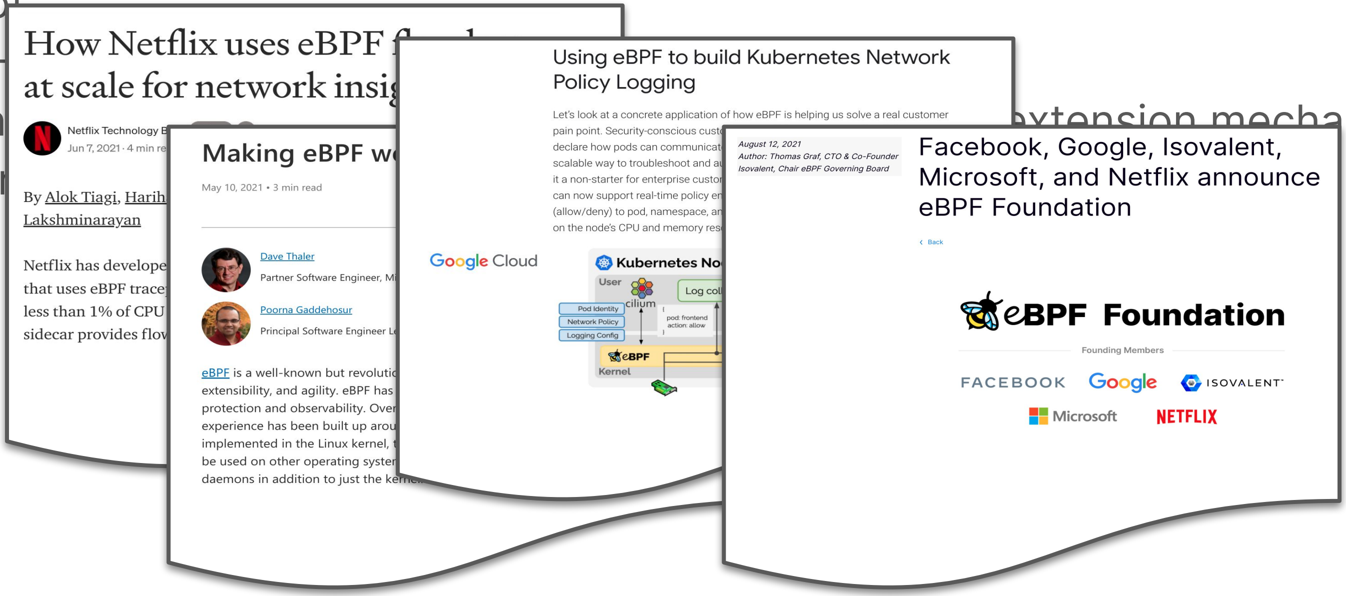
Verified Static Analyzers for Kernel Extensions

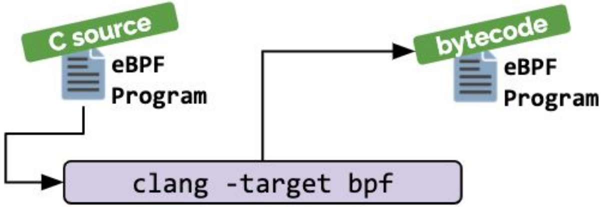
Harishankar Vishwanathan
Advisor: Srinivas Narayana
Co-advisor: Santosh Nagarakatte

Kernel extensions

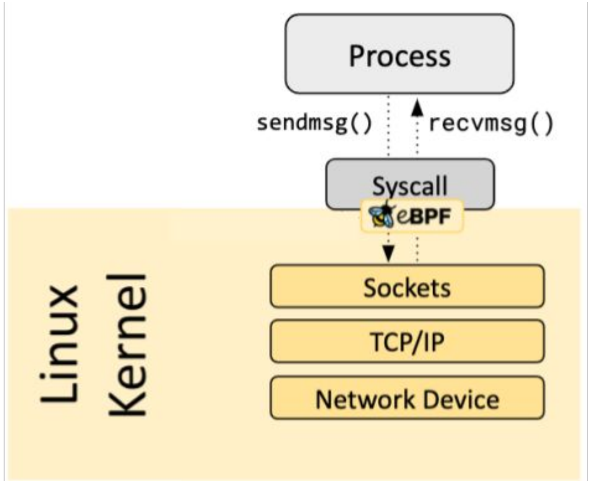
- Ability to extend the operating system kernel without having to recompile or reboot
  - H
- Exten … extension mechanism that is gai



eBPF (Extended Berkeley Packet Filter)

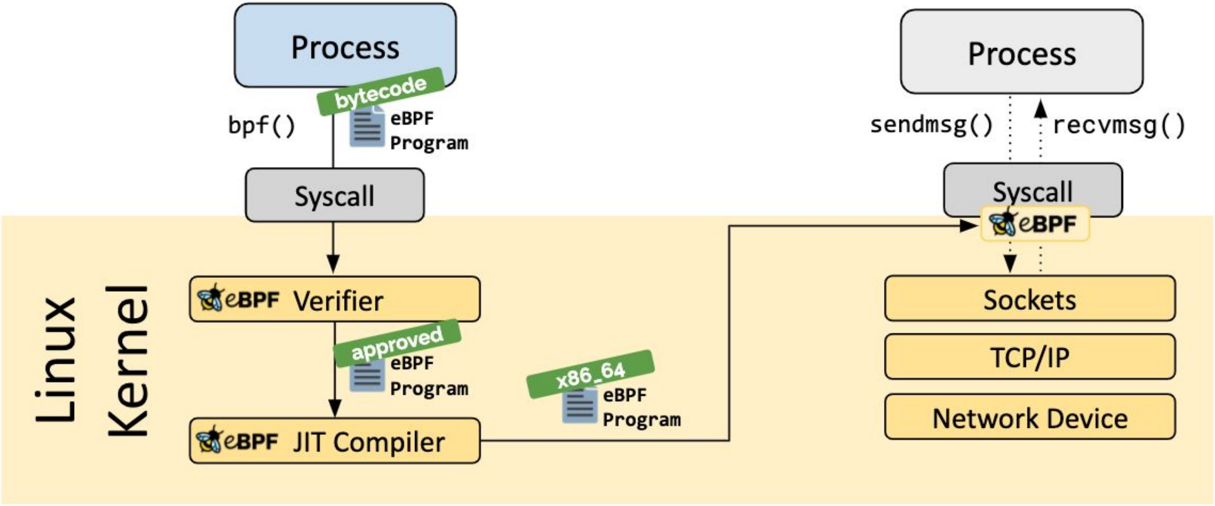


BPF_LD	Non-standard load operations
BPF_LDX	Load into register operations
BPF_ST	Store from immediate operations
BPF_STX	Store from register operations
BPF_ALU	32-bit arithmetic operations
BPF_JMP	64-bit jump operations
BPF_JMP32	32-bit jump operations
BPF_ALU64	64-bit arithmetic operations



eBPF Verifier

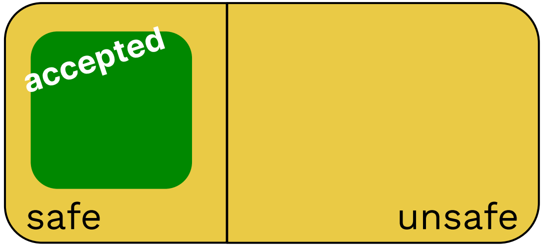
- Issue: running arbitrary user code in the kernel
- Solution: statically prove safety of the program
- Safety checks
  - Termination
  - Illegal operations
  - Memory access



Verification Must be Sound and Precise and Fast

- **Soundness** : Unsafe programs should be rejected
- **Precision** : Safe programs shouldn't be rejected
- **Speed**: Minimal load times + Prompt feedback on rejection

Can we formally verify the soundness and precision of the static analysis in the eBPF verifier?



RUTGERS

Static Analyses in the eBPF verifier

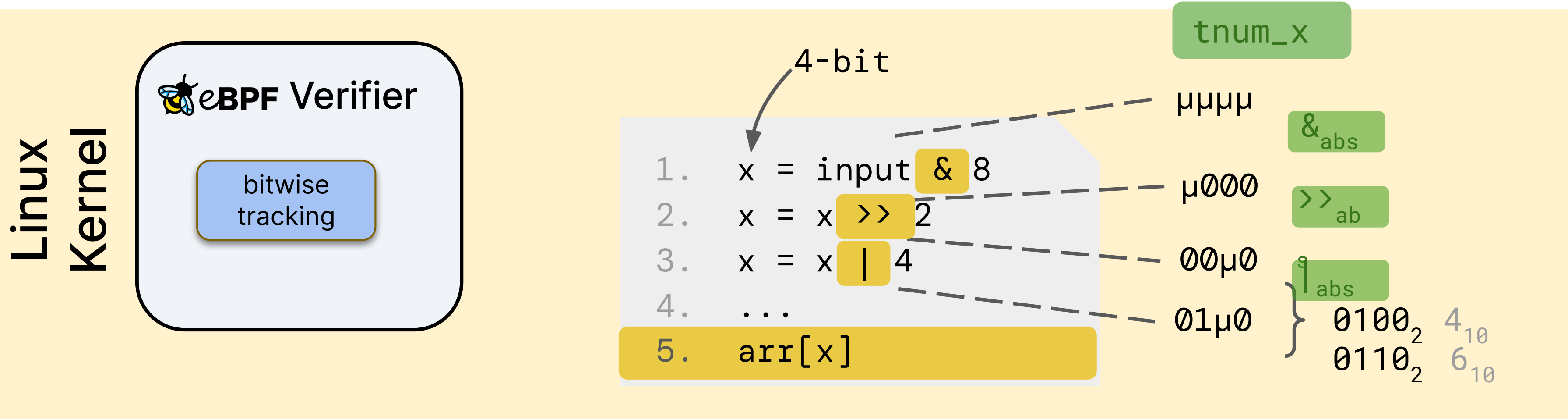
- Tnums [CGO '22]: Reasoning about the soundness and precision of bitwise tracking
- Agni [CAV '23]: Reasoning about the soundness and precision of the range analysis + bitwise tracking + their combination

Tnums

Proving the soundness and precision of bitwise tracking

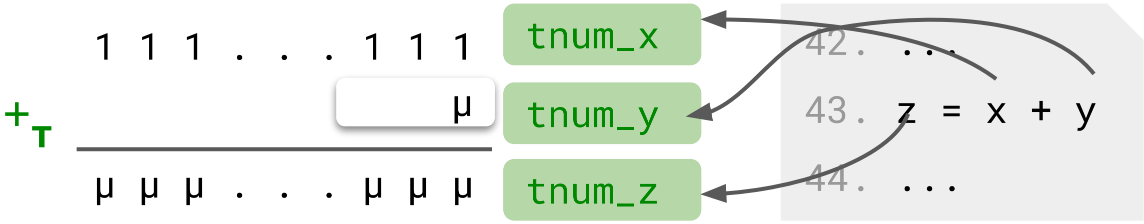
Static Analysis in the eBPF Verifier

- Sub-task: track the **values** of program variables across ***all*** executions
  - Using abstract values from an abstract domain – ***Abstract Interpretation***
- Bitwise domain: track individual bits of a program variable .
  - Kernel term: tristate numbers (tnums) `{0, 1, μ}`



Challenges

- Developing abstract operators is not trivial




```
def tnum_add(tnum P, tnum Q):
   u6
   u6
   u6
   u6
   u6                                   sk
   return tnum(value=sv & ~mu,mask=mu)
```

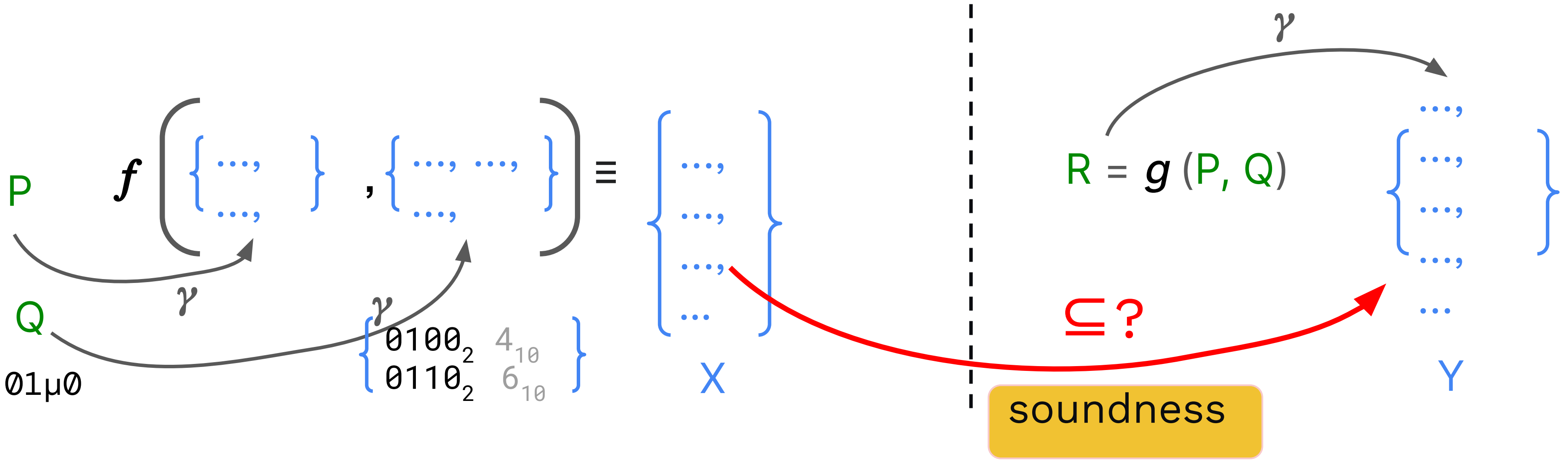
Is tnum_add sound for all tnums P & Q?

Is tnum_add precise?

Soundness of Abstract Operators

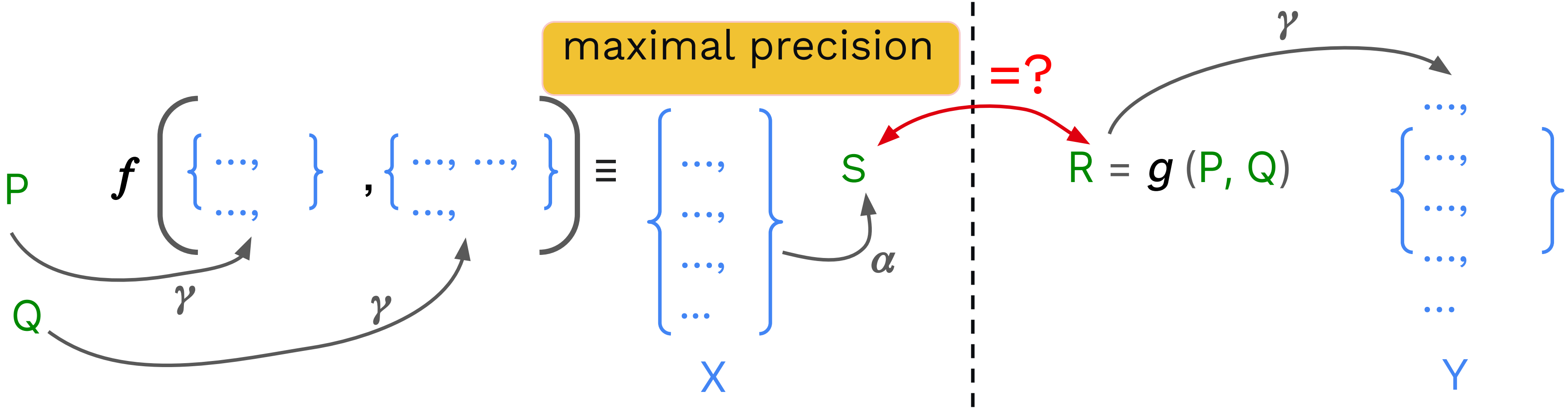
- BPF instruction set:

  `add, sub, mul, div, or, and, lsh, rsh, neg, mod, xor, arsh`
- Concrete operator (eg. integer addition) $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
- Abstract operator (eg. tnum addition) $g : \mathbb{A}_{\text{tnum}} \times \mathbb{A}_{\text{tnum}} \rightarrow \mathbb{A}_{\text{tnum}}$



Maximal Precision of Abstract Operators

- BPF instruction set:
  `add, sub, mul, div, or, and, lsh, rsh, neg, mod, xor, arsh`
- Concrete operator (eg. integer addition) $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
- Abstract operator (eg. tnum addition) $g : \mathbb{A}_{\text{tnum}} \times \mathbb{A}_{\text{tnum}} \to \mathbb{A}_{\text{tnum}}$



SMT Verification (A quick aside)

- The Boolean SATisfiability problem
  - Boolean variables A, B, C
  - Boolean expressions $\psi_1 = (A \lor \neg B) \land C$
    $\psi_2 = (\neg A \lor B) \land (A \lor C) \land (B \lor \neg C)$
  - Ask a SAT Solver: is the set of constraints satisfiable? $\{\psi_1, \psi_2\}$

- Outcomes:
  - SAT
    - `+ model [A = true, B = true, C = true]`
  - UNSAT
  - UNKNOWN

RUTGERS

SATisfiability Modulo Theories

- Satisfiability taking into account theories
- Binary variables are replaced by predicates over a set of non-binary variables
  - e.g A ⇒ 3x + 2y -z >= 4 (linear inequalities)
- Predicates are classified according to theories used
  - Theory defines rules on the (non-binary) variables: operations, and how to combine them
    - if x, y, z are real numbers, we use the theory of linear real arithmetic
- Generally, program analysis relies on the theory of bitvectors

Building A Soundness Specification in First Order Logic

- Membership predicate

$$member_{\texttt{tnum}}(x, P)$$

- Semantics of concrete operator f

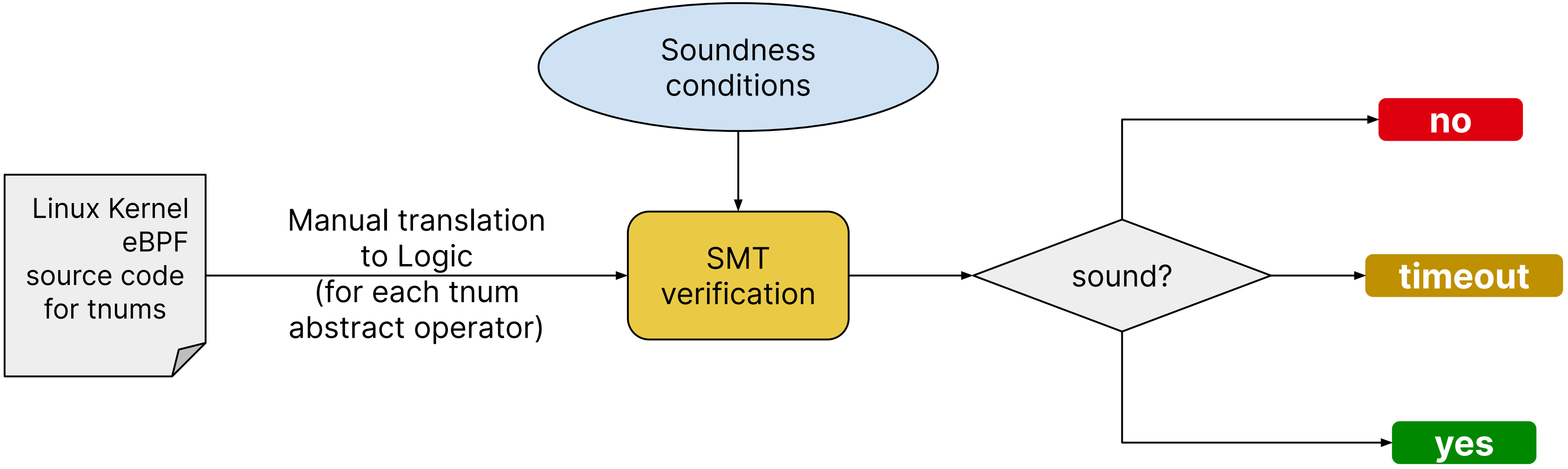
$$z = f(x, y)$$

- Semantics of abstract operator g
  - Manually translate from C to SMT

$$R = g(P, Q)$$

Soundness Specification in First Order Logic

Overview (Verification of Tnum Abstract Operators)



Results from Automated Verification

- Proved the soundness of the following kernel's tnum abstract operators: `add`, `sub`, `mul`, `div`, `or`, `and`, `lsh`, `rsh`, `neg`, `mod`, `xor`, `arsh`
- What about `mul`, `div`, `mod`?
  - `div`, `mod`: The kernel implements these abstract operators by setting the result to completely unknown: μμμμ…μ
  - These operators are trivially sound
  - `mul`?

RUTGERS

Existing Tnum multiplication

- Implementation has a loop, when unrolled leads to a large formula
- Performs multiplication on integers which is expensive to solve when encoded in bitvector theory.
- Verification times out

```
def hma(tnum ACC, u64 x, u64 y):
  for y in range(0...64):
    if (LSB of y is 1):
      ACC := ACC +_T tnum(value=0,
                           mask=x)
    y := y >> 1
    x := x << 1
  return ACC

def tnum_mul(tnum P, tnum Q):
  tnum π := tnum(P.v * Q.v, 0)
  tnum ACC := hma(π, P.m, Q.m|Q.v)
  tnum R := hma(ACC, Q.m, P.v)
  return R
```

Existing algorithm

RUTGERS

A new algorithm for tnum multiplication

- Faster and more precise than the existing implementation
- Analytical proof of soundness

sound

```
def our_mul(tnum P, tnum Q):
    ACCv := tnum(0, 0)
    ...
    ...th):
    ... (P.m[0] == 0):
    ...(ACCv, tnum(Q.v, 0))
    ...(ACCm, tnum(0, Q.m))
    ...):
    ...(ACCm,
    ...m(0, Q.v|Q.m))
    ...1)
    ...1)
    tnum R := tnum_add(ACCv, ACCm)
    return R
```

bpf.vger.kernel.org archive mirror

search help / color / mirror / Atom feed

* **[PATCH bpf-next] bpf: tnums: Provably sound, faster, and more precise algorithm for tnum_mul**
@ 2021-05-28 3:55 hv90
2021-05-30 5:59 ` Andrii Nakryiko
0 siblings, 1 reply; 5+ messages in thread

From: hv90 @ 2021-05-28 3:55 UTC (permalink / raw)
To: ast
Cc: bpf, Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, Santosh Nagarakatte

From: Harishankar Vishwanathan <harishankar.vishwanathan@rutgers.edu>

This patch introduces a new algorithm for multiplication of tristate numbers (tnums) that is provably sound. It is faster and more precise when compared to the existing method.

Our new algorithm

RUTGERS

Analytical proofs of maximal precision for addition and subtraction

maximally precise

```
def tnum_add(tnum P, tnum Q):
  u64 sm = P.mask + Q.mask
  u64 sv = P.value + Q.value
  u64 sigma = sm + sv
  u64 chi = sigma ^ sv
  u64 mu = chi | a.mask | b.mask
  return tnum(value=sv & ~mu,mask=mu)
```

Existing algorithm for tnum addition

maximally precise

```
def tnum_sub(tnum P, tnum Q):
  u64 dv = P.value - Q.value
  u64 alpha = dv + P.mask
  u64 beta = dv - Q.mask
  u64 chi = alpha ^ beta
  u64 mu = chi | P.mask | Q.mask
  return tnum(value=dv & ~mu, mask=mu)
```

Existing algorithm for tnum subtraction

RUTGERS

Summary of Contributions to the Domain of Tristate Numbers

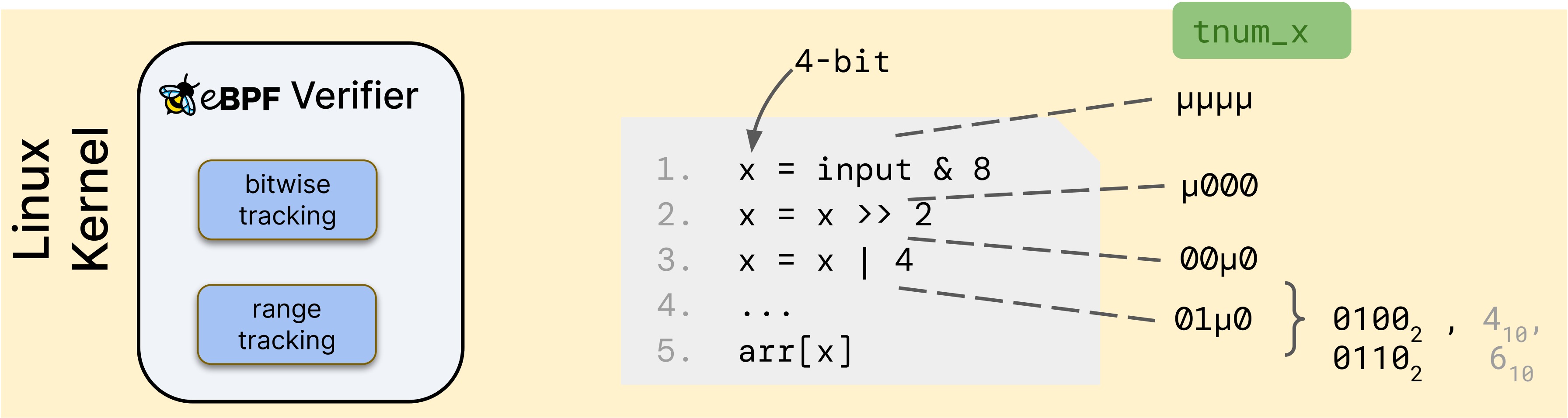
- Proved the soundness of all the kernel existing algorithms for tristate numbers
- A faster, more precise version of tnum multiplication
- Analytical proof of soundness
- Analytical proof that tnum addition and subtraction are maximally precise

Agni

Proving the soundness of the entire value tracking analysis

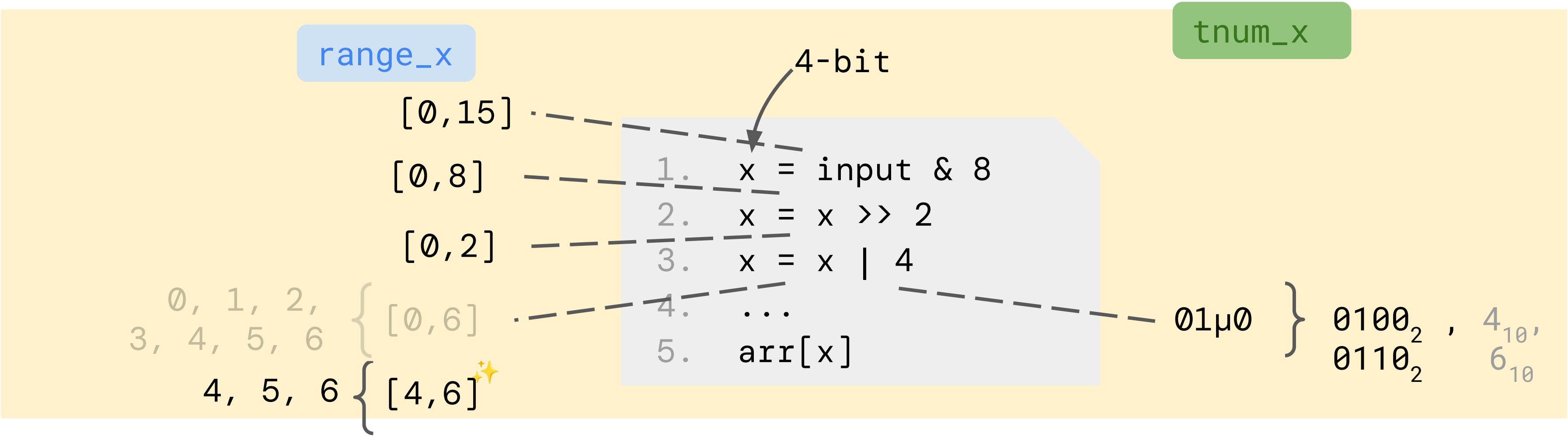
RUTGERS

Range Analysis



- Range Analysis: tracks range of possible values – [min, max]
  - Interval domain

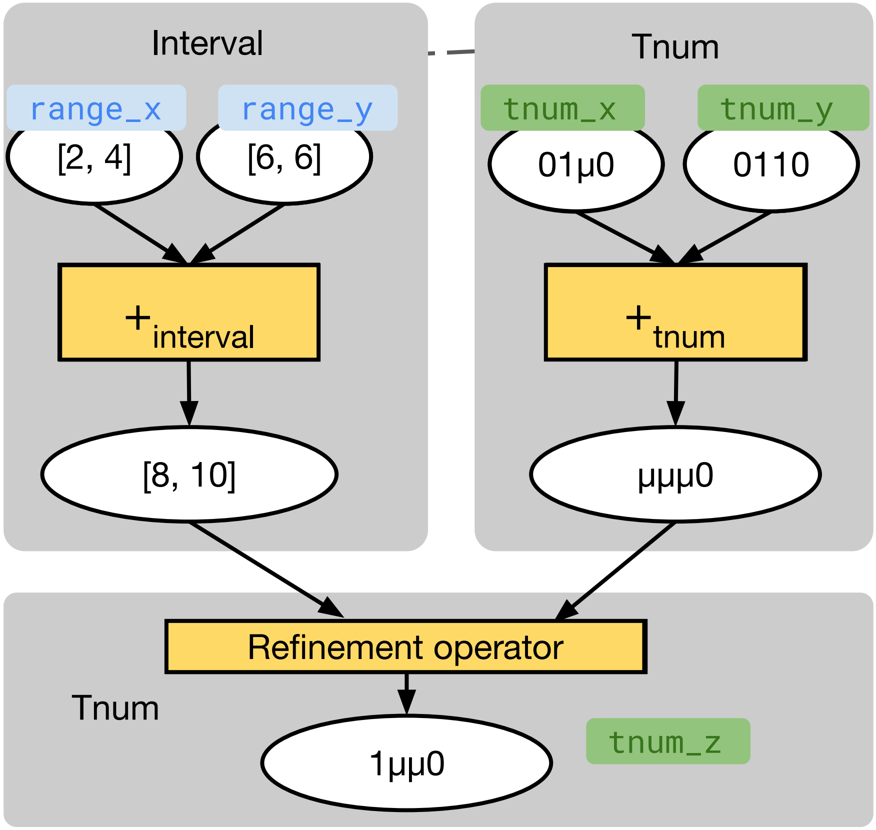
Range Analysis: Refinement



- Range Analysis: tracks range of possible values – [min, max]
    - Interval domain
- Refinement: Abstract values in one domain can be used to refine abstract values in another domain

Typical Refinement in Abstract Interpretation

```
41.   ...
42.   z = x + y
43.   ...
```

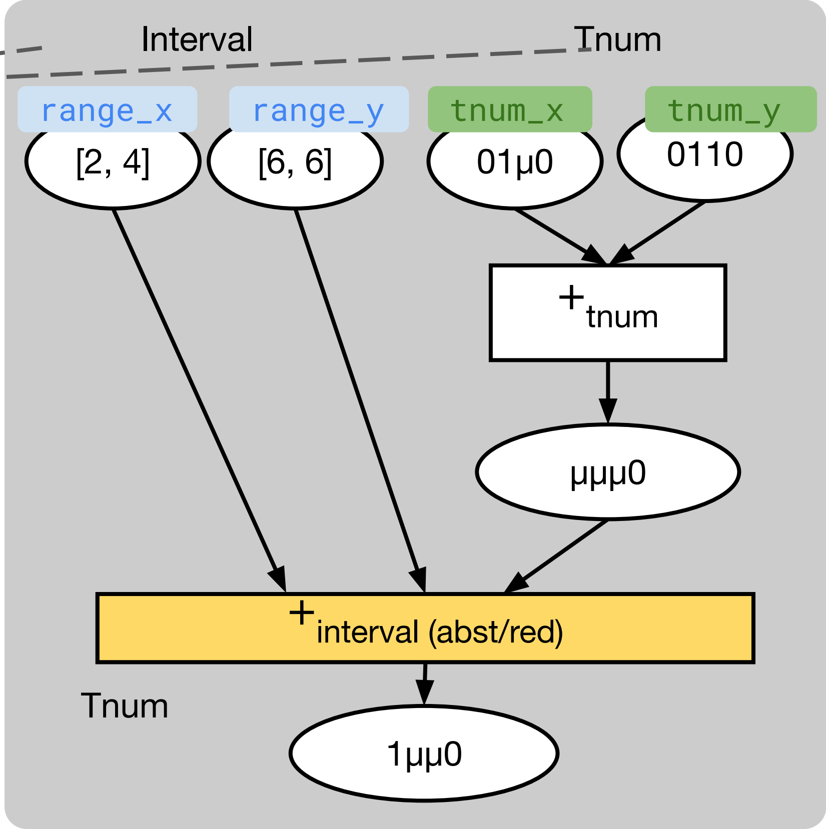


Soundness?

Modular reasoning

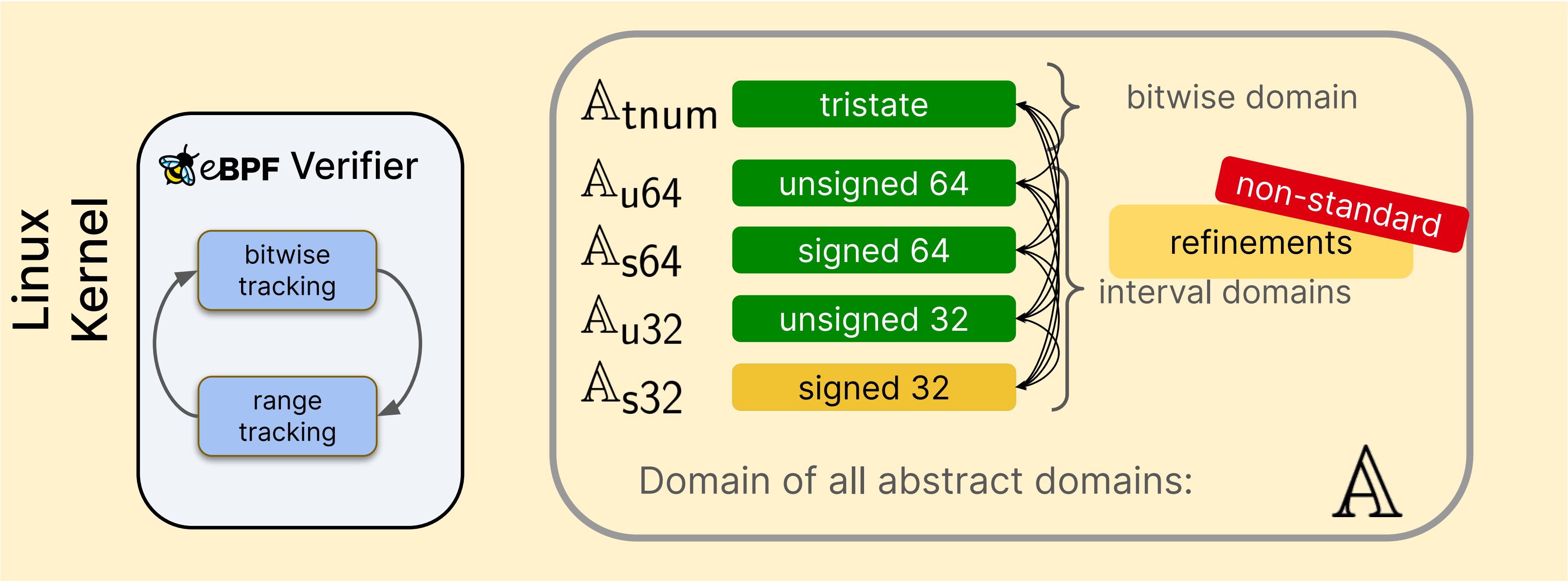
The Linux Kernel's Non-modular Refinement

```
41.   ...
42.   z = x + y
43.   ...
```



"One-shot" reasoning

Value Tracking Abstract Domains in the Linux Kernel



$$\mathbb{A} \triangleq \mathbb{A}_{tnum} \times \mathbb{A}_{u64} \times \mathbb{A}_{s64} \times \mathbb{A}_{u32} \times \mathbb{A}_{s32}$$

Soundness Specification in First Order Logic for Multiple Domains

$$\forall P, Q \in \mathbb{A} : \ {}_{n} :$$

$$\forall x, y \in \mathbb{Z} :$$

$$member(x, P) \wedge member(y, Q) \wedge \ ) \wedge$$

$$z = f(x, y) \ \wedge R = g(P, Q) \wedge$$

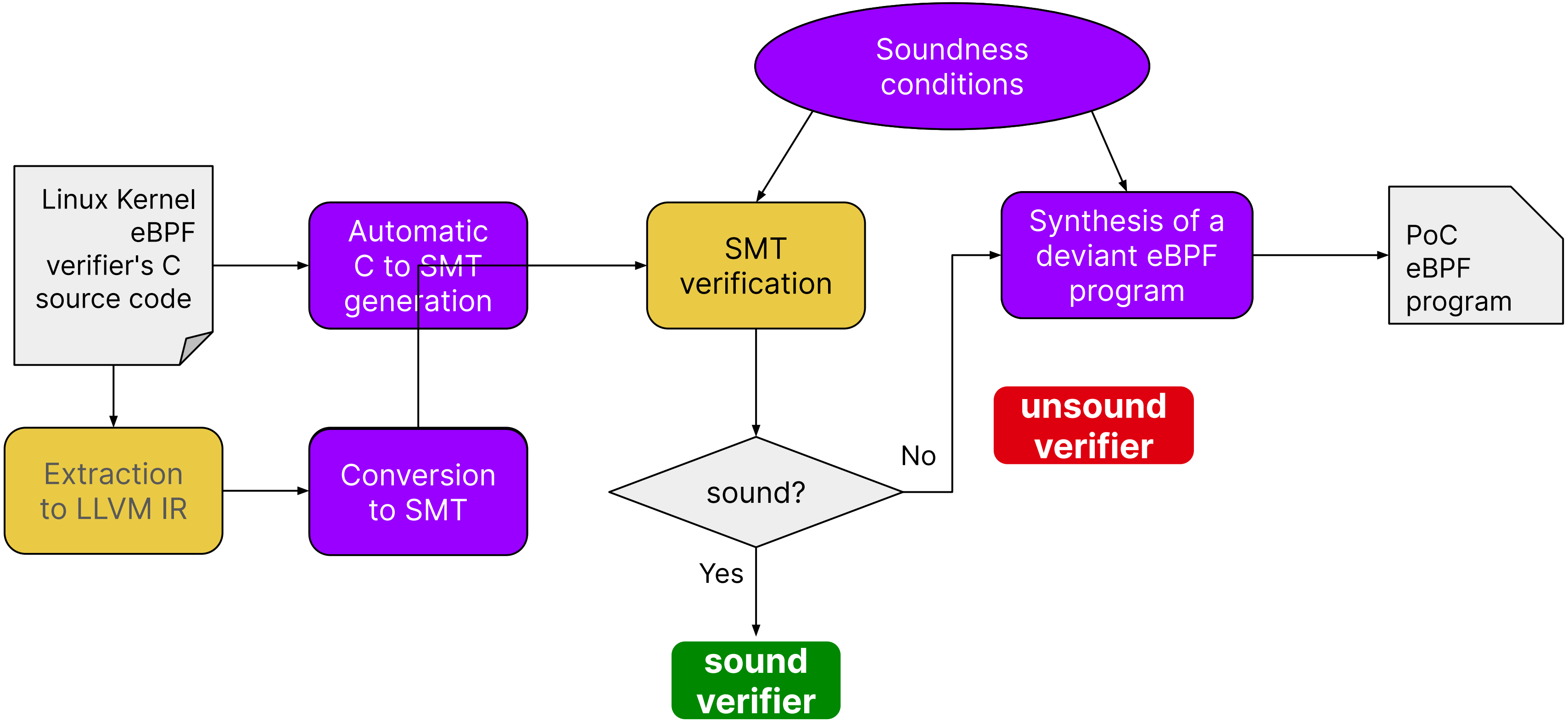
$$\implies$$

$$member(z, R)$$

$$member_{tnum}(z, R)$$

- Tedious and error-prone to write down manually
- Changing across kernel versions - which one to write and verify?
- 🤖 Automate 🤖

Agni: Overview



LLVM To SMT

```c
int foo(int a, int b) {
    int retval;
    if (a <= b)
        retval = b - 10;
    else
        retval = a + 10;
    return retval;
}
```

```llvm
define i32 @max(i32 %a, i32 %b)

entry:
    %x0 = icmp sgt i32 %a, %b
    br i1 %x0, label %btrue, label %bfalse
```

```llvm
btrue:
    %x1 = add i32 %a, 10
    br label %end
```

```llvm
bfalse:
    %x2 = sub i32 %b, 10
    br label %end
```

```llvm
end:
    %retval = phi i32 [%x1, %btrue], [%x2, %bfalse]
    ret i32 %retval
```

LLVM To SMT: Aggregating Basic Blocks' Logic

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const x0 Bool)
(declare-const x1 (_ BitVec 32))
(declare-const x2 (_ BitVec 32))
```

bfalse

```
(= x2 (bvsub b 10)
```

```
define i32 @max(i32 %a, i32 %b)
```

```
entry:
    %x0 = icmp sgt i32 %a, %b
    br i1 %x0, label %btrue, label %bfalse
```

```
btrue:
    %x1 = add i32 %a, 10
    br label %end
```

```
bfalse:
    %x2 = sub i32 %b, 10
    br label %end
```

```
end:
    %retval = phi i32 [%x1, %btrue], [%x2, %bfalse]
    ret i32 %retval
```

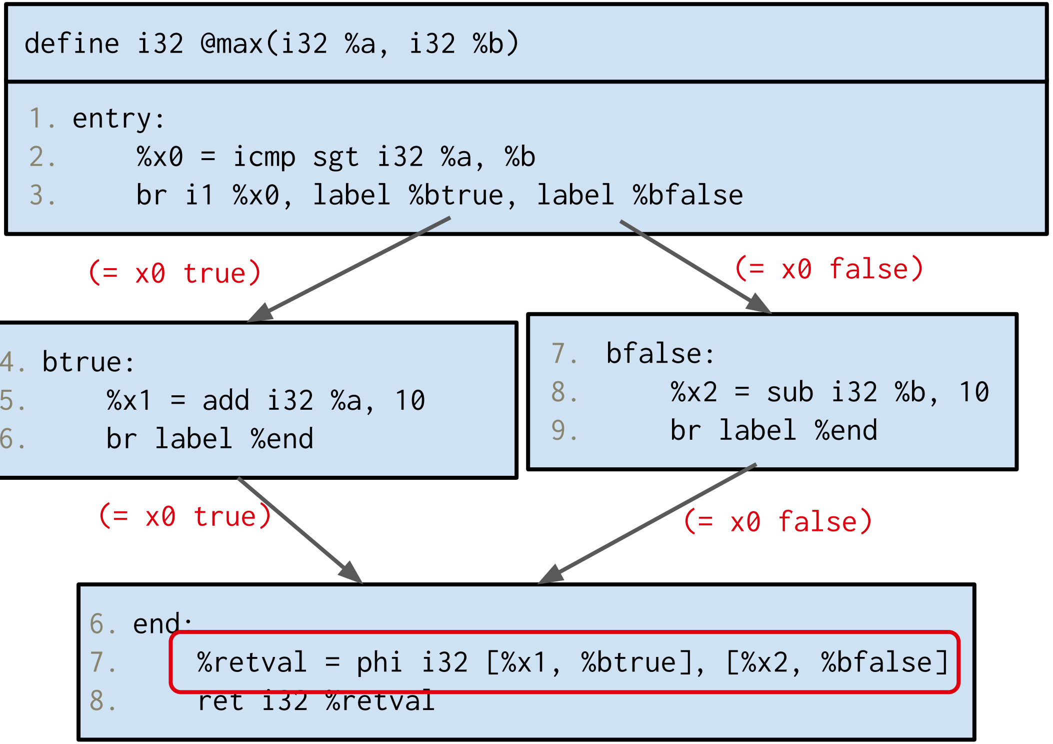
Handling LLVM code: Resolving Phi nodes

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(declare-const x0 Bool)
(declare-const x1 (_ BitVec 32))
(declare-const x2 (_ BitVec 32))
(declare-const retval (_ BitVec 32))
```

end

```
(=> (= x0 true) (= retval x1))

(=> (= x0 false) (= retval x2))
```



Handling LLVM code: Path Conditions

```
define i32 @max(i32 %a, i32 %b)
```

(true)

```
entry:
    %x0 = icmp sgt i32 %a, %b
    br i1 %x0, label %btrue, label %bfalse
```

(= x0 true)

```
btrue:
    %x1 = add i32 %a, 10
    br label %end
```

(= x0 false)

```
bfalse:
    %x2 = sub i32 %b, 10
    br label %end
```

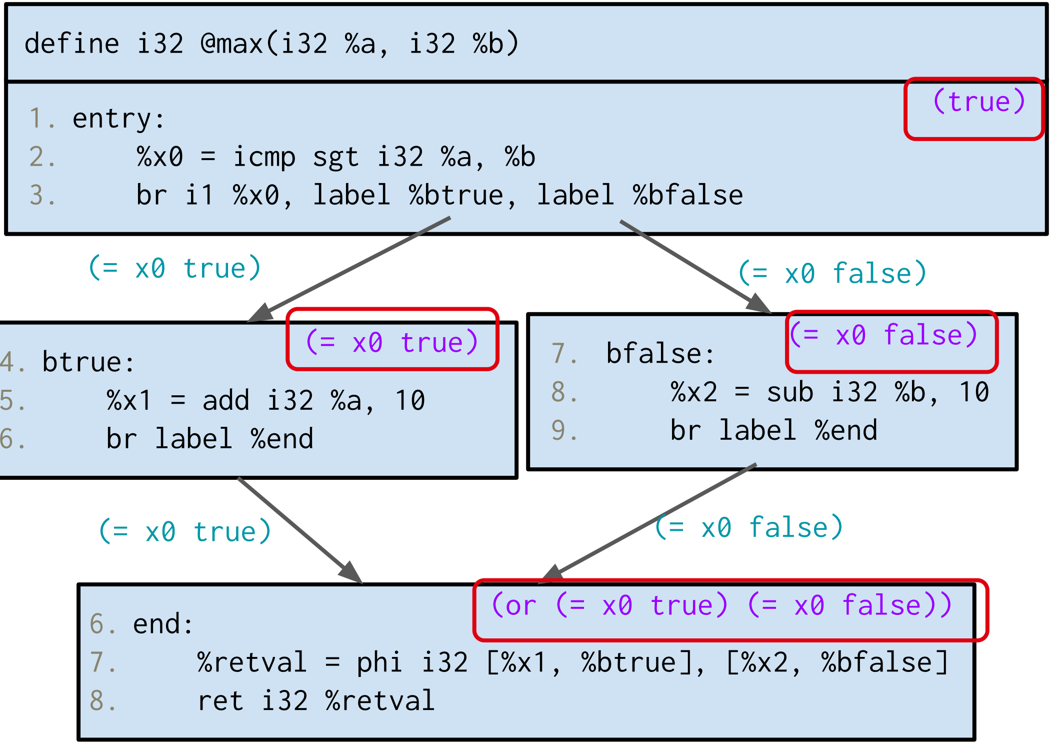
(or (= x0 true) (= x0 false))

```
end:
    %retval = phi i32 [%x1, %btrue], [%x2, %bfalse]
    ret i32 %retval
```

Handling LLVM code: Putting it all together

```
(=> true entry)
(=> (= x0 true) btrue)
(=> (= x0 false) bfalse)
(=> (or (= x0 true) (= x0 false)) end)
```

```
(assert
 (=> (= x0 false) (= retval x2))
)
```



Handling LLVM code: Putting it all together

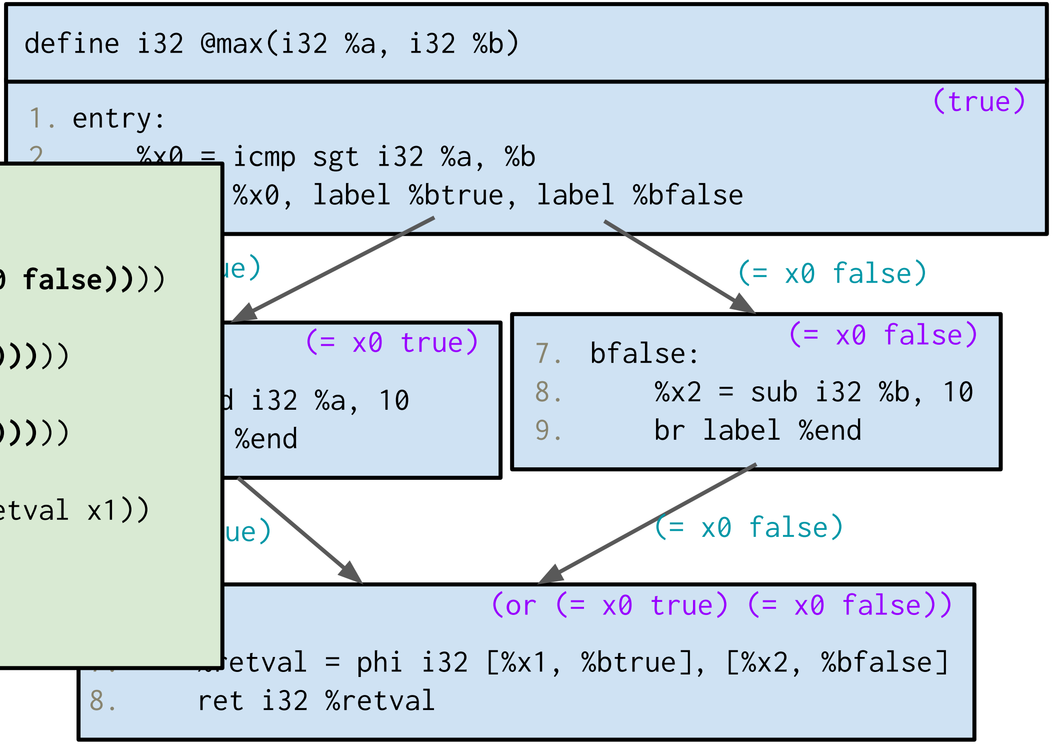
```
(assert (=> true (ite (bvsgt a b) (= x0 true) (= x0 false))))

(assert (=> (= x0 true)  (= x1 (bvadd a (_ bv10 32)))))

(assert (=> (= x0 false) (= x2 (bvsub b (_ bv10 32)))))

(assert (=> (=> (or (= x0 true) (= x0 false)) (= retval x1))
        (and (=> (= x0 true) (= retval x1))
             (=> (= x0 false) (= retval x2)))
))
```

```
(assert
 (=> (= x0 false) (= retval x2))
)
```



Handling Real-World Kernel Code

- Converting C to LLVM IR is not straightforward
  - Custom passes to eliminate dead code, and inline function calls, making it conducive to work with
- Generated IR is much larger than our toy example
  - ~600 llvm code, ~50 basic basic block (per eBPF abstract operator)
- Memory access instructions: structs and pointers, loads and stores to memory.
  - Leverage LLVM's MemorySSA analysis
    - Stores and branch merges are annotated with new versions of memory
    - Loads are annotated with existing versions of memory that they load from
- Testing harness
  - Unit testing SMT translations

Results   But Can We do More?

- Automatically test kernels 4.14 through 5.19 for soundness
- Proved that all abstract operators in kernels v5.13 to v 5.19 are sound
- What can we do about unsound versions?

Generate *actual* eBPF programs!

- Generated eBPF program that manifests the bugs in 97% of the cases

Kernel Version	Sound?
v4.14	❌
v5.5	❌
v5.7	❌
...	❌
v5.12	❌
v5.13	✅
v5.14	✅
v5.15	✅
...	✅

RUTGERS

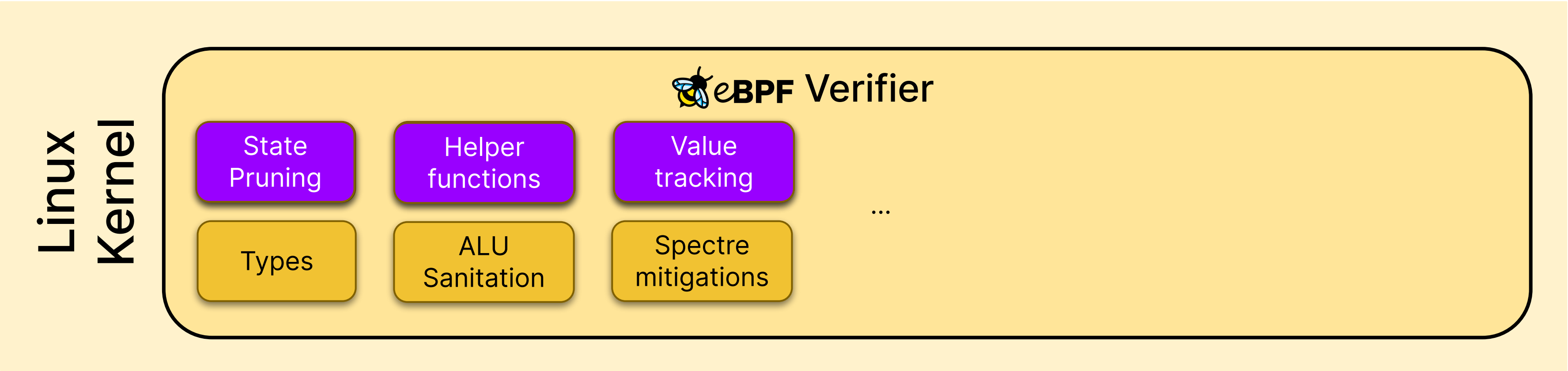
Future

Next Steps

- Agni
  - Pushing Agni to Linux's Continuous Integration
  - Reducing Verification Time
    - Using environments like Rosette with tooling for finding verification bottlenecks
    - Trying other bitvector solvers (Bitwuzla)
  - Completeness of Synthesis
  - Exploring techniques to reduce our TCB by doing conversion to SMT in Coq.

Extending Current Work

- Hardening other parts of the verifier



Long Term Vision

- Fortifying eBPF verification with formal foundations
- Exploring techniques to build a verifier that is correct-by-construction
  - Domain specific requirements: speed, low resource consumption

Questions?