A Flow-Sensitive Refinement Type System for Verifying eBPF Programs

CERTORA

Ameer Hamza[1], **Lucas Zavalía**[1], Arie Gurfinkel[2], Jorge A. Navas[3], Grigory Fedyukovich[1]

[1]Florida State University
[2]University of Waterloo
[3]Certora, Inc.

What is eBPF?

What is eBPF?

- System for hotloading code into kernel space

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for
  - Packet filtering

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for
  - Packet filtering
  - Process monitoring

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for
  - Packet filtering
  - Process monitoring
  - Security auditing

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for
  - Packet filtering
  - Process monitoring
  - Security auditing
  - etc.

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for
  - Packet filtering
  - Process monitoring
  - Security auditing
  - etc.
- Used throughout industry

What is eBPF?

- System for hotloading code into kernel space
- Commonly Used for
  - Packet filtering
  - Process monitoring
  - Security auditing
  - etc.
- Used throughout industry

ANDROID

CLOUDFLARE

What is eBPF?

- Hotloading kernel extensions is unsafe in general

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”

What is eBPF?

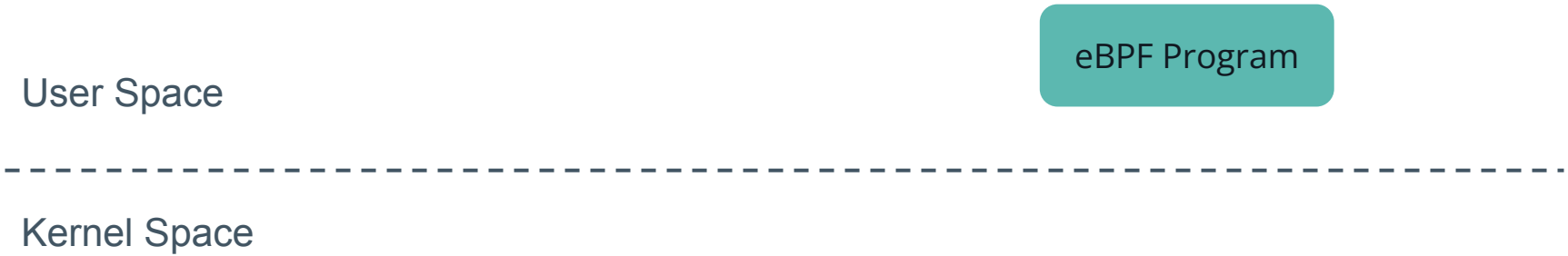
- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted



What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted

What is eBPF?

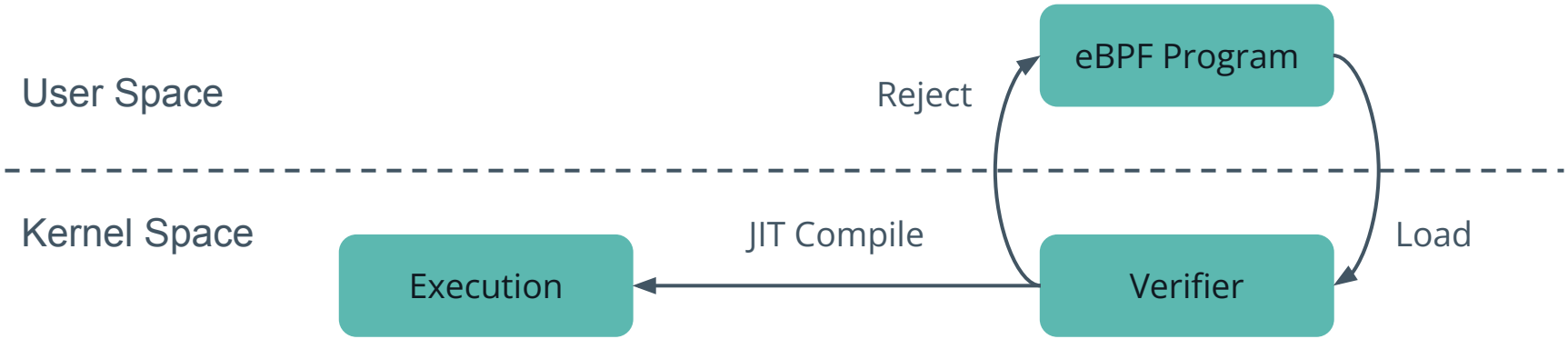
- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted

What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted

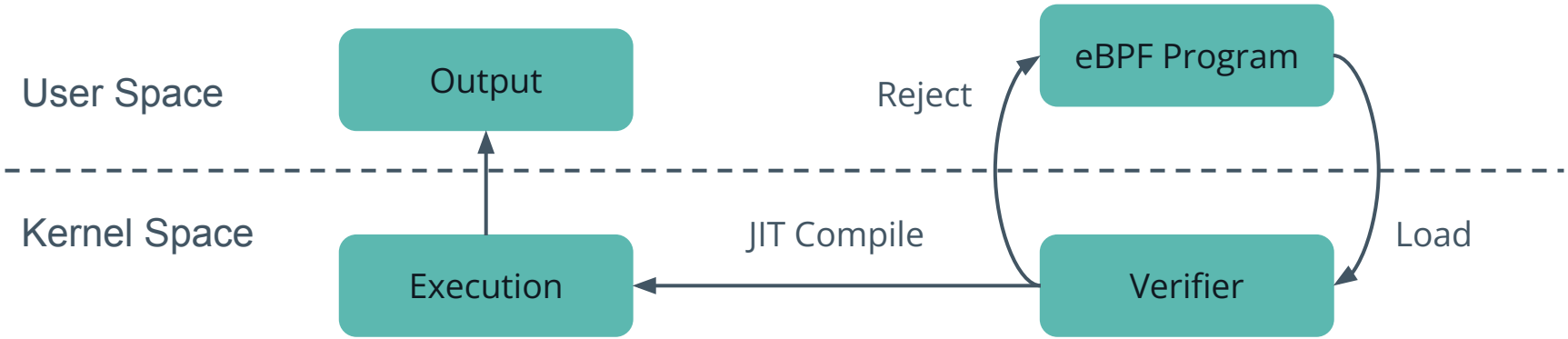
What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted



What is eBPF?

- Hotloading kernel extensions is unsafe in general
  - May reveal secrets
  - May crash the whole OS
  - etc.
- All eBPF programs must be “formally verified”
- eBPF programs are restricted



eBPF Safety

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

Context is passed to eBPF program from OS

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

Load pointer to beginning of packet

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

Load end of packet

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data + 0x3A53E170);
5       return 0;
6   }
```

Exit condition

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

Load/Store 4 bytes

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

UNSAFE 🚫

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

UNSAFE 🚫

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data + sizeof(int) > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

eBPF Safety

- **Information Flow Safety:** An eBPF program never exposes the address space of the kernel
- **Memory Safety:** An eBPF program accesses only certain memory regions

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data + sizeof(int) > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

SAFE ✓

Verifier Log

Verifier Log

```
1
2  Pre-invariant : [
3      meta_offset=[-4098, 0],
4      packet_size=[0, 65534],
5      r1.ctx_offset=0, r1.svalue=[1, 2147418112], r1.type=ctx,
6      r10.stack_offset=512, r10.svalue=[512, 2147418112], r10.type=stack]
7  Stack: Numbers -> {}
8  entry:
9    r0 = 1;
10   assert r1.type in {ctx, stack, packet, shared};
11   assert valid_access(r1.offset, width=4) for read;
12   r2 = *(u32 *)(r1 + 0);
13   assert r1.type in {ctx, stack, packet, shared};
14   assert valid_access(r1.offset+4, width=4) for read;
15   r1 = *(u32 *)(r1 + 4);
16   assert valid_access(r2.offset) for comparison/subtraction;
17   assert valid_access(r1.offset) for comparison/subtraction;
18   assert r2.type in {number, ctx, stack, packet, shared};
19   assert r2.type == r1.type in {ctx, stack, packet};
20   goto 3:4,3:8;
```

Verifier Log

```
126 3:4: Upper bound must be at most packet_size (valid_access(r2.offset, width=4) for read)
127 3:4: Upper bound must be at most packet_size (valid_access(r2.offset, width=4) for write)
128 3:8: Code is unreachable after 3:8
129
130 0,0.000811,5504
```

Existing Verifiers

Existing Verifiers

- **Linux Verifier:**

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier

- **PREVAIL [PLDI, 2019]:**

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier

- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier

- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation
  - Part of eBPF for Windows

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier
- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation
  - Part of eBPF for Windows

- **Different Verifier Same Problems:**

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier
- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation
  - Part of eBPF for Windows

- **Different Verifier Same Problems:**
  - Verifier is monolithic

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier

- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation
  - Part of eBPF for Windows

- **Different Verifier Same Problems:**
  - Verifier is monolithic
  - Lives entirely in kernel space

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier
- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation
  - Part of eBPF for Windows

- **Different Verifier Same Problems:**
  - Verifier is monolithic
  - Lives entirely in kernel space
  - Users cannot provide guidance

Existing Verifiers

- **Linux Verifier:**
  - Uses symbolic execution
  - First eBPF verifier
- **PREVAIL [PLDI, 2019]:**
  - Uses abstract interpretation
  - Part of eBPF for Windows

- **Different Verifier Same Problems:**
  - Verifier is monolithic
  - Lives entirely in kernel space
  - Users cannot provide guidance
  - Verifier logs are hard to read

Type Based Verification

- Type systems addresses architectural problems from existing verifiers

Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space

Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space

Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers

Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers

Type Based Verification

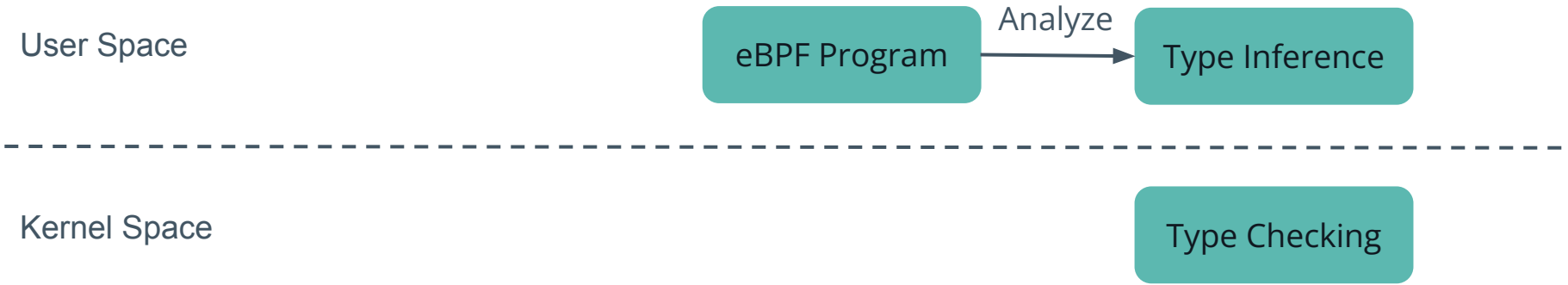
- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers

Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers

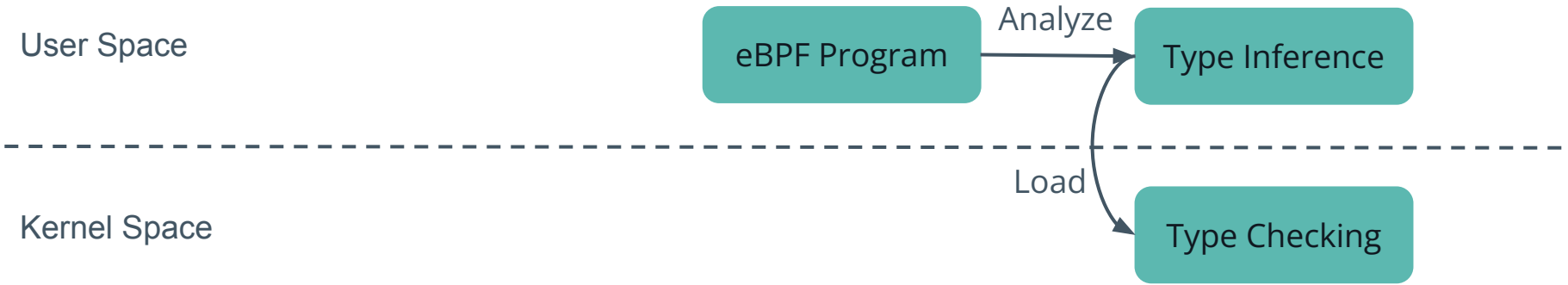
Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers



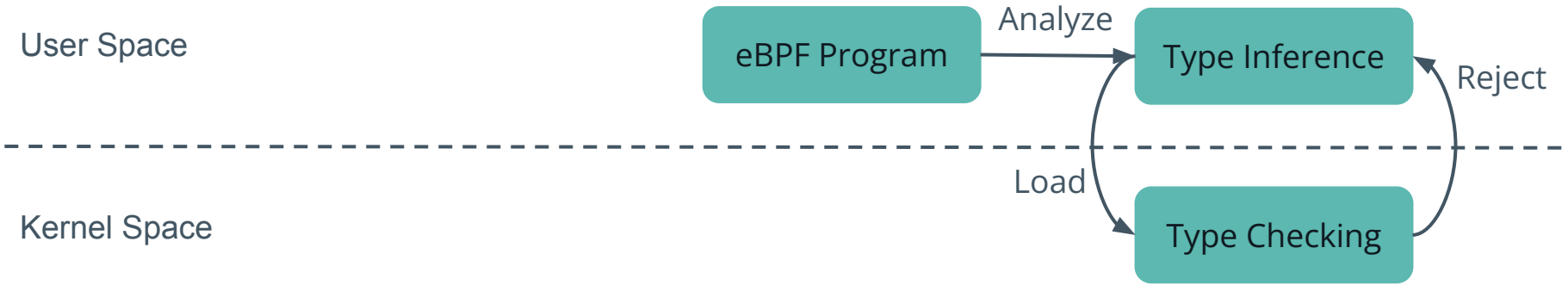
Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers



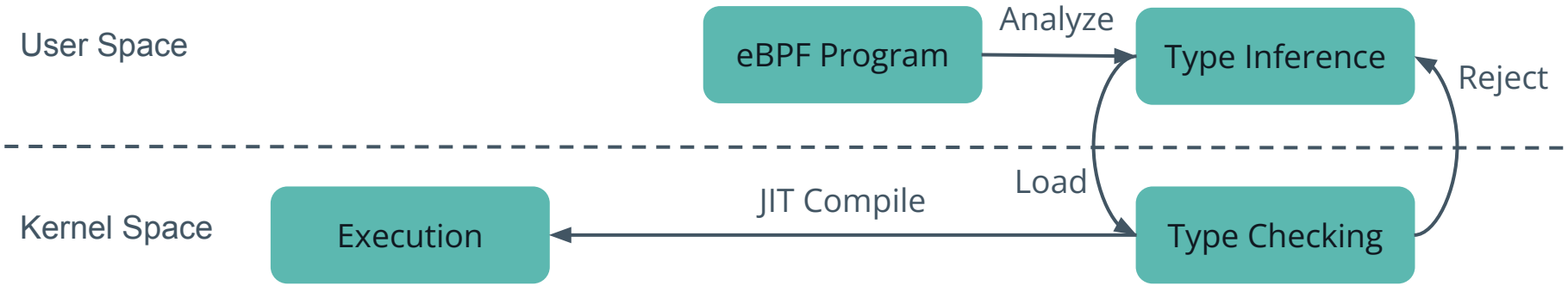
Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers



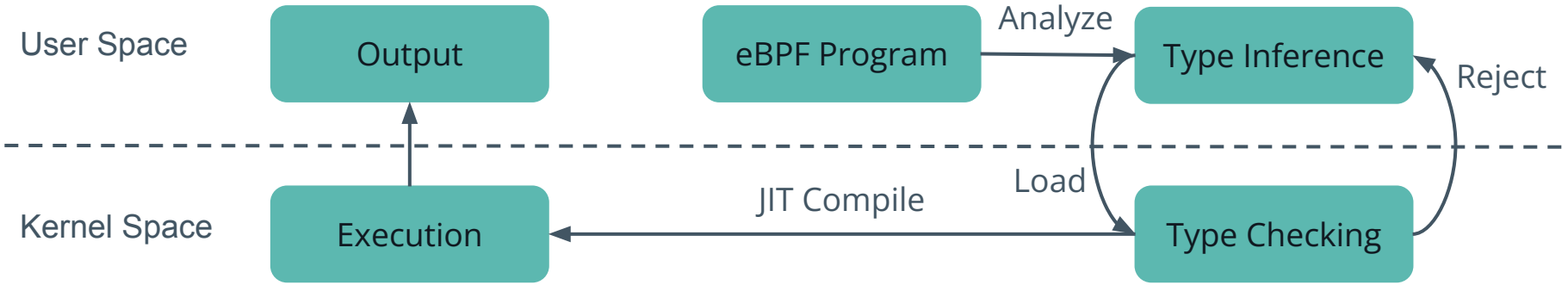
Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers



Type Based Verification

- Type systems addresses architectural problems from existing verifiers
  - *Type inference* done in user space
  - *Type checking* done in kernel space
- Types are relatively more intuitive for programmers



Basic Types

Three Basic Kinds Objects:

Basic Types

Three Basic Kinds Objects:

- Pointers

Basic Types

Three Basic Kinds Objects:

- Pointers
- Numbers

Basic Types

Three Basic Kinds Objects:

- Pointers
- Numbers
- File Descriptors

Basic Types

Three Basic Kinds Objects:

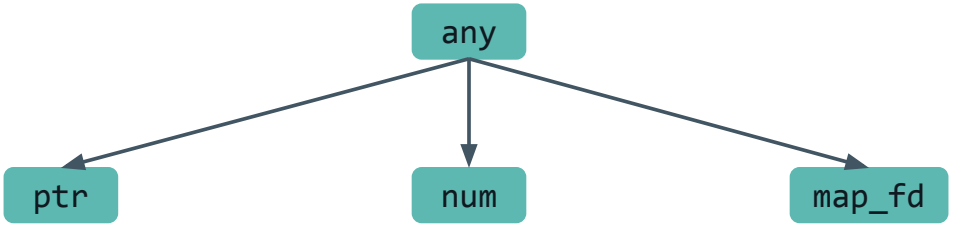
- Pointers
- Numbers
- File Descriptors

any

Basic Types

Three Basic Kinds Objects:

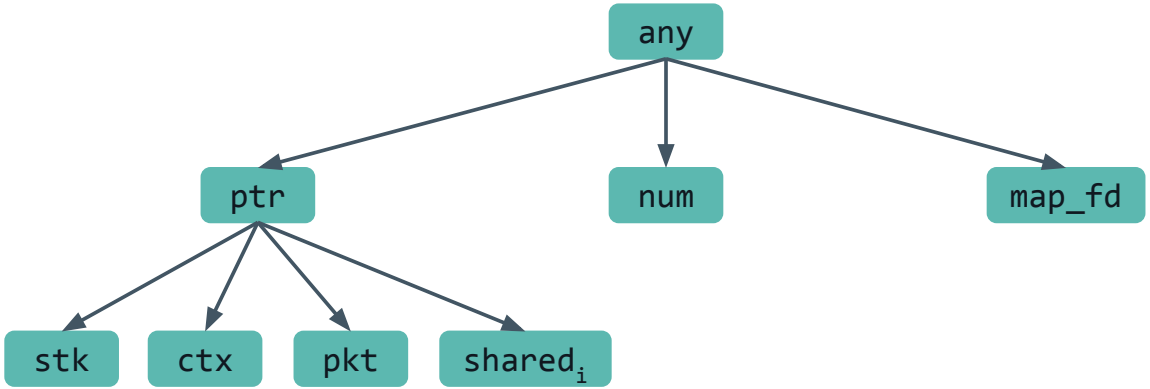
- Pointers
- Numbers
- File Descriptors



Basic Types

Three Basic Kinds Objects:

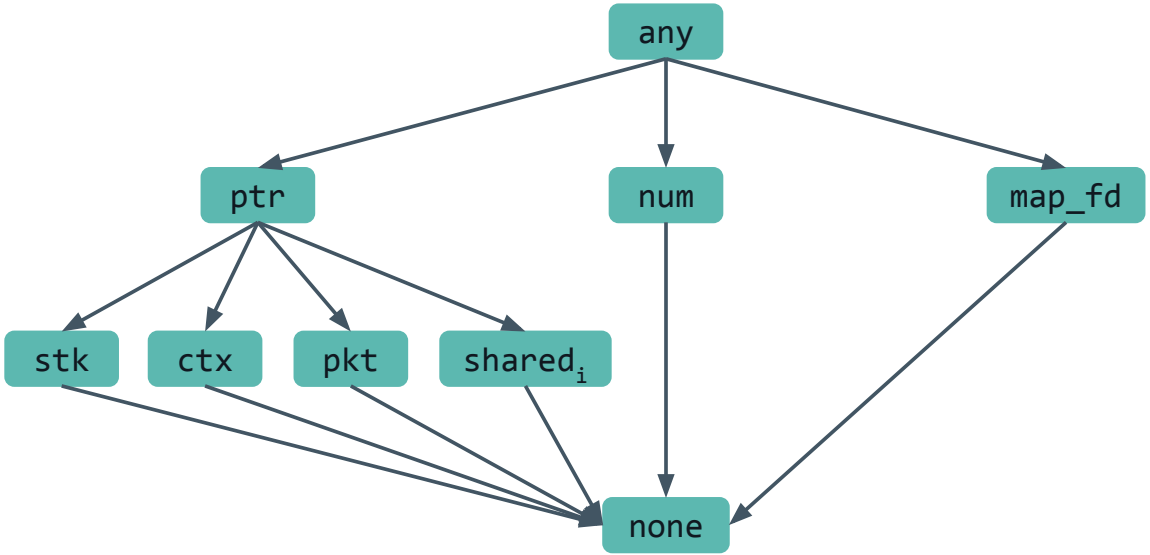
- Pointers
- Numbers
- File Descriptors



Basic Types

Three Basic Kinds Objects:

- Pointers
- Numbers
- File Descriptors

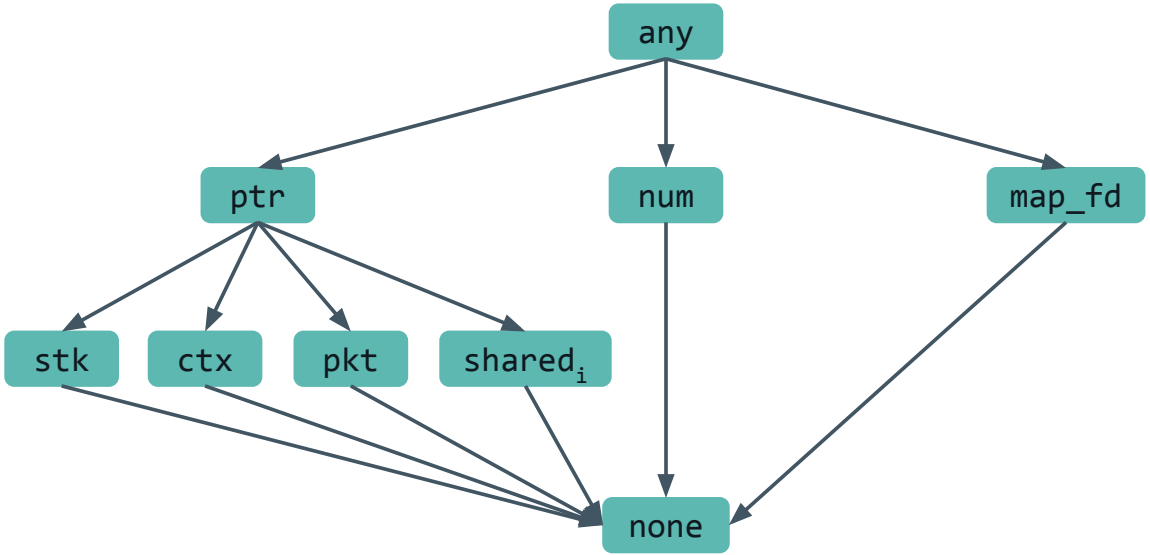


Basic Types

Three Basic Kinds Objects:

- Pointers
- Numbers
- File Descriptors

Information-Flow Safety:



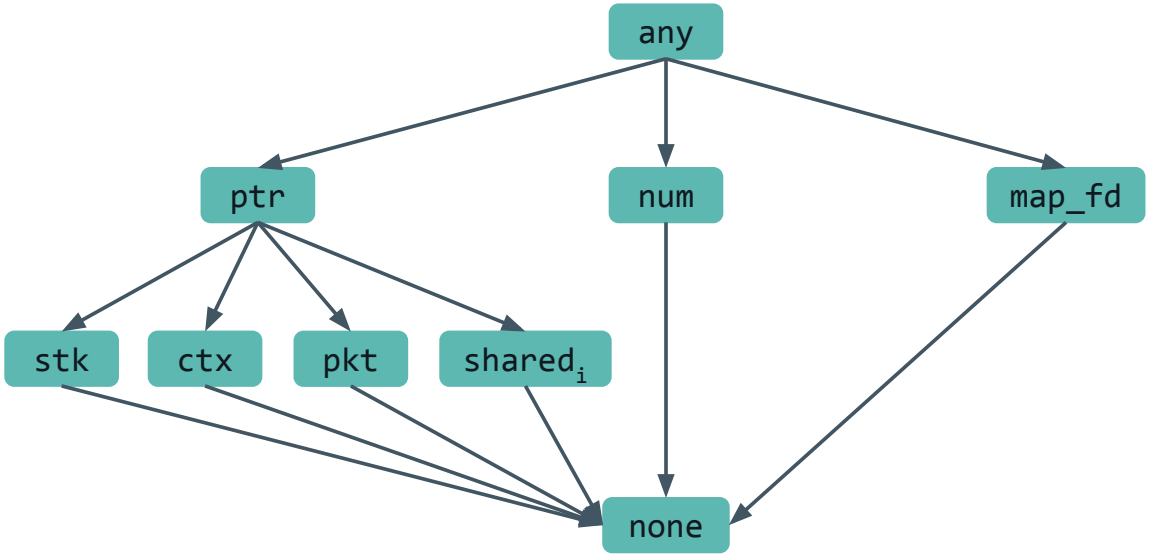
Basic Types

Three Basic Kinds Objects:

- Pointers
- Numbers
- File Descriptors

Information-Flow Safety:

- Cannot store pointer in `shared`$_i$ or `pkt` regions



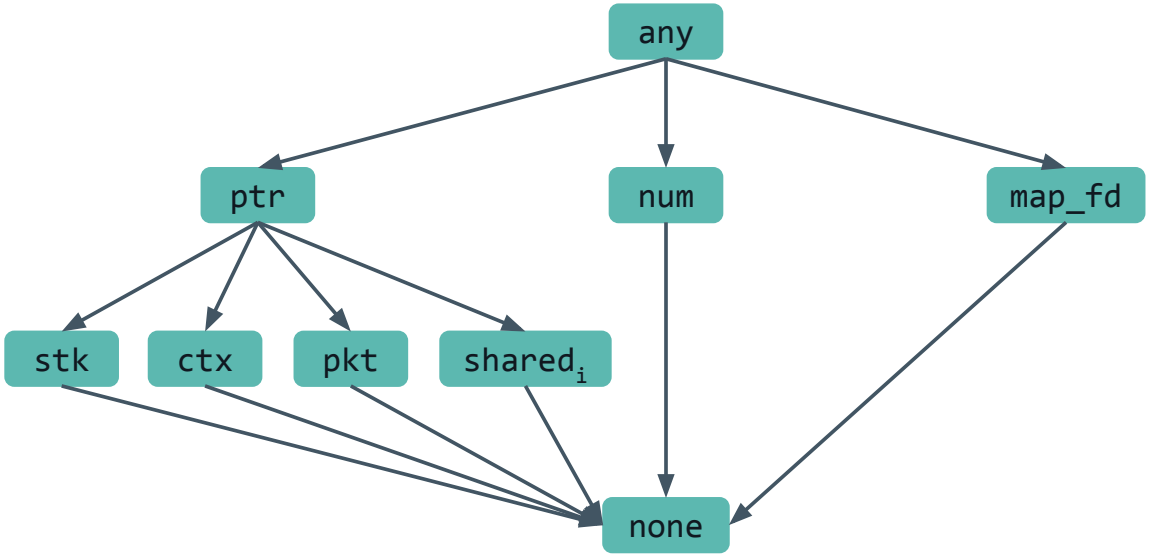
Basic Types

Three Basic Kinds Objects:

- Pointers
- Numbers
- File Descriptors

Information-Flow Safety:

- Cannot store pointer in `shared`$_i$ or `pkt` regions
- OK for `stk` region



Flow Sensitivity and Type Refinement

Type Environments:

Flow Sensitivity and Type Refinement

Type Environments:

- $\Gamma =$ Mapping of registers to types

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:

Flow Sensitivity and Type Refinement

Type Environments:

- $\Gamma =$ Mapping of registers to types
- $\Delta =$ Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

Flow Sensitivity and Type Refinement

Type Environments:

- $\Gamma =$ Mapping of registers to types
- $\Delta =$ Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{ \nu{:}\tau \mid P(\nu) \}$$

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Flow Sensitivity and Type Refinement

Type Environments:

- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:

- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{ \nu{:}\tau \mid P(\nu) \}$$

Value Variable

Refinement Predicate

Flow Sensitivity and Type Refinement

Type Environments:
- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:
- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Refinement Predicate

$\Gamma, \Delta \vdash$ `r1 := 12`

Flow Sensitivity and Type Refinement

Type Environments:
- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:
- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Refinement Predicate

$$\Gamma, \Delta \vdash \texttt{r1 := 12} \qquad \dashv \Gamma[r1 \mapsto \{\nu{:}num \mid \nu = 12\}], \Delta$$

Flow Sensitivity and Type Refinement

Type Environments:
- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:
- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Refinement Predicate

$$\Gamma, \Delta \vdash \texttt{r1 := 12} \qquad \dashv \Gamma[r1 \mapsto \{\nu{:}num \mid \nu = 12\}], \Delta$$

Flow Sensitivity and Type Refinement

FSU

Type Environments:
- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:
- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Refinement Predicate

$$\Gamma, \Delta \vdash \texttt{r1 := 12} \qquad \dashv \Gamma[\text{r1} \mapsto \{\nu{:}\text{num} \mid \nu = 12\}], \Delta$$

$$\Gamma[\text{r10} \mapsto \{\nu{:}\text{stk} \mid \nu = 512\}], \Delta \vdash \texttt{*(r10 - 4) :=}_4 \texttt{ 12}$$

Flow Sensitivity and Type Refinement

FSU

Type Environments:
- Γ = Mapping of registers to types
- Δ = Abstraction of memory regions

Type Judgements:
- $\Gamma, \Delta \vdash E : \tau$
- $\Gamma, \Delta \vdash \iota \dashv \Gamma', \Delta'$

$$\{\nu{:}\tau \mid P(\nu)\}$$

Value Variable

Refinement Predicate

$$\Gamma, \Delta \vdash \texttt{r1 := 12} \dashv \Gamma[r1 \mapsto \{\nu{:}num \mid \nu = 12\}], \Delta$$

$$\Gamma[r10 \mapsto \{\nu{:}stk \mid \nu = 512\}], \Delta \vdash \texttt{*(r10 - 4) :=}_4 \texttt{ 12} \dashv \Gamma, \Delta[stk, 508, \{\nu{:}num \mid \nu = 12\}]$$

Handling Packet Pointers

- Size of packet unknown at compile time

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)
- Invariant: `meta ≤ begin ≤ end`

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)
- Invariant: `meta ≤ begin ≤ end`
- Slack variables are an optimization

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)
- Invariant: `meta ≤ begin ≤ end`
- Slack variables are an optimization

$$\{v{:}\text{pkt} \mid v = \text{begin} + s_0 \wedge s_0 \in [0,4] \wedge \text{begin} + 8 \leq \text{end}\}$$

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)
- Invariant: `meta ≤ begin ≤ end`
- Slack variables are an optimization

$$\{v{:}\text{pkt} \mid v = \text{begin} + s_0 \wedge s_0 \in [0,4] \wedge \text{begin} + 8 \leq \text{end}\}$$

The location where the pointer is pointing to

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)
- Invariant: `meta ≤ begin ≤ end`
- Slack variables are an optimization

$$\{v{:}\text{pkt} \mid v = \text{begin} + s_0 \wedge s_0 \in [0,4] \wedge \text{begin} + 8 \leq \text{end}\}$$

The location where the pointer is pointing to

Constraint on slack variable

Handling Packet Pointers

- Size of packet unknown at compile time
- Make sure program has runtime check
- Special symbols for the packet (`meta`, `begin`, `end`)
- Invariant: `meta ≤ begin ≤ end`
- Slack variables are an optimization

$$\{v{:}\text{pkt} \mid v = \text{begin} + s_0 \wedge s_0 \in [0,4] \wedge \text{begin} + 8 \leq \text{end}\}$$

The location where the pointer is pointing to

Constraint on slack variable

Safety Condition

Handling Joins

Handling Joins

- Branches can create diverging typings

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common

$\texttt{r0} : \{\nu{:}\text{num} \mid \nu = 12\}$

Handling Joins

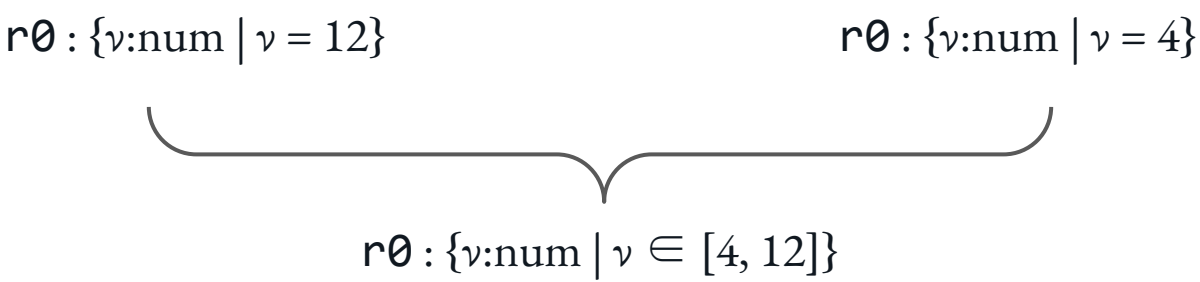
- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common

$\texttt{r0} : \{\nu{:}\text{num} \mid \nu = 12\}$

$\texttt{r0} : \{\nu{:}\text{num} \mid \nu = 4\}$

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common



Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common
- Some typings are incomparable

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common
- Some typings are incomparable

$\texttt{r0} : \{\nu{:}\text{num} \mid \nu = 12\}$

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common
- Some typings are incomparable

$\texttt{r0} : \{\nu{:}\text{num} \mid \nu = 12\}$

$\texttt{r0} : \{\nu{:}\text{stk} \mid \nu = 512\}$

Handling Joins

- Branches can create diverging typings
- What should the types be at the join point?
- Preserve information in common
- Some typings are incomparable

Typed Example

```
0   int packet_proc(struct xdp_md * ctx) {
1      void * data = (void *)(long)ctx->data;
2      void * data_end = (void *)(long)ctx->data_end;
3      if (data > data_end) exit 1;
4      *(int *) data = *(int *)(data & 0x3A53E170);
5      return 0;
6   }
```

Typed Example

FSU

```
0   int packet_proc(struct xdp_md * ctx) {
1      void * data = (void *)(long)ctx->data;
2      void * data_end = (void *)(long)ctx->data_end;
3      if (data > data_end) exit 1;
4      *(int *) data = *(int *)(data & 0x3A53E170);
5      return 0;
6   }
```

data : {v:pkt | v = begin }

Typed Example

data_end : {v:pkt | v = end}

```
0   int packet_proc(struct xdp_md * ctx) {
1      void * data = (void *)(long)ctx->data;
2      void * data_end = (void *)(long)ctx->data_end;
3      if (data > data_end) exit 1;
4      *(int *) data = *(int *)(data & 0x3A53E170);
5      return 0;
6   }
```

Typed Example

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data > data_end) exit 1;
4       *(int *) data = *(int *)(dat & 0x3A53E170);
5       return 0;
6   }
```

data : none

Typed Example

```
0   int packet_proc(struct xdp_md * ctx) {
1      void * data = (void *)(long)ctx->data;
2      void * data_end = (void *)(long)ctx->data_end;
3      if (data > data_end) exit 1;
4      *(int *) data = *(int *)(data & 0x3A53E170);
5      return 0;
6   }
```

data : {v:pkt | v = begin ∧ begin ≤ end }

Typed Example

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data + sizeof(int) > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

Typed Example

```
0   int packet_proc(struct xdp_md * ctx) {
1       void * data = (void *)(long)ctx->data;
2       void * data_end = (void *)(long)ctx->data_end;
3       if (data + sizeof(int) > data_end) exit 1;
4       *(int *) data = *(int *)(data & 0x3A53E170);
5       return 0;
6   }
```

data : {v:pkt | v = begin ∧ begin + 4 ≤ end }

Type Inference

Traverse control graph:

Type Inference

Traverse control graph:

- For one basic block:

Type Inference

Traverse control graph:

- For one basic block:
  - Join with predecessors (if any)

Type Inference

Traverse control graph:

- For one basic block:
  - Join with predecessors (if any)
  - Infer types for instructions

Type Inference

Traverse control graph:

- For one basic block:
  - Join with predecessors (if any)
  - Infer types for instructions
- For loop heads:

Type Inference

Traverse control graph:

- For one basic block:
  - Join with predecessors (if any)
  - Infer types for instructions
- For loop heads:
  - Infer types for basic blocks

Type Inference

Traverse control graph:

- For one basic block:
  - Join with predecessors (if any)
  - Infer types for instructions
- For loop heads:
  - Infer types for basic blocks
  - Widen until a fixed point is reached

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework
- Evaluated on 420 industrial and synthetic benchmarks

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework
- Evaluated on 420 industrial and synthetic benchmarks

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework
- Evaluated on 420 industrial and synthetic benchmarks
  - Cilium, Suricata, OVS, Falco, etc.

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework
- Evaluated on 420 industrial and synthetic benchmarks
  - Cilium, Suricata, OVS, Falco, etc.

	# benchmarks solved	# benchmarks verified	Unique benchs verified	Average time	Maximum time
VeRefine	420	406 (337/69)	15	0.06s	1.12s
Prevail	401	401 (332/69)	10	0.51s	14.59s

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework
- Evaluated on 420 industrial and synthetic benchmarks
  - Cilium, Suricata, OVS, Falco, etc.
- Less precise than prevail (but still sufficiently precise)

	# benchmarks solved	# benchmarks verified	Unique benchs verified	Average time	Maximum time
VeRefine	420	406 (337/69)	15	0.06s	1.12s
Prevail	401	401 (332/69)	10	0.51s	14.59s

Implementation and Evaluation

- Type inference algorithm implemented in a tool called VeRefine
- Implemented as an abstract domain in the PREVAIL framework
- Evaluated on 420 industrial and synthetic benchmarks
  - Cilium, Suricata, OVS, Falco, etc.
- Less precise than prevail (but still sufficiently precise)
- Faster than PREVAIL

	# benchmarks solved	# benchmarks verified	Unique benchs verified	Average time	Maximum time
VeRefine	420	406 (337/69)	15	0.06s	1.12s
Prevail	401	401 (332/69)	10	0.51s	14.59s

Conclusion

Summary:

Conclusion

Summary:

- Type system for verifying eBPF Programs

Conclusion

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types

Conclusion

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types
- Improved modularity of eBPF verification

Conclusion

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types
- Improved modularity of eBPF verification
- Improved performance

Conclusion

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types
- Improved modularity of eBPF verification
- Improved performance

Future Work:

Conclusion

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types
- Improved modularity of eBPF verification
- Improved performance

Future Work:

- Type checker

Conclusion

FSU

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types
- Improved modularity of eBPF verification
- Improved performance

Future Work:

- Type checker
- Termination analysis

Conclusion

FSU

Summary:

- Type system for verifying eBPF Programs
- Enhanced debuggability through the use of types
- Improved modularity of eBPF verification
- Improved performance

Future Work:

- Type checker
- Termination analysis

Code Available Here