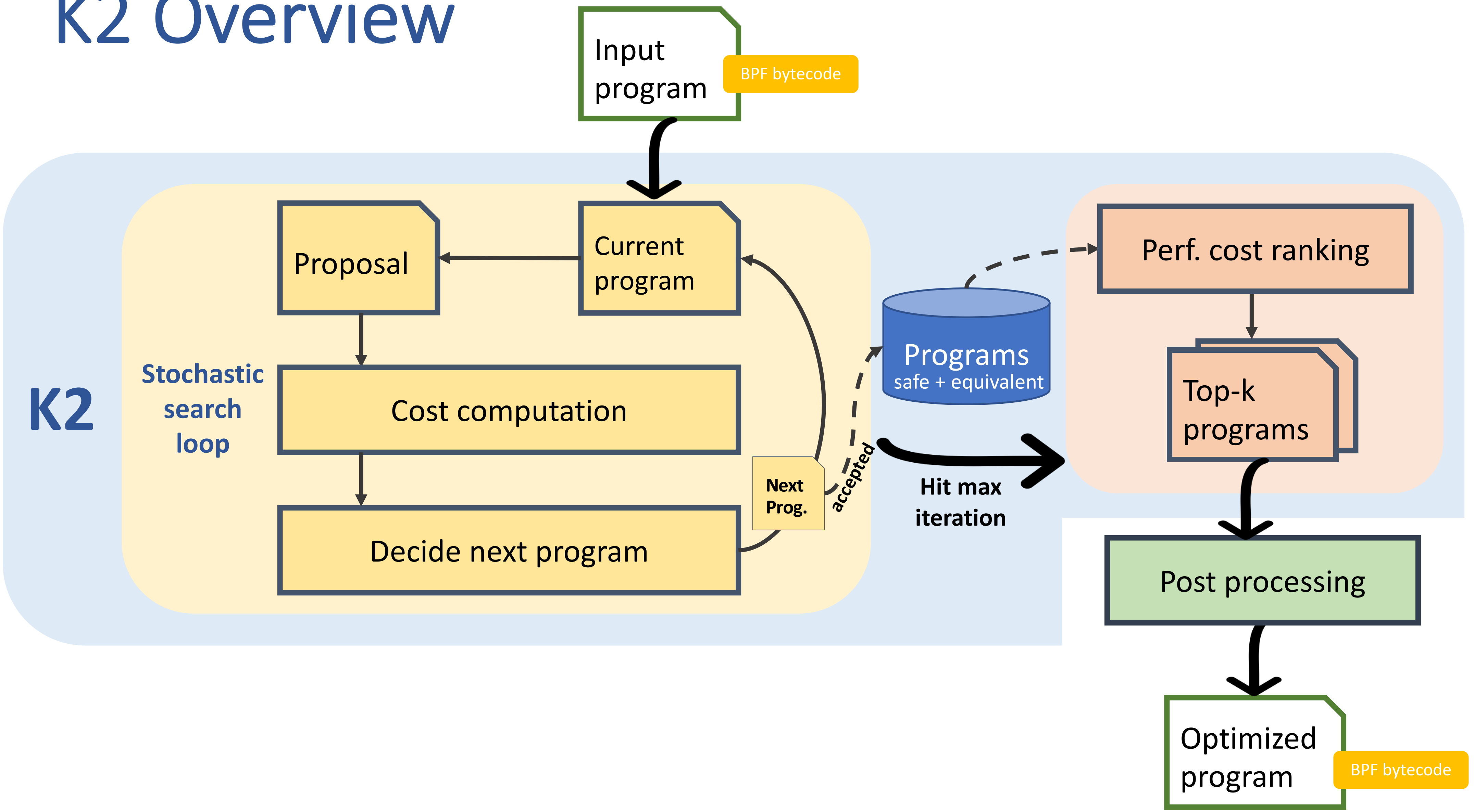

Input program
BPF bytecode
K2
Stochastic search loop
Proposal
Current program
Cost computation
Decide next program
Next Prog.
Optimized program
BPF bytecode

K2 Overview

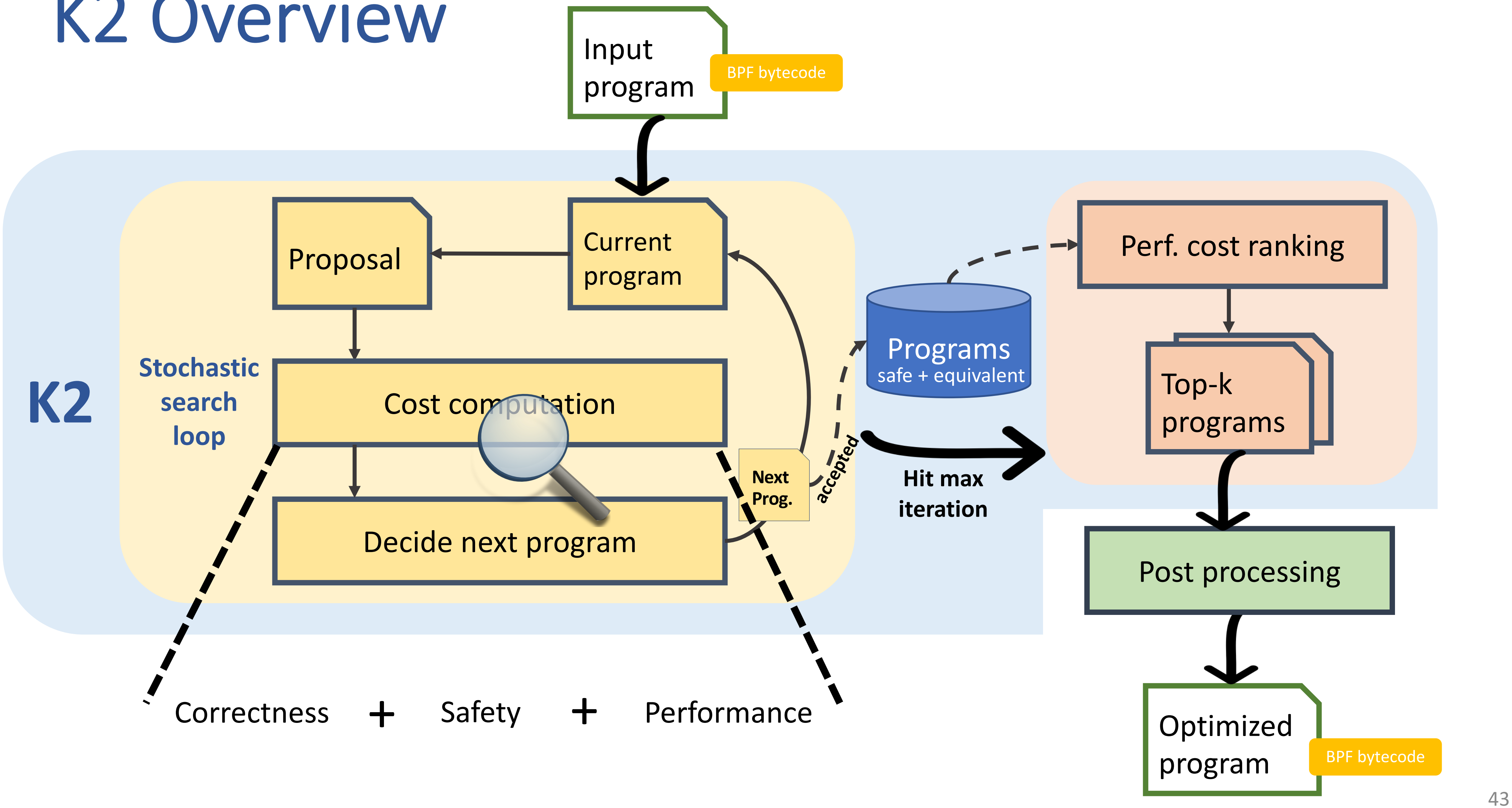


Input
program
BPF bytecode
K2
Stochastic
search
loop
Proposal
Current
program
Cost computation
Decide next program
Next
Prog.
accepted
Programs
safe + equivalent
Optimized
program
BPF bytecode

K2 Overview

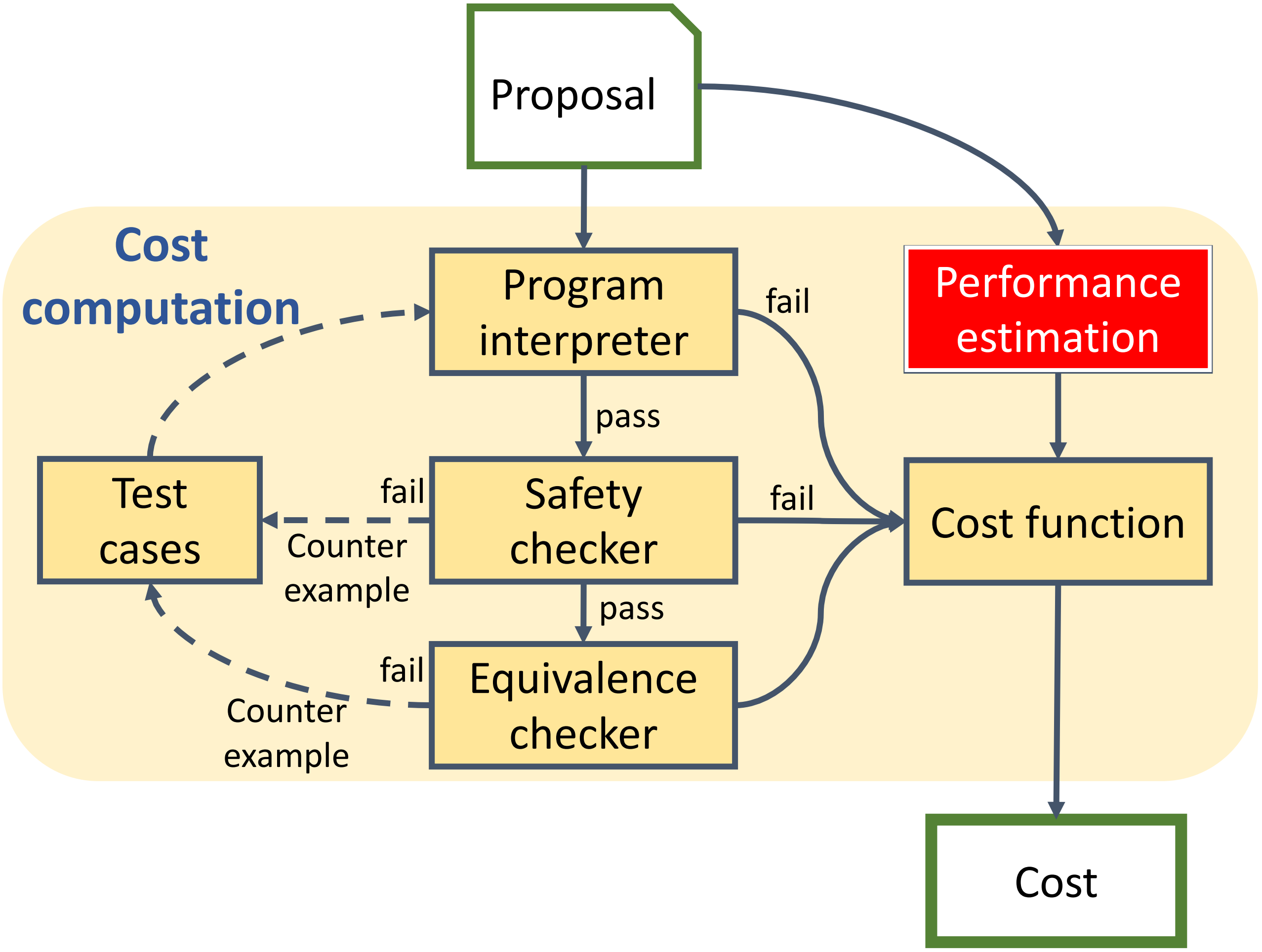

Input
program
BPF bytecode
K2
Stochastic
search
loop
Proposal
Current
program
Cost computation
Decide next program
Next
Prog.
accepted
Programs
safe + equivalent
Hit max
iteration
Perf. cost ranking
Top-k
programs
Post processing
Optimized
program
BPF bytecode

K2 Overview



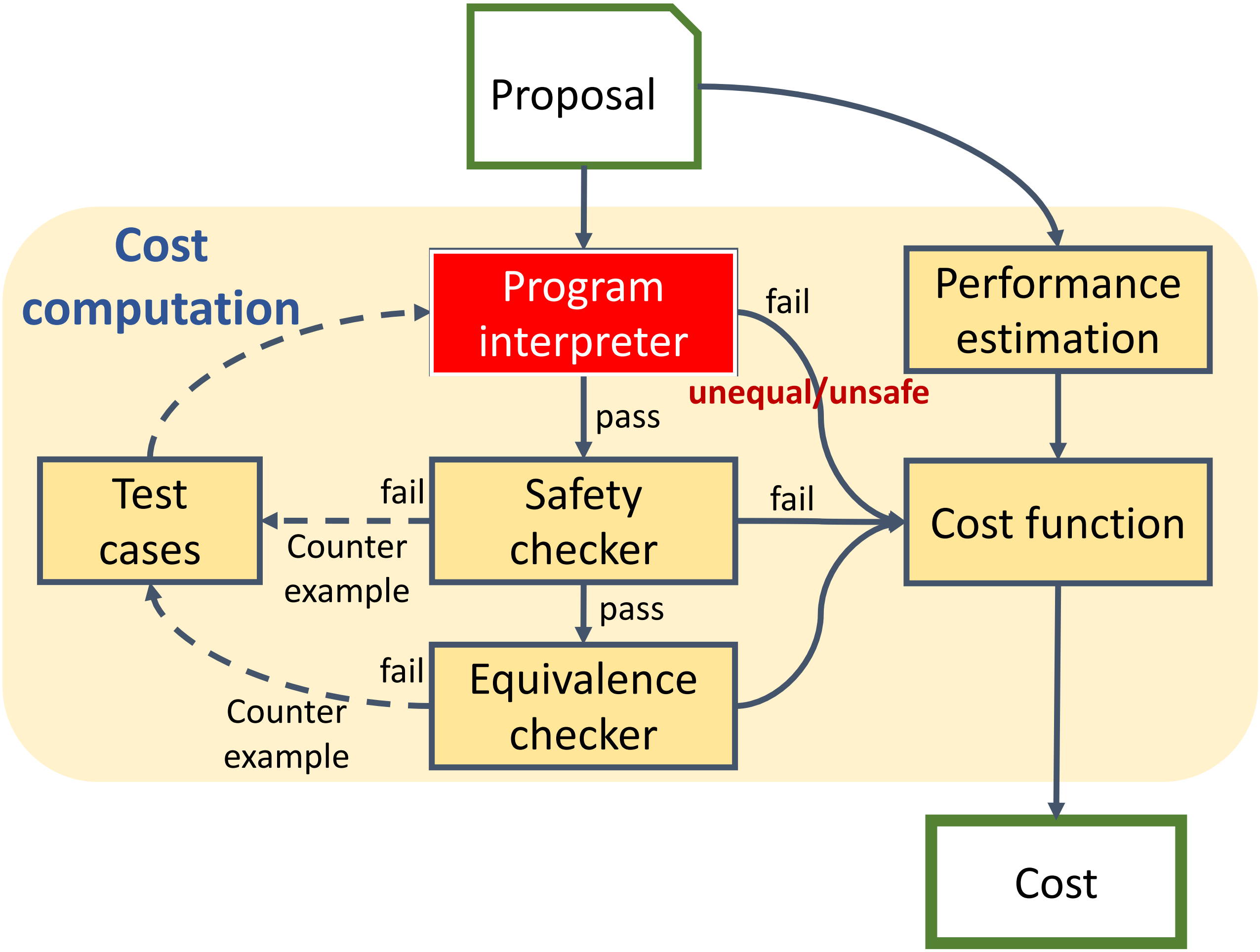
Input program
BPF bytecode
K2
Stochastic search loop
Proposal
Current program
Cost computation
Decide next program
Next Prog.
accepted
Programs
safe + equivalent
Hit max iteration
Perf. cost ranking
Top-k programs
Post processing
Optimized program
BPF bytecode
Correctness + Safety + Performance

Computing cost



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

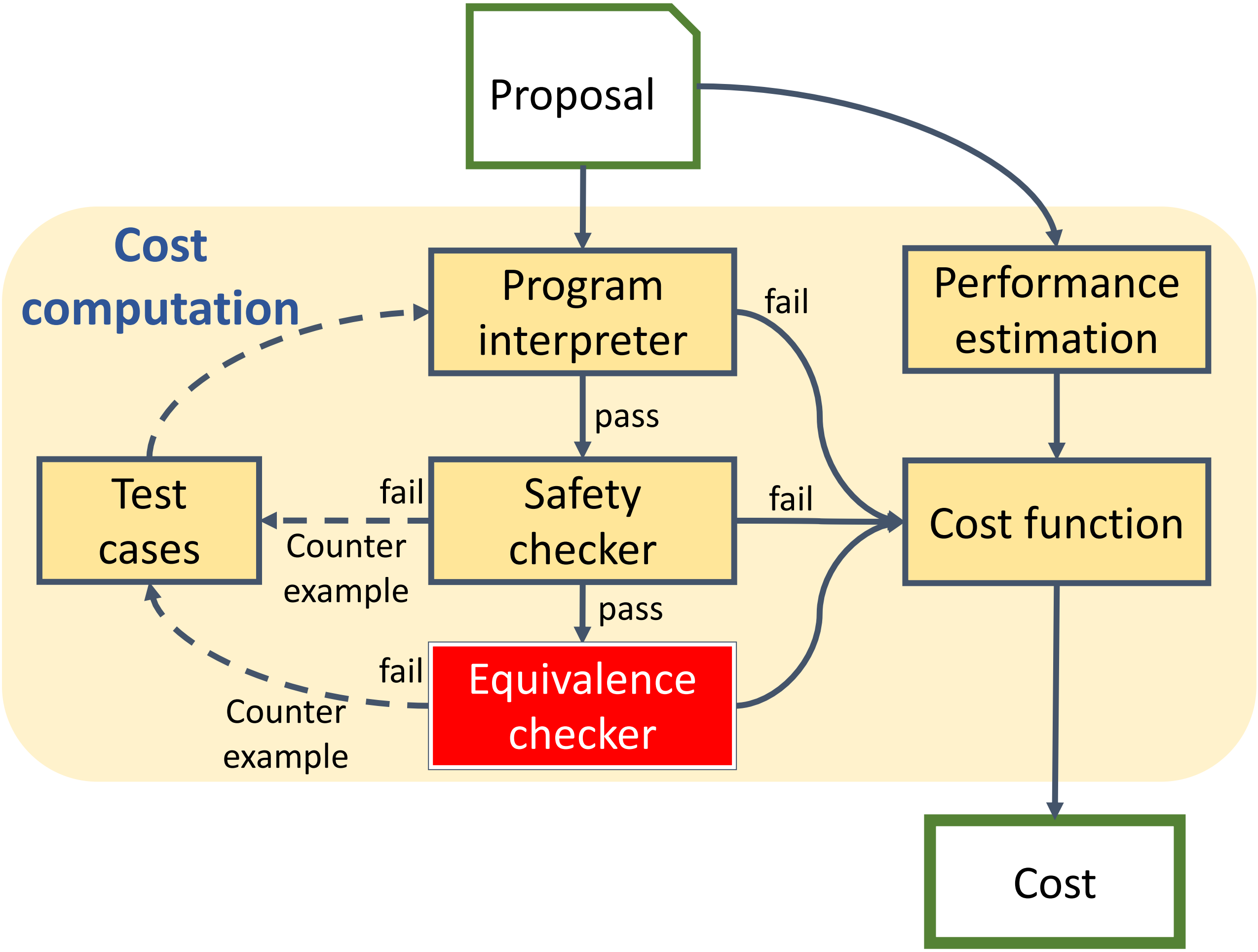
Computing cost



- 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

inputs to program 1 == inputs to program2
$\wedge$ input-output behavior of program 1
$\wedge$ input-output behavior of program 2
$\Rightarrow$ outputs of program 1 != outputs of program2

Equivalence check

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

input: r1, r10
output: r0

program 1

```
r0 = r1
exit
```

program 2

```
r0 = r10
exit
```

inputs to program 1 == inputs to program2
∧ input-output behavior of program 1
∧ input-output behavior of program 2
⇒ outputs of program 1 != outputs of program2

r1_p1 == r1_p2
∧ r10_p1 == r10_p2

∧ r0_p1 == r1_p1

∧ r0_p2 == r10_p2

⇒ r0_p1 != r0_p2

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 program2
∧ input-output behavior of program 1
∧ input-output behavior of program 2
⇒ outputs of program 1 != outputs of program2

program 1

```
r0 = r1
exit
```

program 2

```
r0 = r10
exit
```

r1_p1 == r1_p2
∧ r10_p1 == r10_p2

∧ r0_p1 == r1_p1

∧ r0_p2 == r10_p2

⇒ r0_p1 != r0_p2

SMT solver

SAT: program1 != program 2
model:
r1_p1 = 1, r10_p1 = 2
r1_p2 = 1, r10_p2 = 2
(same inputs)
r0_p1 = 1
r0_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)

> 24 hours → **Selective concretization** → 1 minute → **Modular verification** → 7 milliseconds

Selective concretization:

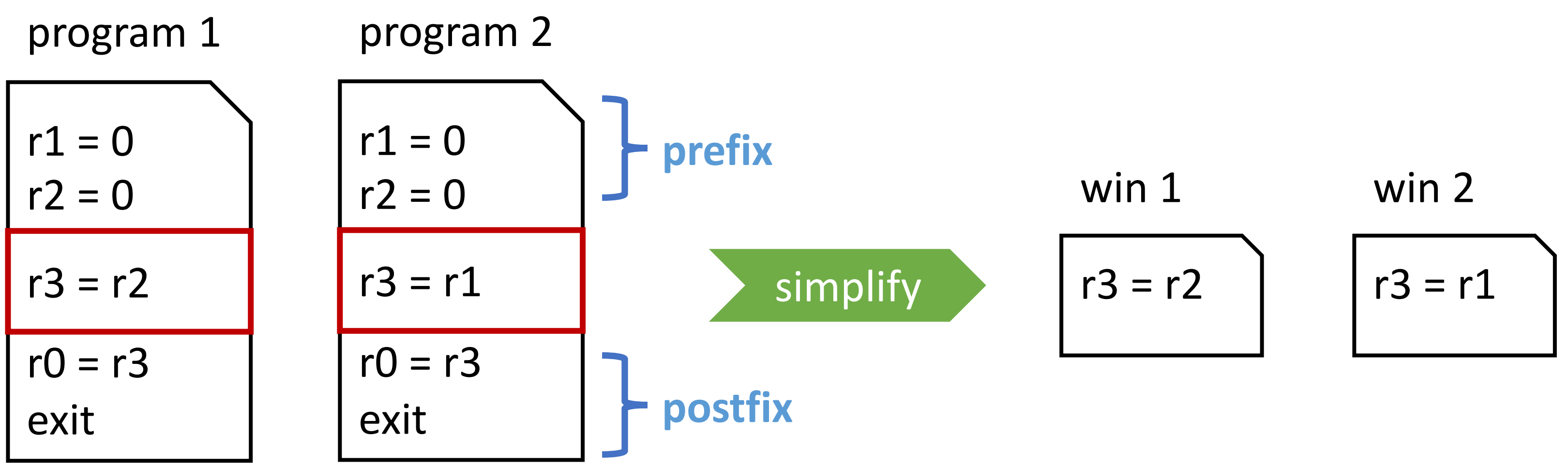
- memory type concretization
- map type concretization
- memory offset concretization

Modular verification:

- check two windows instead of two programs

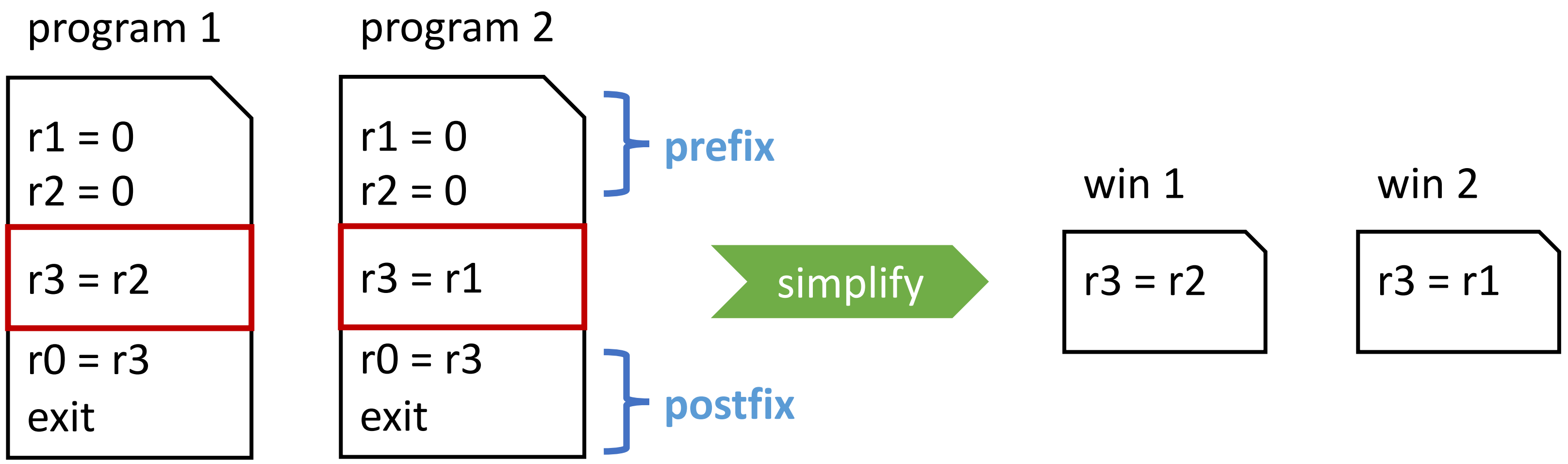
Fast equivalence check: Modular verification

- Suppose programs only differ in a small window of instructions



Fast equivalence check: Modular verification

- Suppose programs only differ in a small window of instructions

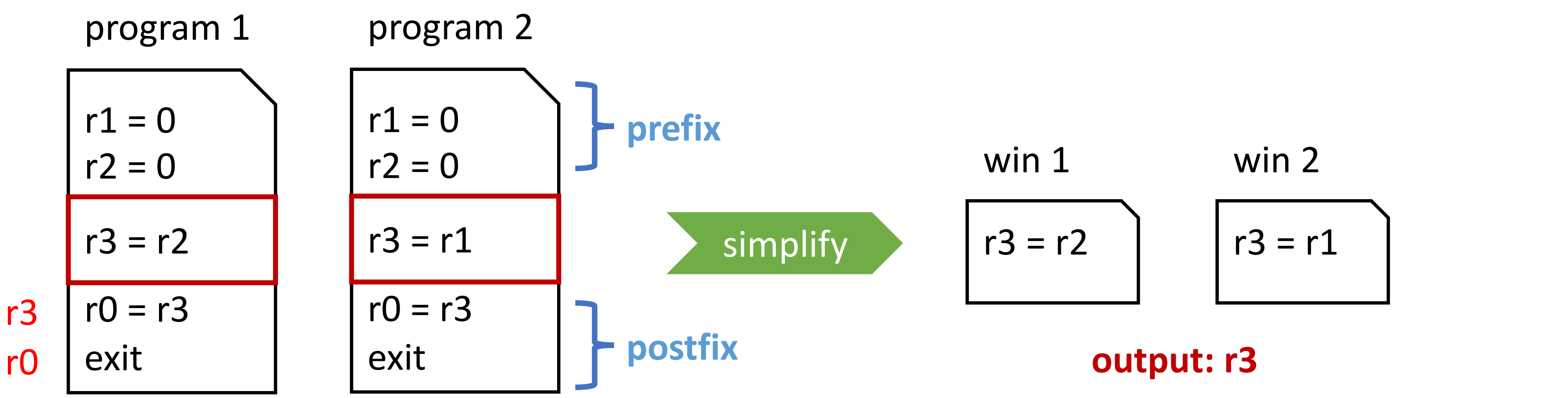


What are output variables to be compared?

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

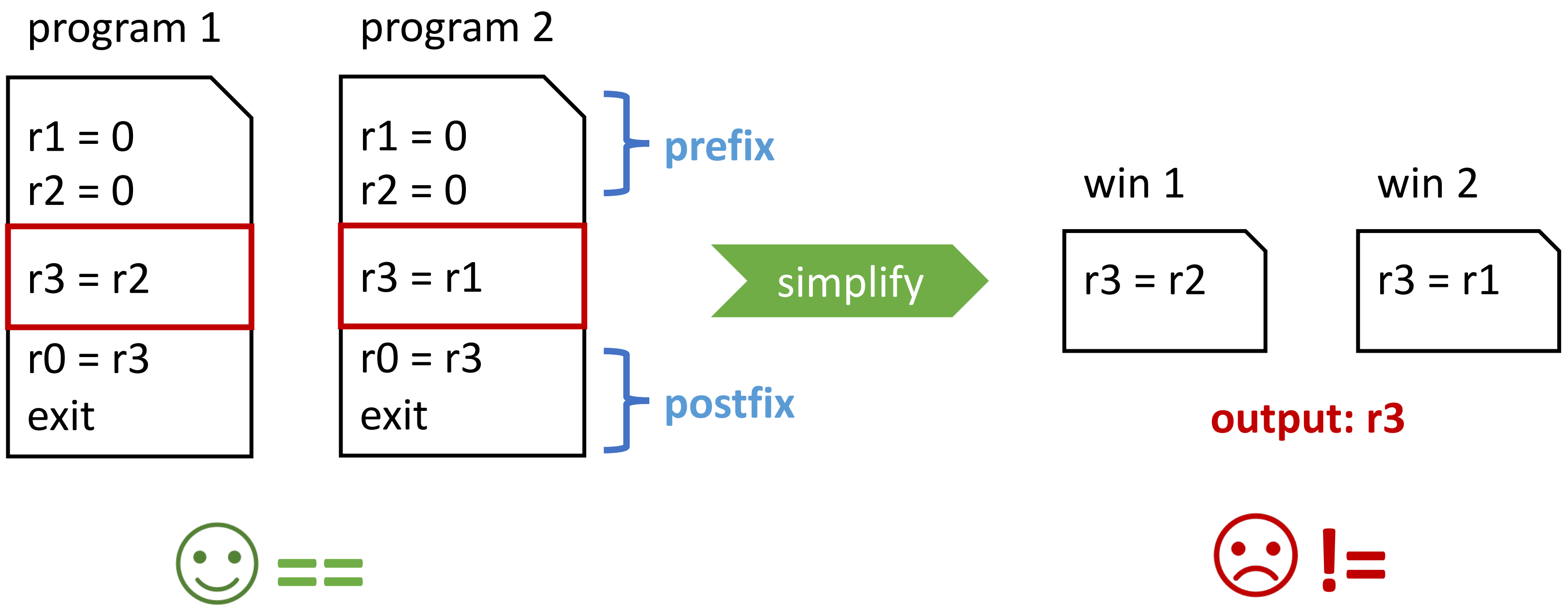


What are output variables to be compared?

Live variables out of the window
(infer from the postfix program)

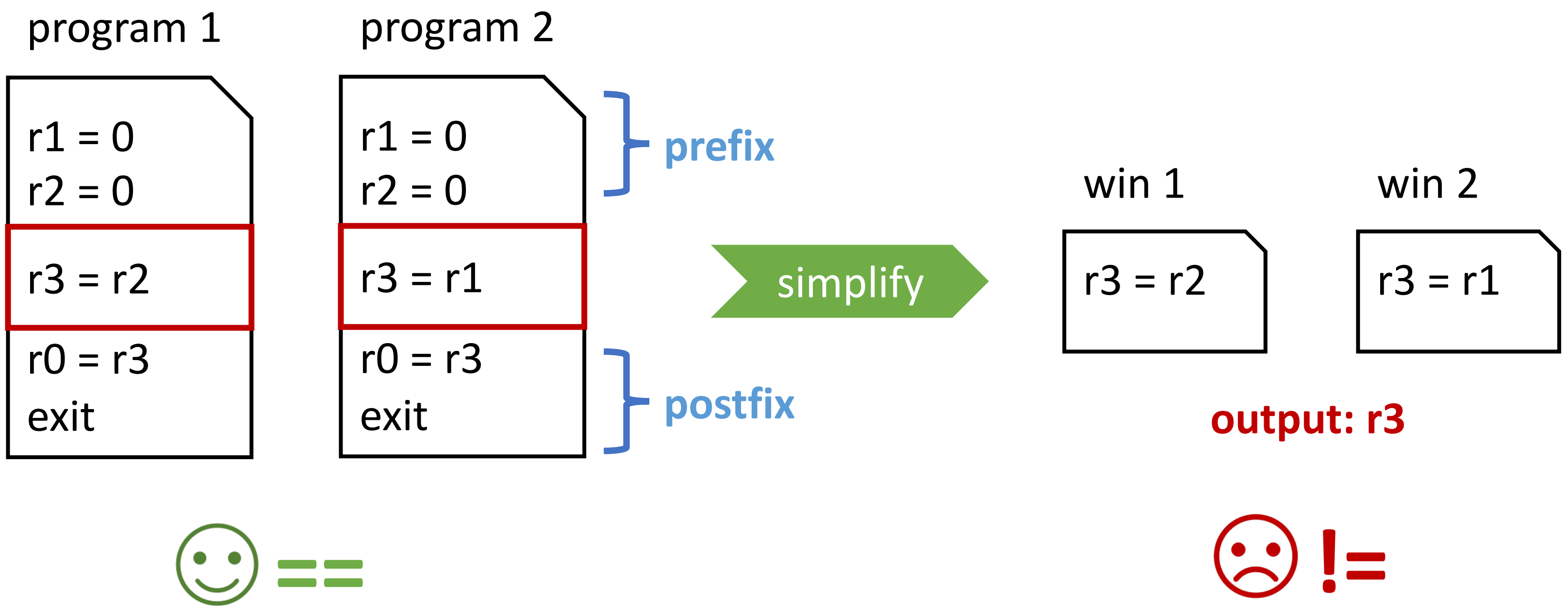
Fast equivalence check: Modular verification

- Suppose programs only differ in a small window of instructions



Fast equivalence check: Modular verification

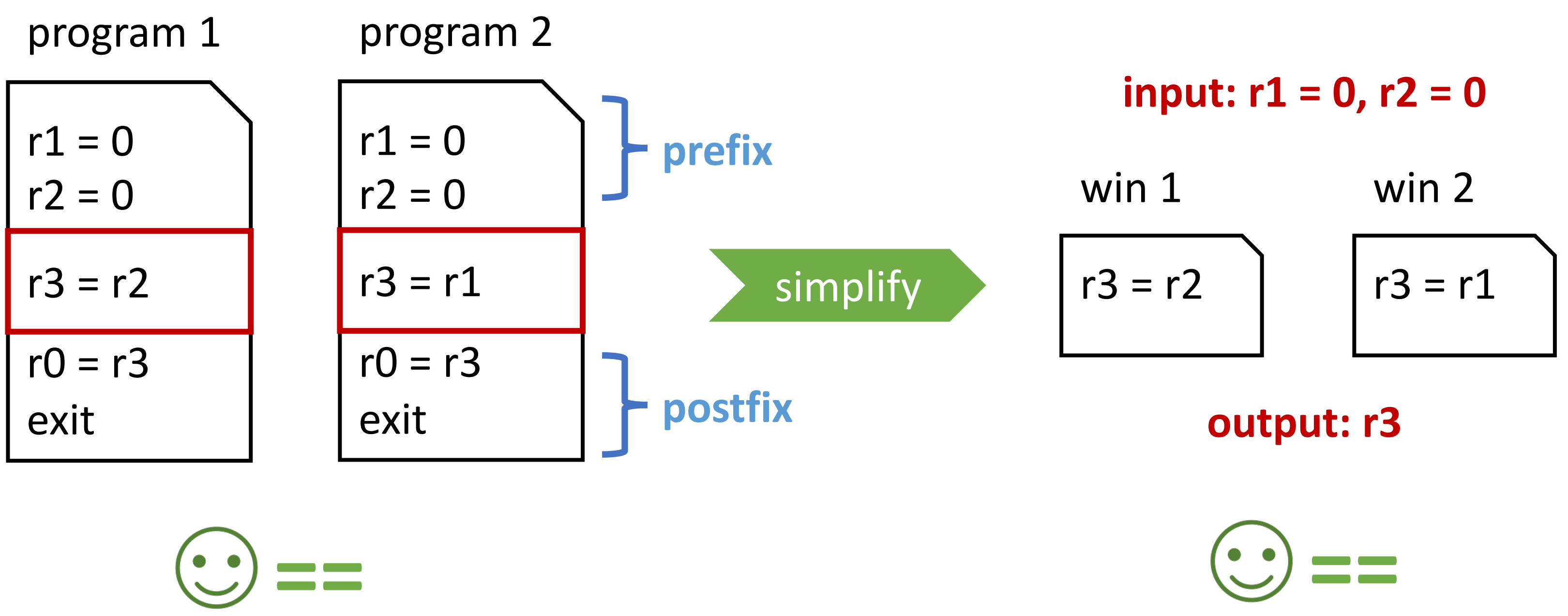
- Suppose programs only differ in a small window of instructions



Infer input variables and preconditions from the prefix program

Fast equivalence check: Modular verification

- Suppose programs only differ in a small window of instructions



Infer input variables and preconditions from the prefix program

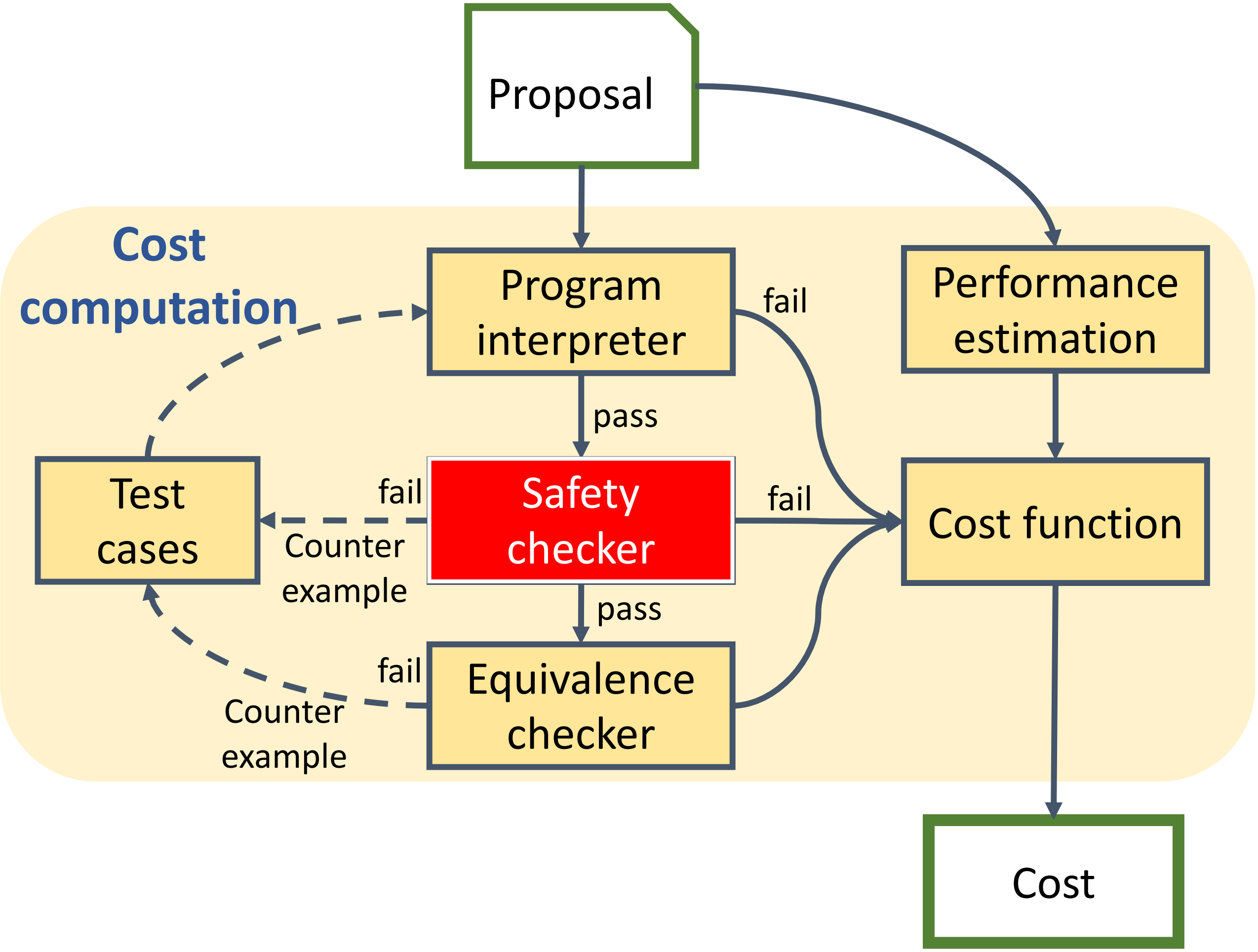
Fast equivalence check: Modular verification

- 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

win input variables preconditions inferred from the prefix program
$\wedge$ variables live into win 1 == variables live into win 2
$\wedge$ input-output behavior of win 1
$\wedge$ input-output behavior of win 2
$\Rightarrow$ variables live out of win 1 != variables live out of win 2

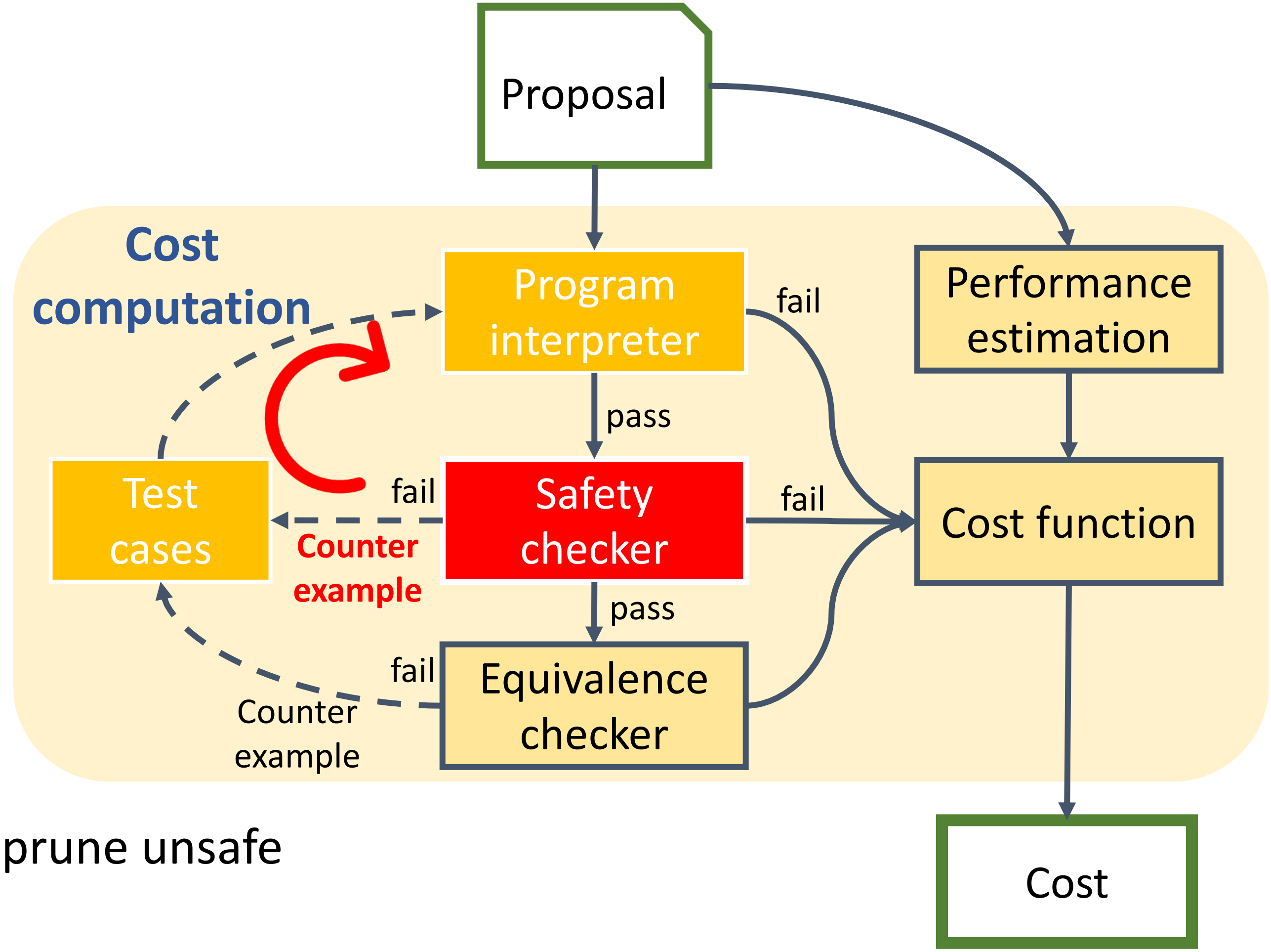
Outline

- 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?

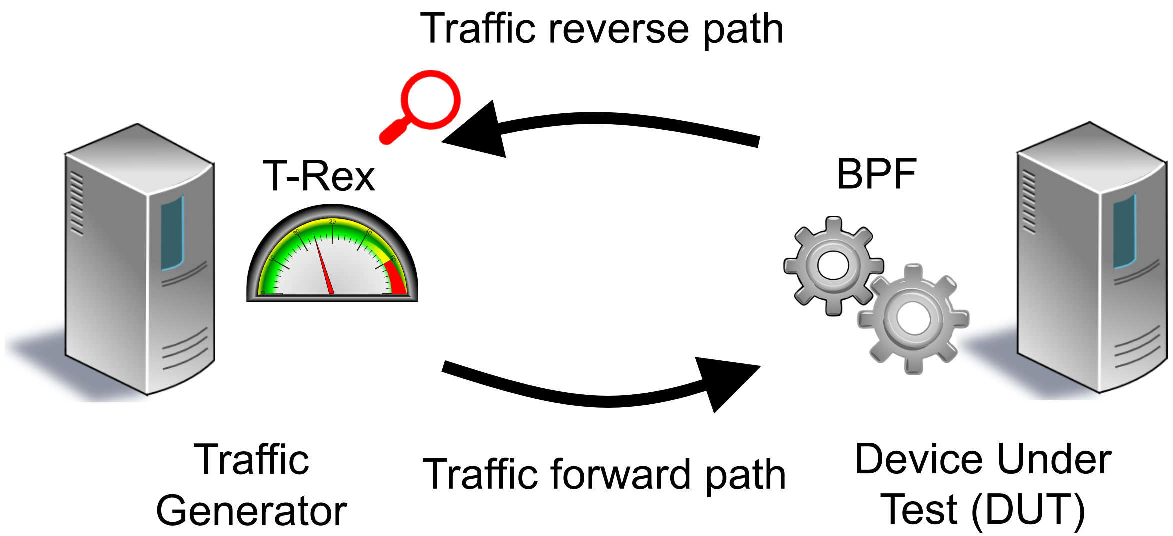
- 19 benchmarks
  - Cilium, Facebook, hXDP, kernel samples
  - Instruction count: 18-1771
- Compression: 6-26%
  - Mean: 13.95%
- Compiling time
  - Mean: 22 minutes (excluding Facebook's Katran xdp-balancer)

Benchmark[1]	Number of instructions			Compiling time (sec)
	clang[2]	K2	Compression	
xdp_router_ipv4	111	99	10.81%	898
xdp_map_access	30	26	13.33%	27
xdp_redirect	43	35	18.60%	523
from-network	39	29	25.64%	6871
xdp_pktcntr	22	19	13.64%	288
xdp-balancer	1771	1607	9.26%	167,428

[1] More benchmark results are in the paper

[2] The smallest program across clang -O1/O2/O3/Os

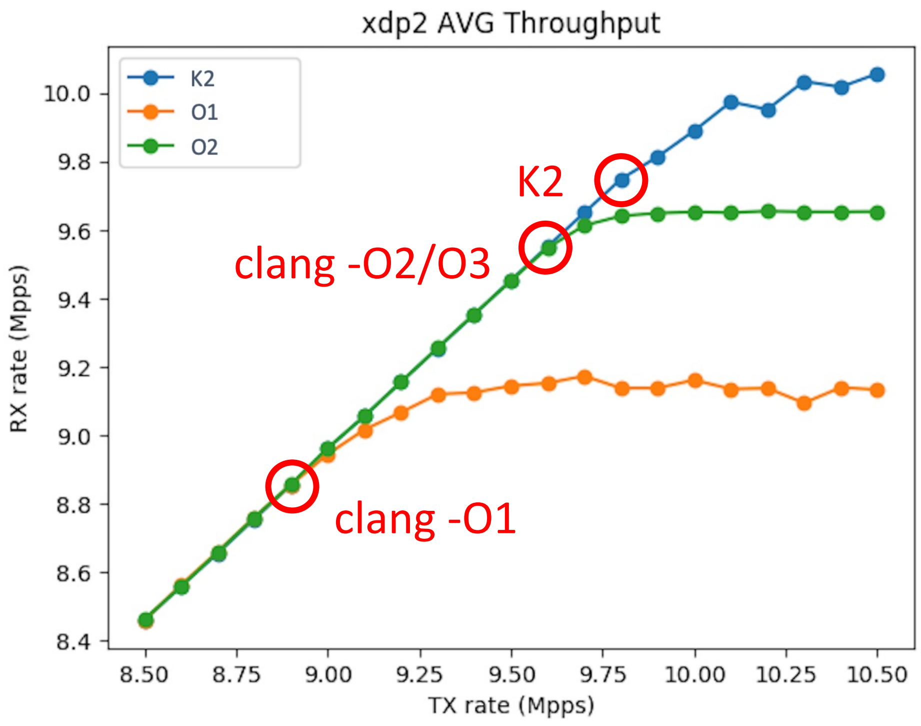
How beneficial is K2 to packet throughput and latency?



- 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



Higher throughput is better

How beneficial is K2 to packet throughput and latency?

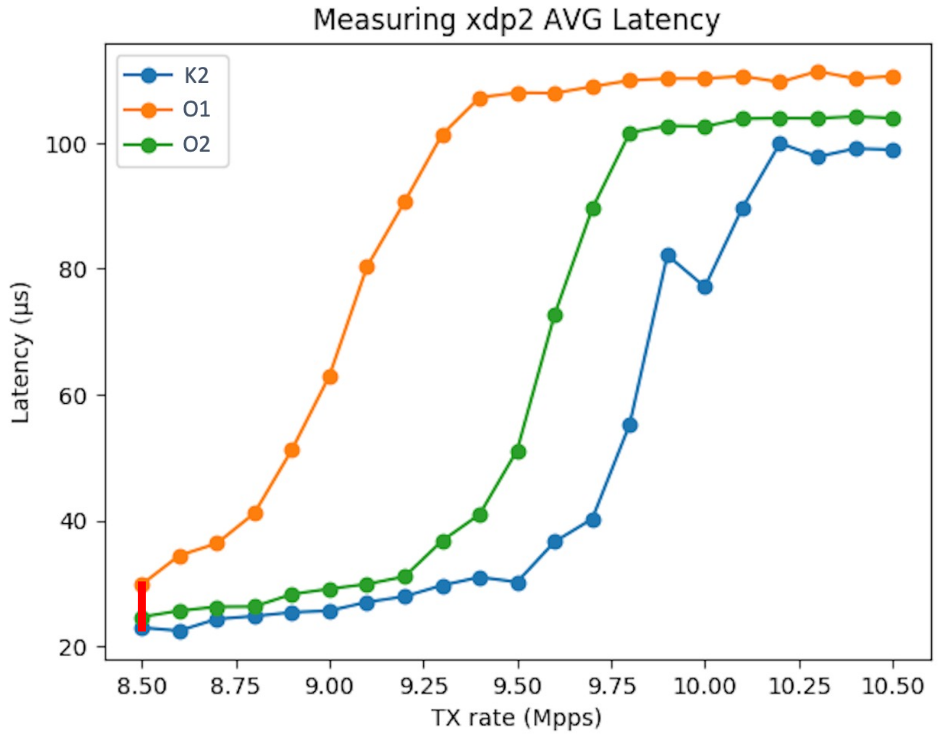
- Throughput: the maximum loss-free forwarding rate (MLFFR) in Mpps (millions of packets per second) per core
- Avg. throughput improvement across 6 benchmarks: 0–4.75%

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

* 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

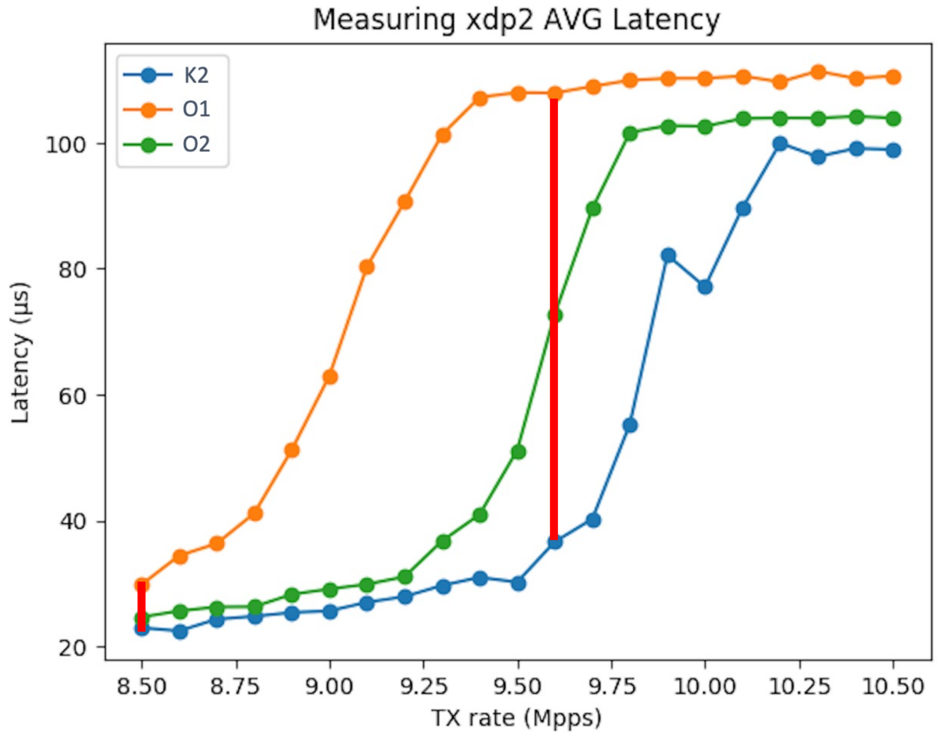


Smaller latency is better

- 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

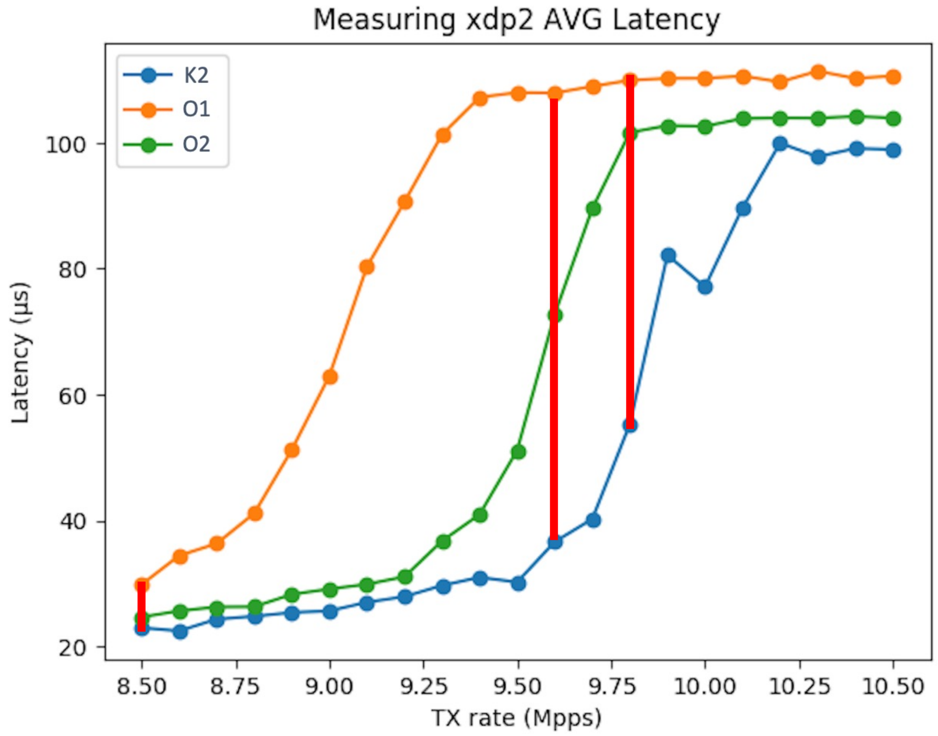


Smaller latency is better

- 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

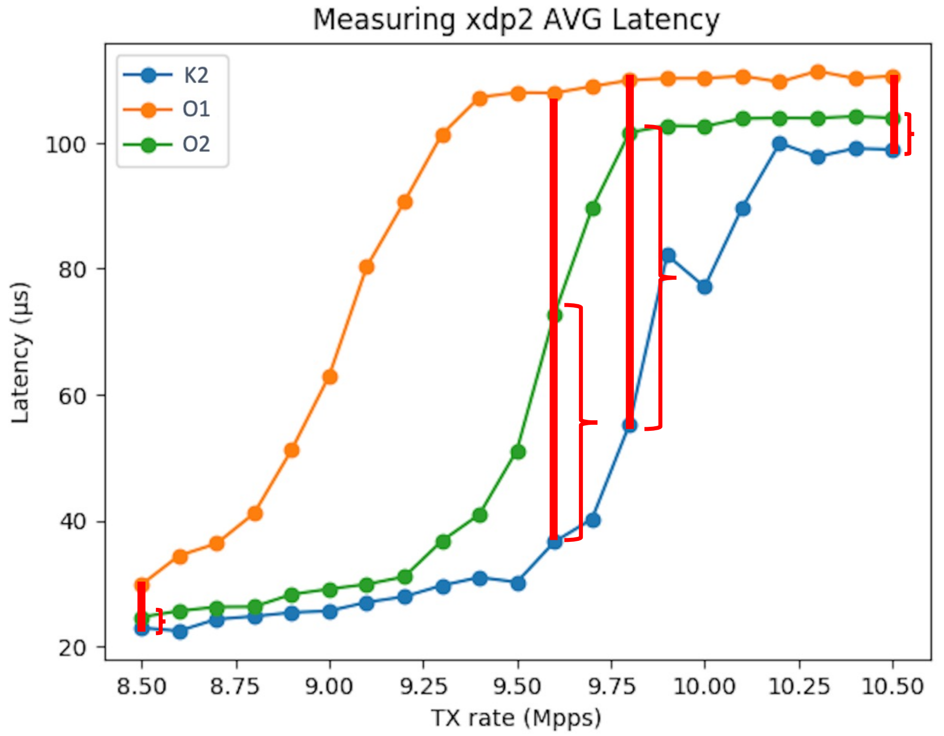


Smaller latency is better

- 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_pktcntr)

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

two 32-bit writes

```
*(u64*)(r10-8) = 0
```

one 64-bit write

Optimizations discovered by K2

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

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

→

```
*(u64*)(r10-8) = 0
```

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

```
r0 = r2
r0 = r0 & r3
r0 = r0 >> 21
```

→

```
r0 = lower32(r2)
r0 = r0 >> 21
```

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