14 May 2024

Value tracking in BPF verifier

Shung-Hsi Yu

LSF/MM/BPF 2024

About me

Shung-Hsi Yu (pronounced like Shawn see you)

Works at 

About me

Maintains BPF stack of SUSE Enterprise Linux and openSUSE

- BPF subsystem, libbpf, bpftool, bpftrace, bcc, xdp-tools
  - backport CVE fixes (along with selftests)
    - BPF verifier

Wish

The verifier is **simpler** to understand

More people **understand** the verifier

More thorough **testing** of the verifier

Wish

The verifier is **simpler** to understand

More people **understand** the verifier

More thorough **testing** of the verifier

- Agni/Verifying the Verifier
- range bounds tester (reg_bounds.c)

Wish

The verifier is **simpler** to understand

More people **understand** the verifier

~~More thorough **testing** of the verifier~~

- Agni/Verifying the Verifier
- range bounds tester (reg_bounds.c)

Wish

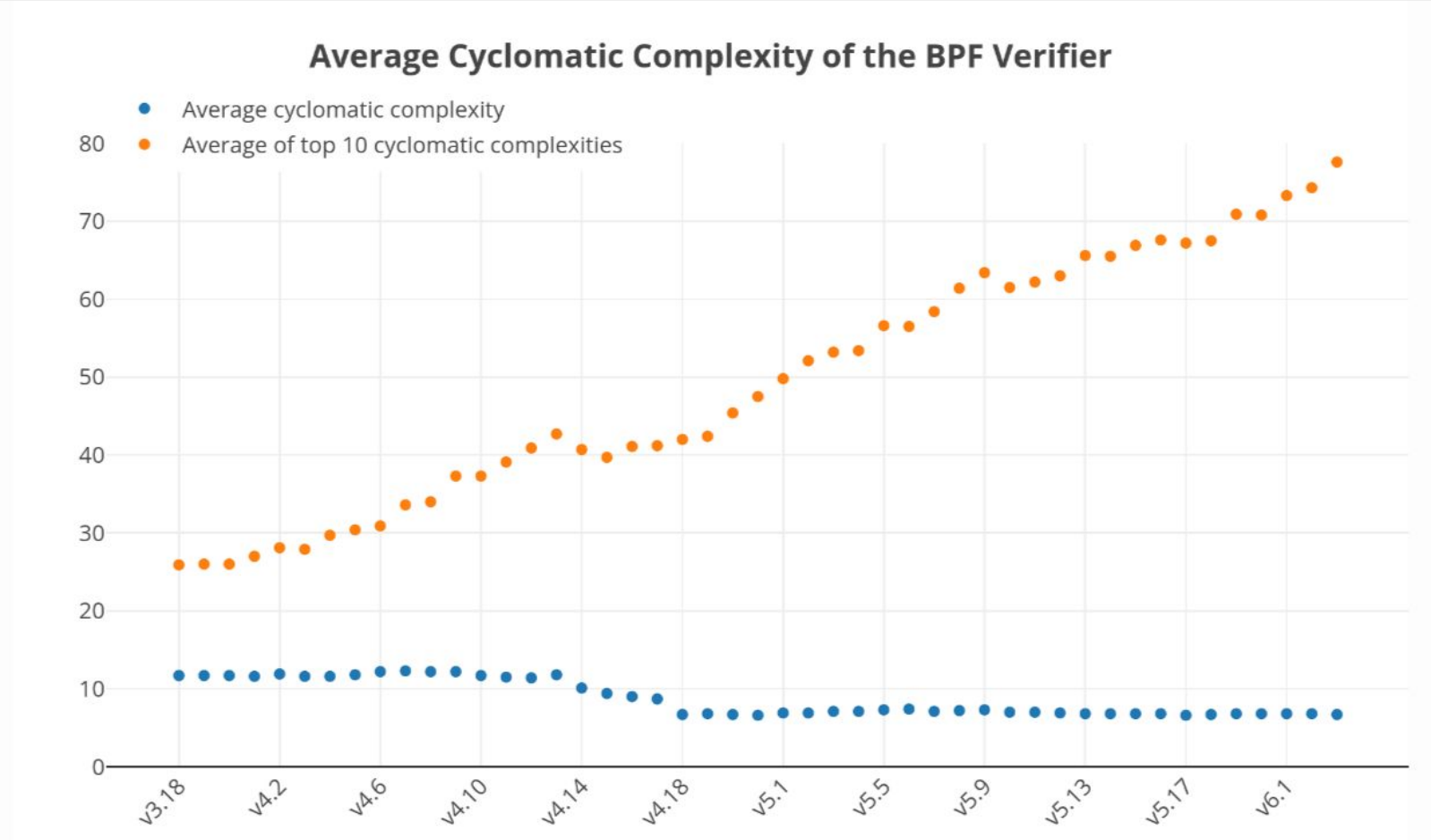
The verifier is **simpler** to understand

More people **understand** the verifier

Wish

The verifier is **simpler** to understand

More people **understand** the verifier



"Complexity of the BPF Verifier" by Paul Chaignon

Focus

I still don’t understand **BPF verifier** as a whole

Focus

BPF verifier

Focus

value tracking

dead code elimination

control flow analysis

BPF verifier

backtracking

spectre mitigation

type tracking

liveness tracking

instruction rewrite

Focus

value tracking

dead code elimination

control flow analysis

BPF verifier

backtracking

spectre mitigation

type tracking

liveness tracking

instruction rewrite

SUSE

Value Tracking

Why it is used

```
e = bpf_map_lookup_elem();
val = *(e + offset); /* out of bound? */
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* back in the days */
    s64 min_value;
    u64 max_value;
    u32 min_align;
    u32 aux_off;
    u32 aux_off_align;
    bool value_from_signed;
};
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

Tnum

Short for **tri-state number** or **tracked number**

```
struct tnum {
    u64 value; /* which bits are set (if known) */
    u64 mask;  /* which bits are _un_known */
};
```

SUSE

Tnum

Short for **tri-state number** or **tracked number**

Tracks knowledge about the bits of a value

- each bit can be either **known (0 or 1)**, or **unknown (x)**

Tnum

Each bit can be either known (0 or 1), or unknown (x)

```
    { 0b00 }     => 00 => mask=0b00, value=0b00
    { 0b01 }     => 01 => mask=0b00, value=0b01
{ 0b00, 0b01 } => 0x => mask=0b01, value=0b00
{ 0b00..0b11 } => xx => mask=0b11, value=0b00
```

SUSE

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

Range a.k.a interval domain

Tracks **minimum** and **maximum** possible value

SUSE

Range a.k.a interval domain

Tracks minimum and maximum possible value

```
  { 0 }  => umin(_value)=0, umax(_value)=0
{ 0, 1 } => umin(_value)=0, umax(_value)=1
```

SUSE

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

Value Tracking

Efficient data structure to track a **set** of values

- **in expense** of not being able to track precisely

Value Tracking

In expense of not being able to track precisely

```
  { 1, 3 }    => umin=1, umax=3   =>   { 1, 2, 3 }

{ 0b01, 0b10 } => mask=0b11, val=0b00 =>  { 0b00..0b11 }
```

Value Tracking

Why both tnum and ranges?

```
   { 1, 3 }     =>    umin=1, umax=3     => { 1, 2, 3 }
{ 0b01, 0b11 } => mask=0b10, val=0b01 => { 0b01, 0b11 }
                                       => { 1, 3 }
```

Value Tracking

Why both tnum and ranges?

```
   { 1, 2 } =>   umin=1, umax=2    => { 1, 2 }
{ 0b01, 0b10 } => mask=0b10, val=0b01 => { 0b00..0b11 }
                                      => { 0b01, 0b10 }
```

Value Tracking

Why both signed and unsigned ranges?

```
BPF_LD_IMM64(src, 0xfffffffffffffffe) /* U64MAX-1 */
BPF_JMP_REG(BPF_JLT, dst, src, off)

BPF_LD_IMM64(src, 0xfffffffffffffffe) /* -2 */
BPF_JMP_REG(BPF_JSLT, dst, src, off)
```

SUSE

Value Tracking

Why both 64-bit and 32-bit ranges?

```
/* Unsigned comparison of full 64-bit in register */
BPF_JMP_IMM(BPF_JLT, dst, imm, off)

/* Unsigned comparison of lower 32-bit in register */
BPF_JMP32_IMM(BPF_JLT, dst, imm, off)
```

Wish

The verifier is **simpler** to understand

More people **understand** the verifier

Less bounds

Things we do for precision

Propagating the knowledge about possible values

```
static void reg_bounds_sync(struct bpf_reg_state *reg)
{
    /* tnum -> u64, s64, u32, s32 */
    __update_reg_bounds(reg);
    /* u64 -> u32, s32; s64 -> u32, s32
     * u64 -> s64; s64 -> u64
     * u32 -> u64, s64; s32 -> u64, s64 */
    __reg_deduce_bounds(reg);
    __reg_deduce_bounds(reg); /* 2nd time */
    /* u64 -> tnum; u32 -> tnum */
    __reg_bound_offset(reg);
    /* tnum -> u64, s64, u32, s32 */
    __update_reg_bounds(reg);
}
```

```
static void reg_bounds_sync(struct bpf_reg_state *reg)
{
    /* tnum -> u64, s64, u32, s32 */
    __update_reg_bounds(reg);
    /* u64 -> u32, s32; s64 -> u32, s32
     * u64 -> s64; s64 -> u64
     * u32 -> u64, s64; s32 -> u64, s64 */
    __reg_deduce_bounds(reg);
    __reg_deduce_bounds(reg); /* 2nd time */
    /* u64 -> tnum; u32 -> tnum */
    __reg_bound_offset(reg);
    /* tnum -> u64, s64, u32, s32 */
    __update_reg_bounds(reg);
}
```

```
static void reg_bounds_sync(struct bpf_reg_state *reg)
{
    /* tnum -> u64, s64, u32, s32 */
    __update_reg_bounds(reg);
    /* u64 -> u32, s32; s64 -> u32, s32
     * u64 -> s64; s64 -> u64
     * u32 -> u64, s64; s32 -> u64, s64 */
    __reg_deduce_bounds(reg);
    __reg_deduce_bounds(reg); /* 2nd time */
    /* u64 -> tnum; u32 -> tnum */
    __reg_bound_offset(reg);
    /* tnum -> u64, s64, u32, s32 */
    __update_reg_bounds(reg);
}
```

```c
static void __reg64_deduce_bounds(struct bpf_reg_state *reg)
{
    /* u64 -> s64 */
    if ((s64)reg->umin_value <= (s64)reg->umax_value) {
     reg->smin_value = max_t(s64, reg->smin_value, reg->umin_value);
     reg->smax_value = min_t(s64, reg->smax_value, reg->umax_value);
    }
    /* s64 -> u64 */
    if ((u64)reg->smin_value <= (u64)reg->smax_value) {
     reg->umin_value = max_t(u64, reg->smin_value, reg->umin_value);
     reg->umax_value = min_t(u64, reg->smax_value, reg->umax_value);
    }
}
```

```
tnum -> u64, s64, u32, s32
u64 -> s64
s64 -> u64
u64 -> u32, s32
s64 -> u32, s32
u32 -> u64, s64
s32 -> u64, s64
u32 -> s32
s32 -> u32
u64 -> tnum
u32 -> tnum
s64 -> tnum
s32 -> tnum
```

Things we do for precision

Less propagation?

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 3 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    u64 min_value; /* allow min > max */
    u64 max_value;
    u32 subreg_min_value; /* allow min > max */
    u32 subreg_max_value;
};
```

Wrapped Range

Tracks the possible starting **from min**, and all value encountered by **iteratively adding 1**, **until umax**

```
min=0xfffffffffffffffe, max=0
                    => { 0, UMAX-1, UMAX }
     inverted range => { 0..UMAX } - { max+1..min-1 }
```

SUSE

Wrapped Range

Tracks the possible starting **from min**, and all value encountered by **iteratively adding 1**, **until umax**

```
min=0xfffffffffffffffe, max=0
                    => { 0, UMAX-1, UMAX }
    inverted range => { 0..UMAX } - { max+1..min-1 }
                    => { -2, -1, 0 }
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 3 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    u64 min_value; /* allow min > max */
    u64 max_value;
    u32 subreg_min_value; /* allow min > max */
    u32 subreg_max_value;
};
```

```
tnum -> u64, s64, u32, s32
u64 -> s64
s64 -> u64
u64 -> u32, s32
s64 -> u32, s32
u32 -> u64, s64
s32 -> u64, s64
u32 -> s32
s32 -> u32
u64 -> tnum
u32 -> tnum
s64 -> tnum
s32 -> tnum
```

```
tnum -> u64, u32
u64 -> u32
u32 -> u64
u64 -> tnum
u32 -> tnum
```

Reference

RFC Unifying signed and unsigned min/max tracking

Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss

```c
static void scalar32_min_max_add(struct bpf_reg_state *dst_reg,
                 struct bpf_reg_state *src_reg)
{
    s32 smin_val = src_reg->s32_min_value;
    s32 smax_val = src_reg->s32_max_value;
    u32 umin_val = src_reg->u32_min_value;
    u32 umax_val = src_reg->u32_max_value;

    if (signed_add32_overflows(dst_reg->s32_min_value, smin_val) ||
     signed_add32_overflows(dst_reg->s32_max_value, smax_val)) {
      dst_reg->s32_min_value = S32_MIN;
      dst_reg->s32_max_value = S32_MAX;
    } else {
      dst_reg->s32_min_value += smin_val;
      dst_reg->s32_max_value += smax_val;
    }
    if (dst_reg->u32_min_value + umin_val < umin_val ||
     dst_reg->u32_max_value + umax_val < umax_val) {
      dst_reg->u32_min_value = 0;
      dst_reg->u32_max_value = U32_MAX;
    } else {
      dst_reg->u32_min_value += umin_val;
      dst_reg->u32_max_value += umax_val;
    }
}
```

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {
    struct tnum var_off; /* possible bit pattern */
    struct wrange { /* wrapped range */
        u64 start;
        u64 end;
    } wr64;
    struct wrange32 {
        u32 start;
        u32 end;
    } wr32;
};
```

```
struct wrange32 wrange32_add(struct wrange32 a,
                             struct wrange32 b)
{
    u32 a_len = a.end - a.start;
    u32 b_len = b.end - b.start;
    u32 new_len = a_len + b_len;

    /* the new start/end pair goes full circle
     * so any value is possible */
    if (new_len < a_len || new_len < b_len)
        return WRANGE32(U32_MIN, U32_MAX);
    else
        return WRANGE32(a.start + b.start, a.end + b.end);
}
```

Plan

Make wrapped range fit in the current ecosystem

Plan

1. Create helper that transform current signed and unsigned min/max from/to wrange
2. Use `wrange*_{add,sub,...}` inside `scalar*_min_max_{add,sub,...}` instead
3. Run selftests to check
4. Switch to `wrange` in `struct bpf_reg_state`

Concerns

- Hidden/implicit assumptions of value tracking
  - does umax/umin and smax/smin *always* intersects?
  - …

Abstract value tracking details

Implementation detail

Requires knowing tnum and range to work on verifier

- `umin` or `var_off.value` for minimum value?
- `tnum_is_const()` or `umin == umax`?
- how to get maximum offset? `var_off` or `umax`?
  (don't forget the '`off`' field for base offset)

```
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {   /* track 5 kinds of bounds */
    struct tnum var_off; /* possible bit pattern */
    s64 smin_value; /* minimum possible (s64)value */
    s64 smax_value; /* maximum possible (s64)value */
    u64 umin_value; /* minimum possible (u64)value */
    u64 umax_value; /* maximum possible (u64)value */
    s32 s32_min_value; /* min (s32)value */
    s32 s32_max_value; /* max (s32)value */
    u32 u32_min_value; /* min (u32)value */
    u32 u32_max_value; /* max (u32)value */
};
```

```c
/* include/linux/bpf_verifier.h */
struct bpf_reg_state {
    struct tval val;
    /* ... */
};
```

```
static int is_scalar_branch_taken(
        struct bpf_reg_state *reg1, struct bpf_reg_state *reg2,
         u8 opcode, bool is_jmp32)
{
    /* ... */
    switch (opcode) {
    case BPF_JEQ:
     if (tnum_is_const(t1) && tnum_is_const(t2) &&
         t1.value == t2.value)
         return ALWAYS;
     /* non-overlapping ranges */
     if (umin1 > umax2 || umax1 < umin2)
         return NEVER;
     if (smin1 > smax2 || smax1 < smin2)
         return NEVER;
     return MAYBE;
```

```
static int is_scalar_branch_taken(
      struct bpf_reg_state *reg1,
      struct bpf_reg_state *reg2,
       u8 opcode, bool is_jmp32)
{
    /* ... */
    switch (opcode) {
    case BPF_JEQ:
        intersects = tval_intersect(reg1->val, reg2->val2,
                                    &out);
       if (!intersects)
           return NEVER;
       return tval_eq(reg1->val, reg2->val) ? ALWAYS : MAYBE;
```

```
static void regs_refine_cond_op(
            struct bpf_reg_state *reg1,
            struct bpf_reg_state *reg2,
                u8 opcode, bool is_jmp32)
{
    switch (opcode) {
    case BPF_JEQ:
        reg1->umin_value = max(reg1->umin_value, reg2->umin_value);
        reg1->umax_value = min(reg1->umax_value, reg2->umax_value);
        reg1->smin_value = max(reg1->smin_value, reg2->smin_value);
        reg1->smax_value = min(reg1->smax_value, reg2->smax_value);
        reg2->umin_value = reg1->umin_value;
        reg2->umax_value = reg1->umax_value;
        reg2->smin_value = reg1->smin_value;
        reg2->smax_value = reg1->smax_value;
        reg1->var_off = tnum_intersect(reg1->var_off, reg2->var_off);
        reg2->var_off = reg1->var_off;
```

```
static int regs_refine_cond_op(
            struct bpf_reg_state *reg1,
            struct bpf_reg_state *reg2,
             u8 opcode, bool is_jmp32)
{
    switch (opcode) {
    case BPF_JEQ:
    if(!tval_intersect(reg1->val, reg2->val, &out))
            return -EINVAL; /* should not happen */
    reg2->val = reg1->val = out;
    /* ... */
```

tval helpers

```
/* copy tnum */
tval_{add,sub,mul,div}()
tval_{and,or,xor}()
tval_{l,r,ar}shift()
/* minimum and maximum */
tval_{u,s}{min,max}()
```

```
/* with __must_check */
tval_intersect()
tval_diff()
tval_union()
```

SUSE

Question

- How much abstraction is too much?
- `__must_check` semantic too verbose?
- …

Other topics

Other verifier topics

- **Documentation** improvement
- Simplification/refactoring of codebase
- Removing `tnum` from `bpf_reg_state`
- Tracks complexity metric of verifier

Other verifier topics

- Standardization for verifier
- Testing across different verifier
- Further reducing loop/branch states
- Lazier precision tracking
- …

Thank you! and ...

Shameless plug

BPF BoF in Asia

Taipei

- (TBD) COSCUP, August 3-4th

Tokyo

- (TBD) OSS Japan, October 28-29th
- (TBD) openSUSE Asia Summit, November 2-3rd

SUSE