A proof-carrying approach to building correct and flexible in-kernel verifiers

***Luke Nelson*, Xi Wang, Emina Torlak**

Paul G. Allen School

University of Washington

LINUX PLUMBERS CONFERENCE September 20-24, 2021

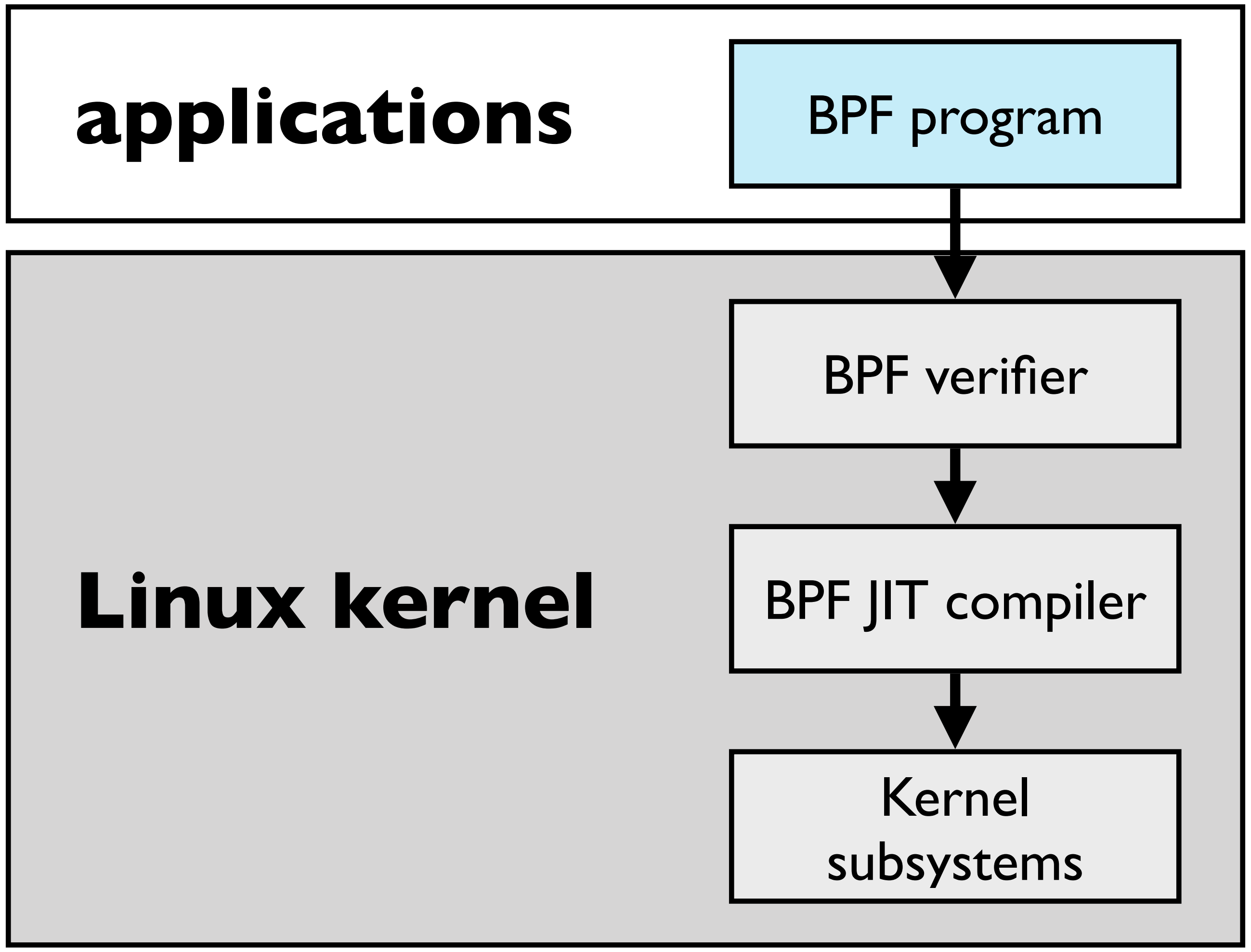
Applying formal methods to the BPF ecosystem

BPF enables applications to extend the Linux kernel

Bugs are critical: BPF programs run in kernel address space

Last year: Improving the BPF JITs using formal verification

This year: How to improve the BPF verifier?



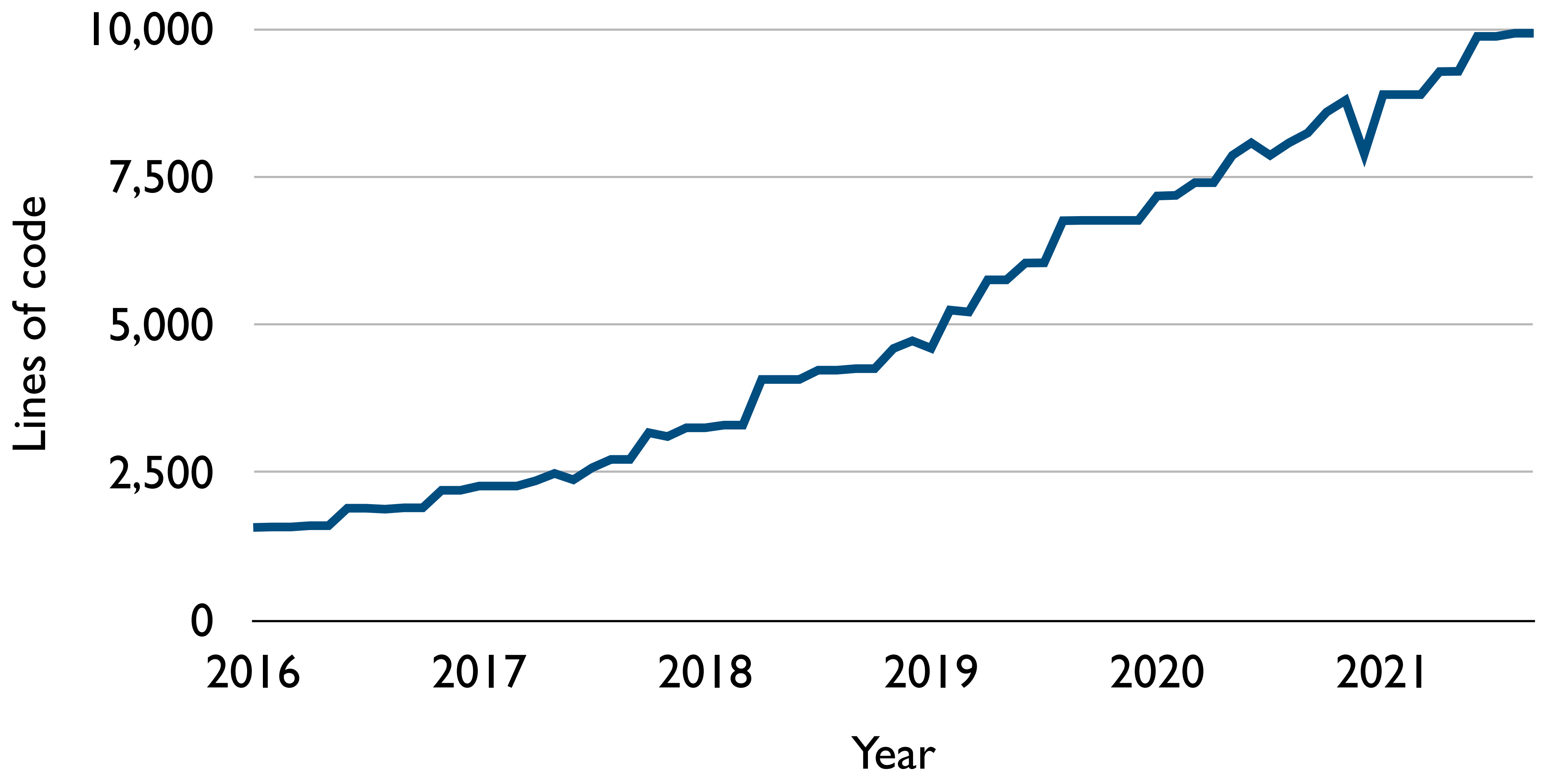
The BPF verifier's complexity is growing

The BPF verifier prevents unsafe programs from running

Static analyzer for BPF programs in the kernel

verifier.c is ≈10,000 lines of code, and growing

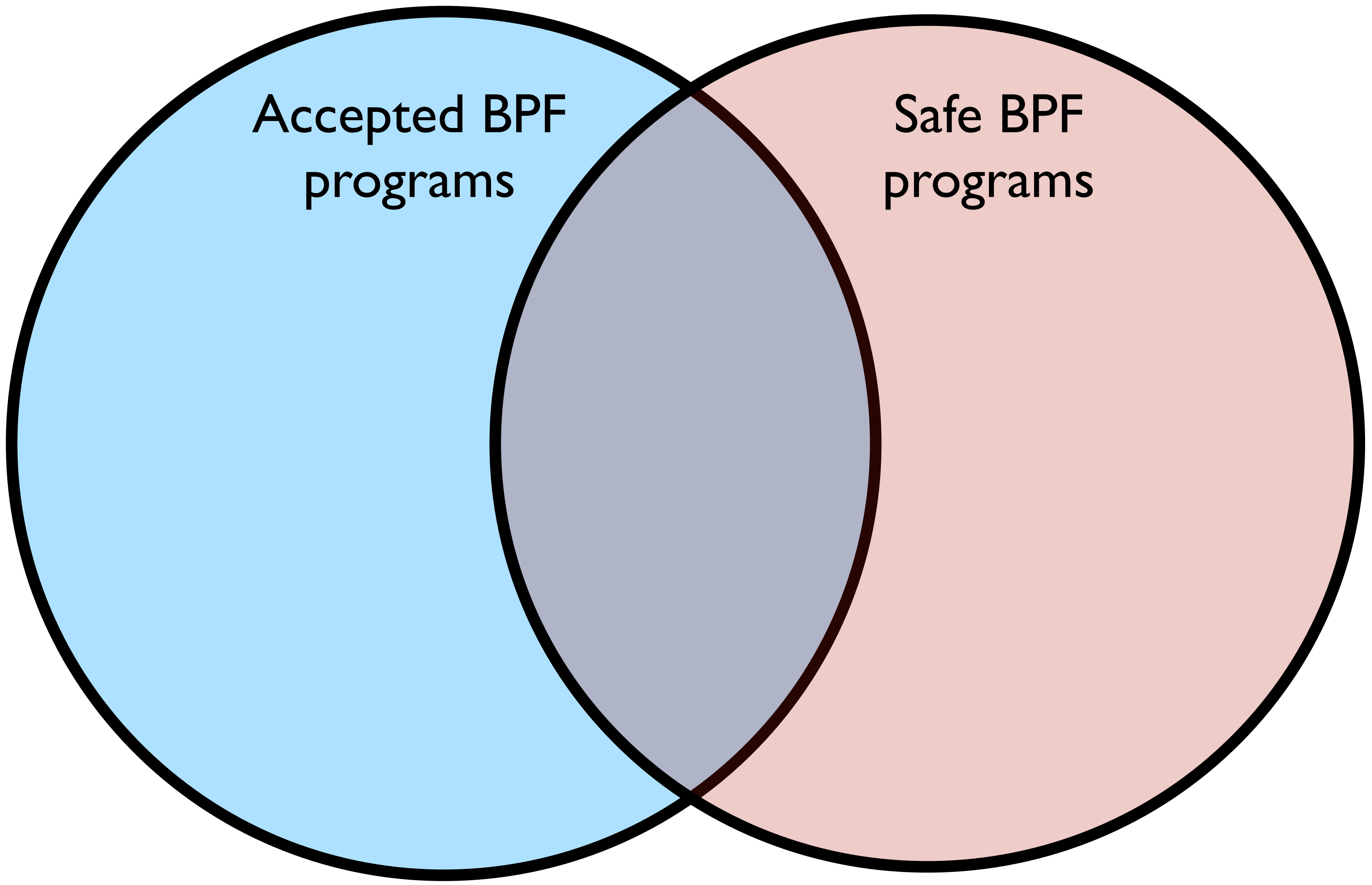
verifier.c lines of code 2016–2021



Verifier complexity leads to two kinds of issues

Correctness bugs: verifier accepts some unsafe programs

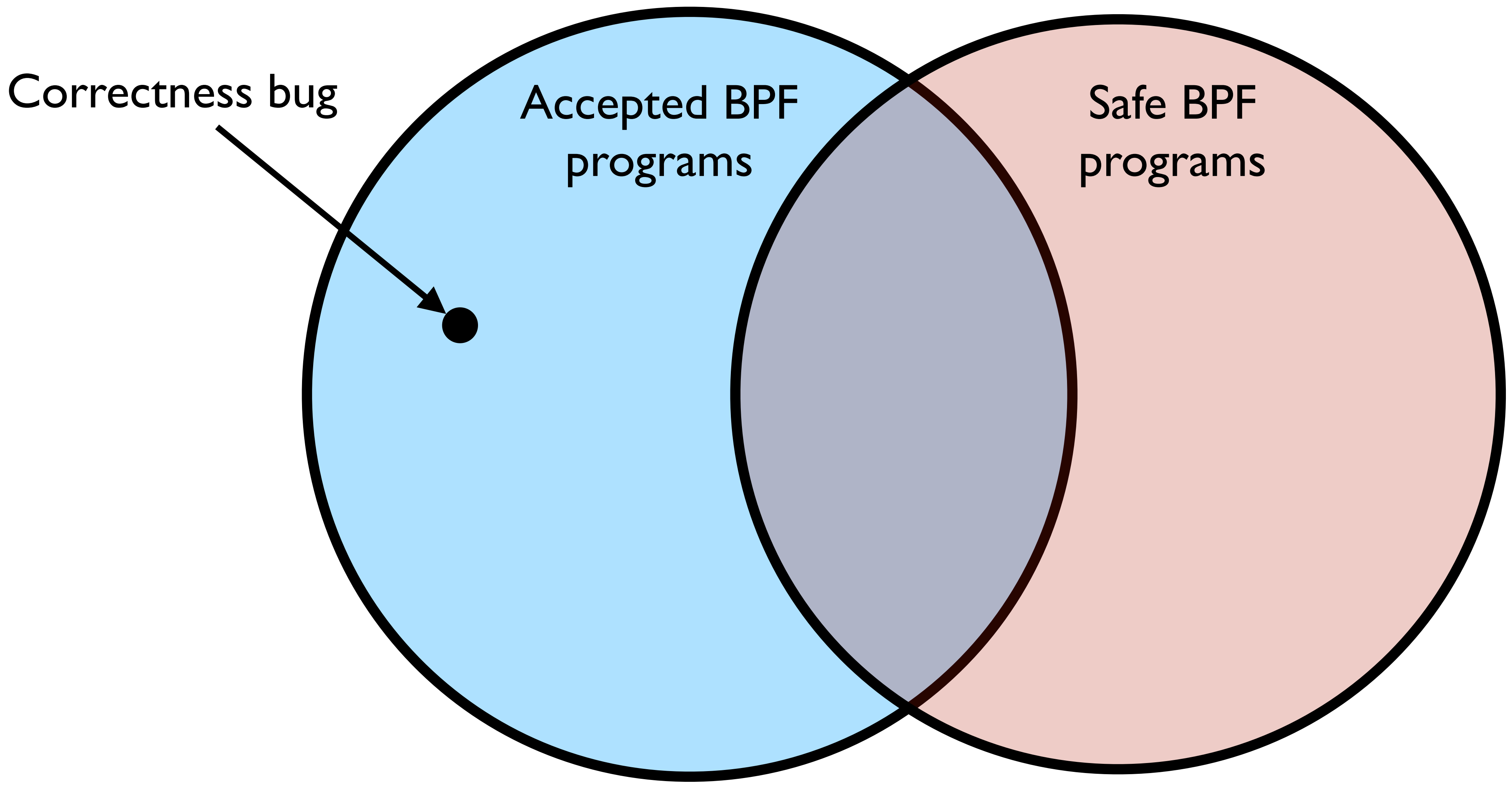
Flexibility issues: verifier rejects some safe programs



Verifier complexity leads to two kinds of issues

Correctness bugs: verifier accepts some unsafe programs

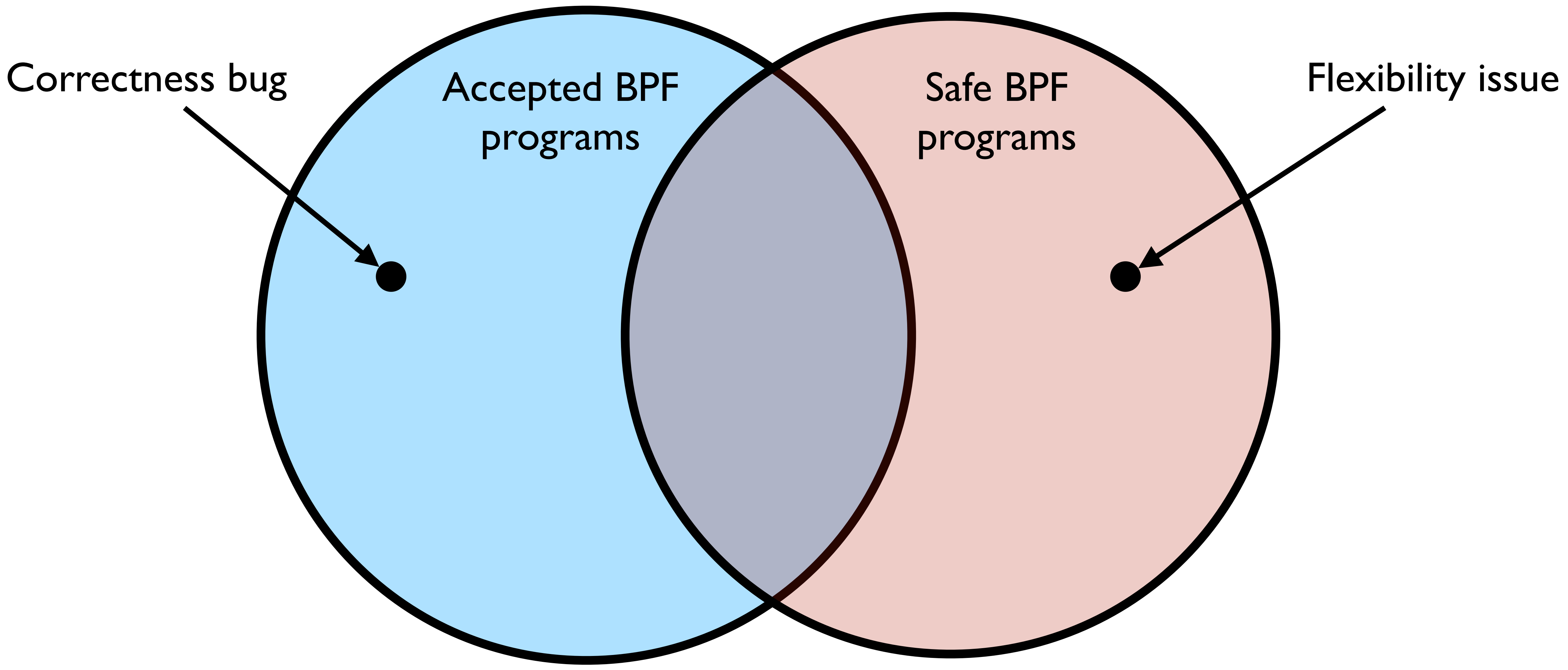
Flexibility issues: verifier rejects some safe programs



Verifier complexity leads to two kinds of issues

Correctness bugs: verifier accepts some unsafe programs

Flexibility issues: verifier rejects some safe programs



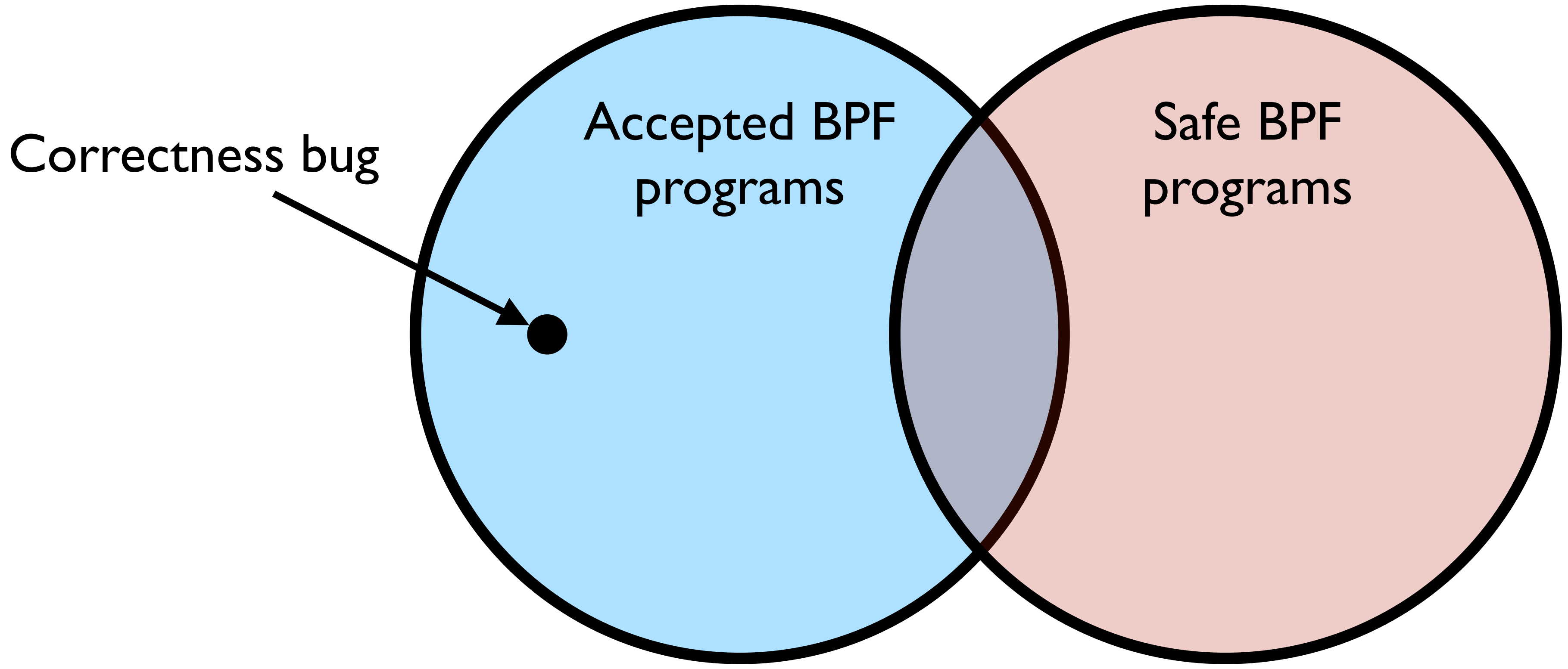
Correctness bugs in the BPF verifier

Verifier bugs are hard to find and fix

- 10 CVEs in 2021
- Bug fixes can introduce new bugs themselves

Writing a correct static analysis is hard

Reasoning about C code which reasons about BPF programs



The BPF verifier is overly strict

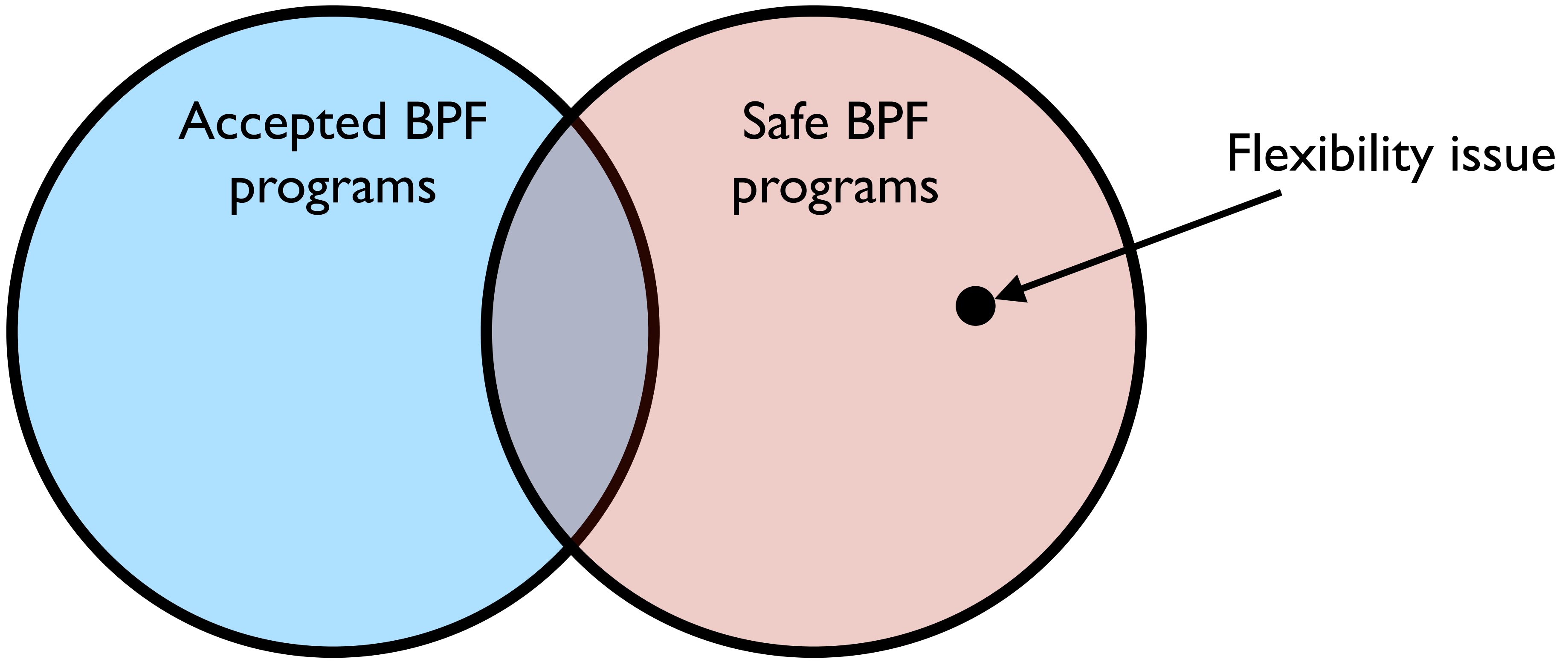
The verifier rejects some safe BPF programs

- Programs that are too complex for verifier to reason about
- Optimizations introduced by LLVM can elude verifier

Hard to write a verifier that accepts all safe programs

BPF program complexity is increasing

Frustrating experience for application developers



“If you've spent any time using eBPF, you must have experienced first hand the dreaded eBPF verifier. It's a merciless judge of all eBPF code that will reject any programs that it deems not worthy of running in kernel-space.”

Jakub Sitnicki (https://blog.cloudflare.com/ebpf-cant-count/)

How are we dealing with these issues now?

Verifier fuzzing and testing

Disabling optimizations in LLVM / tweaking C source code

Extending verifier with more sophisticated analyses

Limitations:

- Search space for testing/fuzzing is very large
- Fixes are brittle as BPF programs and LLVM evolve
- Extending verifier introduces new opportunities for bugs

This talk: exploring an alternative approach to verifiers

Approach: minimize the kernel's job using proof-carrying code

- User-space produces proofs for their BPF programs
- Kernel performs proof checking against a specification

Correctness: cannot fabricate invalid proofs

Flexibility: applications select method of proof generation

Preliminary results: ExoBPF

Work-in-progress: many design and implementation challenges

Use Lean theorem prover for specification, proofs, and proof checker

Two user-space proof generators

- Abstract interpreter mimicking current BPF verifier
- Symbolic execution + SAT solver

Limitations: no rewrites/optimizations, no spectre mitigations

Outline

Bug case study

ExoBPF overview

Demo

Limitations & discussion

Example correctness bug: CVE-2018-18445

BPF semantics for 32-bit right shift instruction:

- `dst = (u32)dst >> 31`

Verifier tracks bounds of `dst` using (`dst_lo`, `dst_hi`)

Bug: verifier truncates bounds *after* the right shift

- `dst_lo = (u32)(dst_lo >> 31)`
- `dst_hi = (u32)(dst_hi >> 31)`

Can trick the verifier into accepting a program with illegal pointer

Example correctness bug: CVE-2018-18445

```
/* Initially assume r0 points
 * to an 8-byte array */


r2 = 2
r2 = (u64)r2 << 31
r2 = (u32)r2 >> 31
r2 -= 2

r0 += r2
*(u8 *)r0 = 0

exit
```

Runtime value of r2	Verifier bounds (r2_lo, r2_hi)

Example correctness bug: CVE-2018-18445

```
/* Initially assume r0 points
 * to an 8-byte array */


r2 = 2
r2 = (u64)r2 << 31
r2 = (u32)r2 >> 31
r2 -= 2

r0 += r2
*(u8 *)r0 = 0

exit
```

Runtime value of r2	Verifier bounds (r2_lo, r2_hi)
2	(2, 2)

Example correctness bug: CVE-2018-18445

```
/* Initially assume r0 points
 * to an 8-byte array */


r2 = 2
r2 = (u64)r2 << 31
r2 = (u32)r2 >> 31
r2 -= 2

r0 += r2
*(u8 *)r0 = 0

exit
```

Runtime value of r2	Verifier bounds (r2_lo, r2_hi)
2	(2, 2)
0x1'0000'0000	(0x1'0000'0000, 0x1'0000'0000)

Example correctness bug: CVE-2018-18445

```
/* Initially assume r0 points
 * to an 8-byte array */


r2 = 2
r2 = (u64)r2 << 31
r2 = (u32)r2 >> 31
r2 -= 2

r0 += r2
*(u8 *)r0 = 0

exit
```

Runtime value of r2	Verifier bounds (r2_lo, r2_hi)
2	(2, 2)
0x1'0000'0000	(0x1'0000'0000, 0x1'0000'0000)
0	(2, 2)

Example correctness bug: CVE-2018-18445

```
/* Initially assume r0 points
 * to an 8-byte array */


r2 = 2
r2 = (u64)r2 << 31
r2 = (u32)r2 >> 31
r2 -= 2

r0 += r2
*(u8 *)r0 = 0

exit
```

Runtime value of r2	Verifier bounds (r2_lo, r2_hi)
2	(2, 2)
0x1’0000’0000	(0x1’0000’0000, 0x1’0000’0000)
0	(2, 2)
-2	(0, 0)

Example correctness bug: CVE-2018-18445

```
/* Initially assume r0 points
 * to an 8-byte array */


r2 = 2
r2 = (u64)r2 << 31
r2 = (u32)r2 >> 31
r2 -= 2

r0 += r2
*(u8 *)r0 = 0

exit
```

Runtime value of r2	Verifier bounds (r2_lo, r2_hi)
2	(2, 2)
0x1'0000'0000	(0x1'0000'0000, 0x1'0000'0000)
0	(2, 2)
-2	(0, 0)

Unsafe program accepted by verifier: runtime access r0[-2], verifier believes it accesses r0[0]

Example flexibility issue: missing relational bounds

BPF verifier used to reject this program

Fixed by tracking equality among registers

```
/* Initially assume r0 points to
 * 8-byte array, and r2 is an
 * arbitrary scalar. */

r1 = r2

if r1 >= 8 goto out

/* Write to r0[r2] */
r0 += r2
*(u8 *)(r0) = 0

out: exit
```

Example flexibility issue: missing relational bounds

BPF verifier used to reject this program

Fixed by tracking equality among registers

```
/* Initially assume r0 points to
 * 8-byte array, and r2 is an
 * arbitrary scalar. */

r1 = r2

if r1 >= 8 goto out

/* Write to r0[r2] */
r0 += r2
*(u8 *)(r0) = 0

out: exit
```

r1 $\in$ [0, UMAX] r2 $\in$ [0, UMAX]

Example flexibility issue: missing relational bounds

BPF verifier used to reject this program

Fixed by tracking equality among registers

```
/* Initially assume r0 points to
 * 8-byte array, and r2 is an
 * arbitrary scalar. */

r1 = r2

if r1 >= 8 goto out

/* Write to r0[r2] */
r0 += r2
*(u8 *)(r0) = 0

out: exit
```

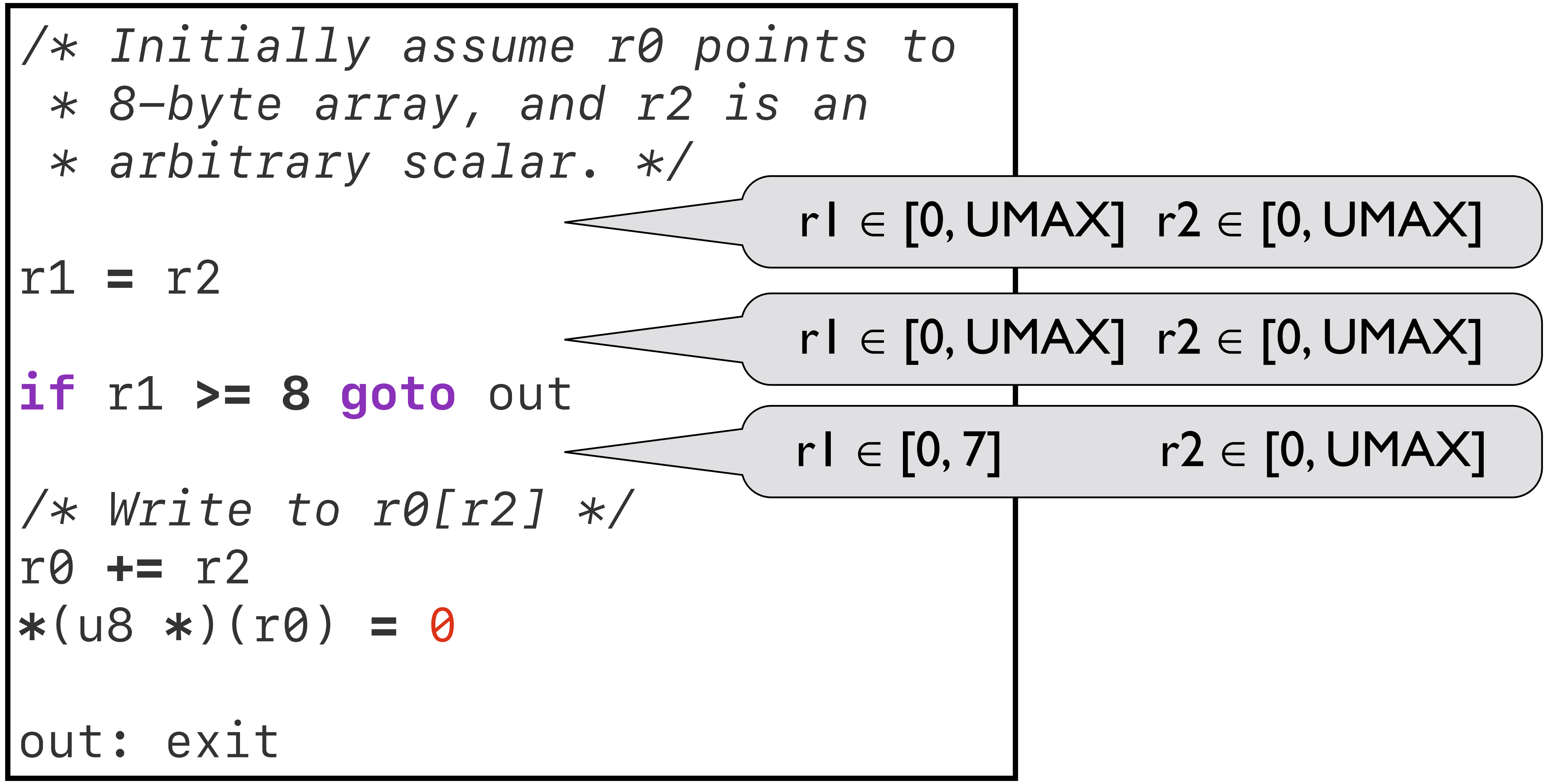
$r1 \in [0, \mathrm{UMAX}]$ $r2 \in [0, \mathrm{UMAX}]$

$r1 \in [0, \mathrm{UMAX}]$ $r2 \in [0, \mathrm{UMAX}]$

Example flexibility issue: missing relational bounds

BPF verifier used to reject this program

Fixed by tracking equality among registers



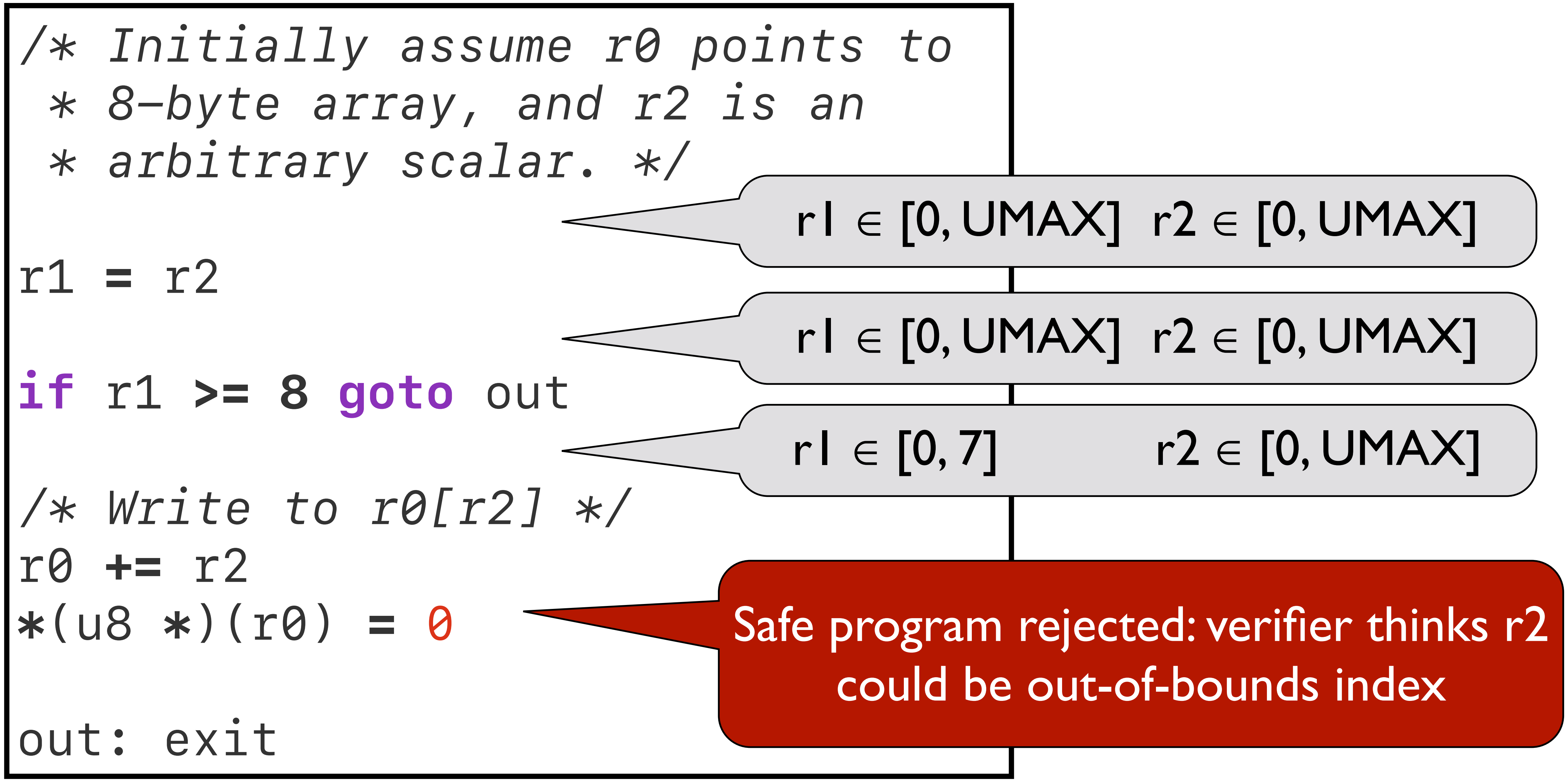
```
/* Initially assume r0 points to
 * 8-byte array, and r2 is an
 * arbitrary scalar. */

r1 = r2

if r1 >= 8 goto out

/* Write to r0[r2] */
r0 += r2
*(u8 *)(r0) = 0

out: exit
```

r1 ∈ [0, UMAX] r2 ∈ [0, UMAX]
r1 ∈ [0, UMAX] r2 ∈ [0, UMAX]
r1 ∈ [0, 7] r2 ∈ [0, UMAX]

Example flexibility issue: missing relational bounds

BPF verifier used to reject this program

Fixed by tracking equality among registers



```
/* Initially assume r0 points to
 * 8-byte array, and r2 is an
 * arbitrary scalar. */

r1 = r2

if r1 >= 8 goto out

/* Write to r0[r2] */
r0 += r2
*(u8 *)(r0) = 0

out: exit
```

$r1 \in [0, UMAX]$ $r2 \in [0, UMAX]$
$r1 \in [0, UMAX]$ $r2 \in [0, UMAX]$
$r1 \in [0, 7]$ $r2 \in [0, UMAX]$
Safe program rejected: verifier thinks r2 could be out-of-bounds index

Summary of verifier issues

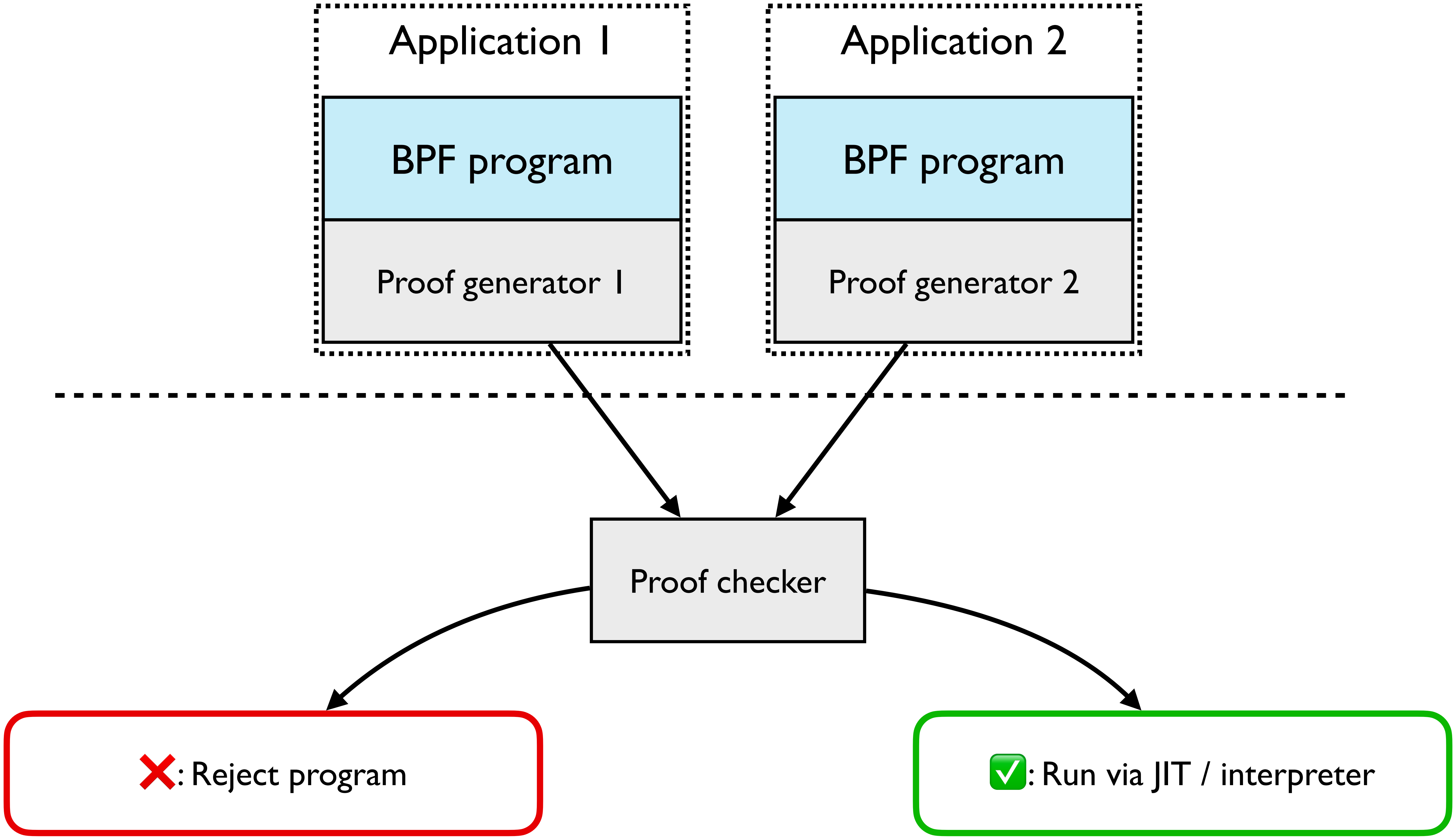
Correctness bugs: verifier accepts unsafe programs

Flexibility issues: rejects safe programs

How to build a verifier that avoids these issues?

- Minimize job of the kernel: only proof checking
- Untrusted analysis / proof generators in user space

ExoBPF overview



Logics

Need a logic in which to write specifications and proofs

Specification – the property that BPF programs should meet

Proof – formal argument that a BPF program meets specification

Example: prove “Socrates is mortal”

First-order logic

- Set of deduction rules for deriving true statements
- e.g.: →L rule says “If *A* → *B* and *A*, then *B*” is a valid deduction

Proof generator:

“All men are mortal. Socrates is a man. Therefore, Socrates is mortal”

$$\dfrac{\dfrac{\dfrac{}{P(x) \vdash P(x)}\,I \qquad \dfrac{}{Q(x) \vdash Q(x)}\,I}{P(x) \to Q(x), P(x) \vdash Q(x)}\,{\to}L}{\forall m.P(m) \to Q(m), P(x) \vdash Q(x)}\,\forall L$$

Proof checker: validates that a proof follows the deduction rules

Requirements on the logic for ExoBPF

Well-understood logic & proof-checking algorithm

Enable expressive specifications, e.g., memory safety

Enable different proof strategies

- Applications select best approach for their programs
- Examples: kernel verifier, SAT solving

Lean theorem prover

Rich logic: Used to formalize modern mathematics (mathlib)

Logic has been thoroughly-analyzed

Active community

Independent proof checkers (C++, Scala, Rust, Haskell)

Example: BPF safety specification

BPF safety: no division by zero, no OOB memory access, etc.

Formalize execution of BPF programs as a state machine

Each BPF instruction steps from one state to next

Safety definition: program execution cannot get stuck

Demo: BPF safety specification in Lean

ExoBPF: Specification + proof checker

Well-known algorithm for checking Lean proofs

Multiple, independent implementations of proof checkers

Is the proof checker simpler than the kernel BPF verifier?

- Uses a stable, well-documented algorithm
- Independent of BPF program or specific verifier strategies
- One checker written in Scala is 1,730 lines of code

Export format: “assembly code” for proofs and theorems

```
8082 #EA 1101 1
8083 #EL #BD 179 3 8082
8084 #EA 8081 8083
8085 #EA 8084 2
8086 #EA 8085 1
8087 #EC 603
8088 #EA 8087 19
8089 #EA 8086 8088
8090 #EL #BD 108 7899 8089
8091 #EL #BI 95 3 8090
8092 #EL #BI 4 3 8091
#DEF 602 8078 8092
8093 #EA 6912 1064
8094 #EA 8093 32
8095 #EP #BD 328 19 8094
8096 #EA 6912 7959
8097 #EP #BD 328 34 1064
8098 #EA 8096 8097
8099 #EP #BD 108 8095 8098
8100 #EP #BI 352 4 8099
8101 #EP #BI 26 28 8100
8102 #EP #BI 4 0 8101
```

Automating proof generation

Writing safety proofs for every BPF program is tedious

Approach: automate proof generation

- Write a BPF verifier in Lean (e.g., reimplement Linux verifier)
- Manually prove verifier is correct once for all programs
- Safety proof = verifier is correct + verifier accepts BPF program

Proof generator (1/2): Abstract interpretation

Inspired by kernel BPF verifier & CompCert's abstract interpreter

Compute bounds + tri-state numbers for each BPF register

Free of correctness bugs: mistakes will be caught by proof checker

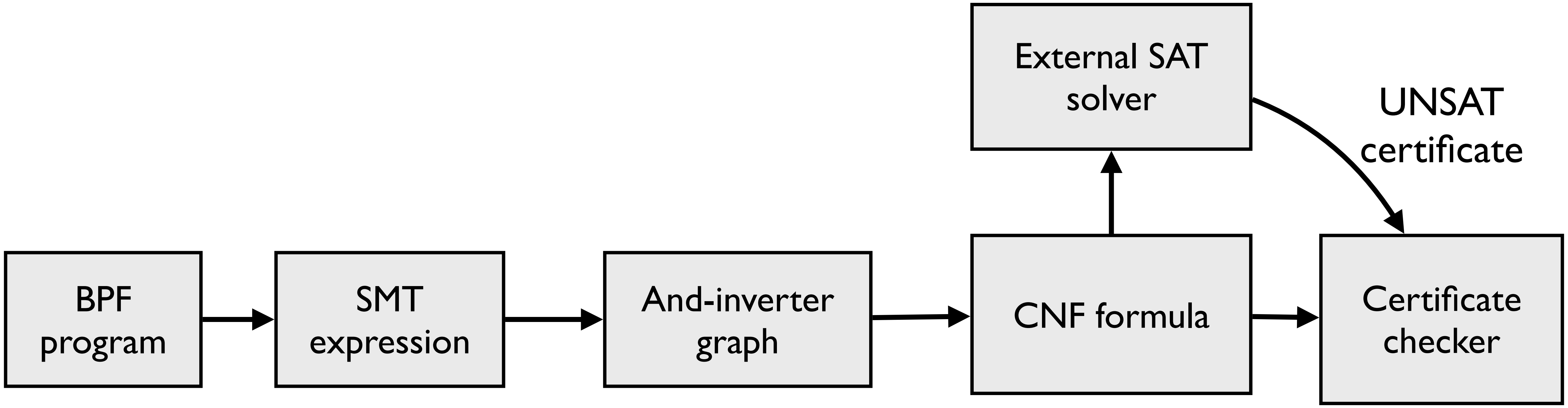
Can reject safe BPF programs

Proof generator (2/2): Symbolic execution + SAT

Compiles BPF program to a Boolean formula

- Use a SAT solver to prove validity of formula
- Embed certificate from SAT solver into program safety proof

More general, larger & slower-to-check proofs



Demo: proof generation & proof checking

Breaking down proof size & proof-checking time

Safety proof consists of two parts:

- General proof of verifier correctness
- Proof specific to a particular BPF program

Could improve proof size & proof-checking time by caching

	General proof of verifier correctness	Proof specific to example BPF program
Proof size	28MB	8kB
Proof-checking time (Using proof checker in Rust)	7.5s	1.3s

Barriers to integration with Linux

Performance on real-world programs requires more study

Embedding a proof checker into the Linux kernel

Implementing and maintaining proof generators in Lean

Conclusion

ExoBPF explores a different approach to building BPF verifiers

Would like to get feedback from kernel community

Preliminary prototype at https://github.com/uw-unsat/exoverifier