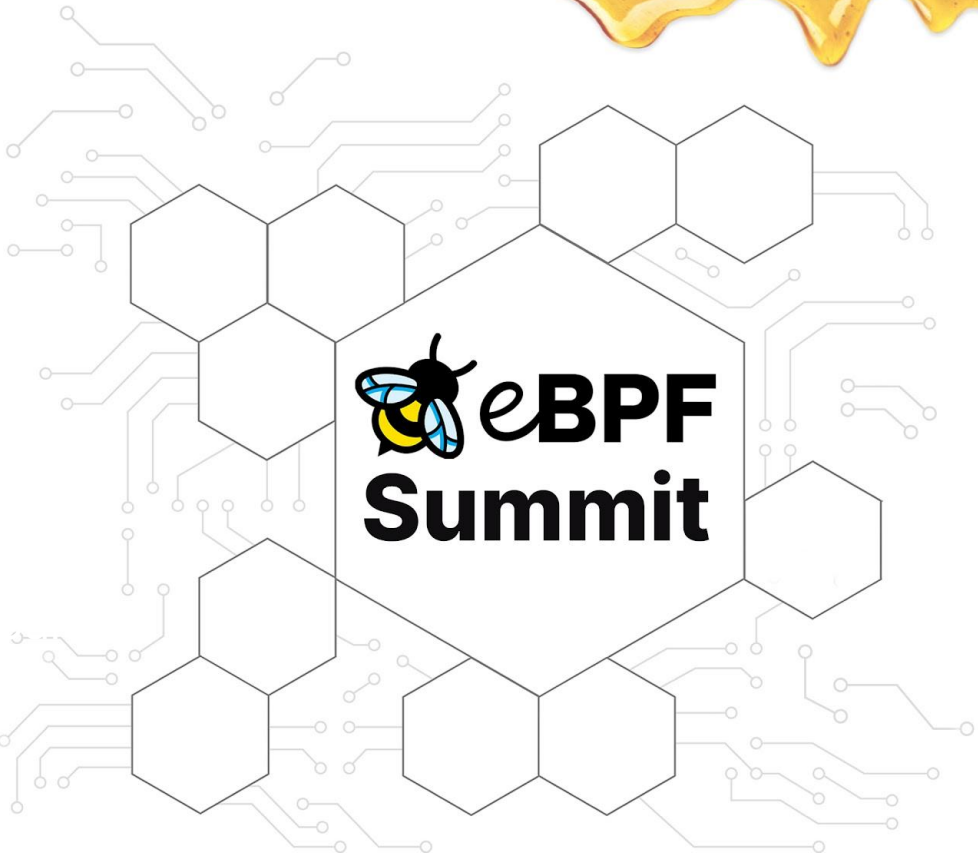
Peeking into the eBPF verifier



Shung-Hsi Yu

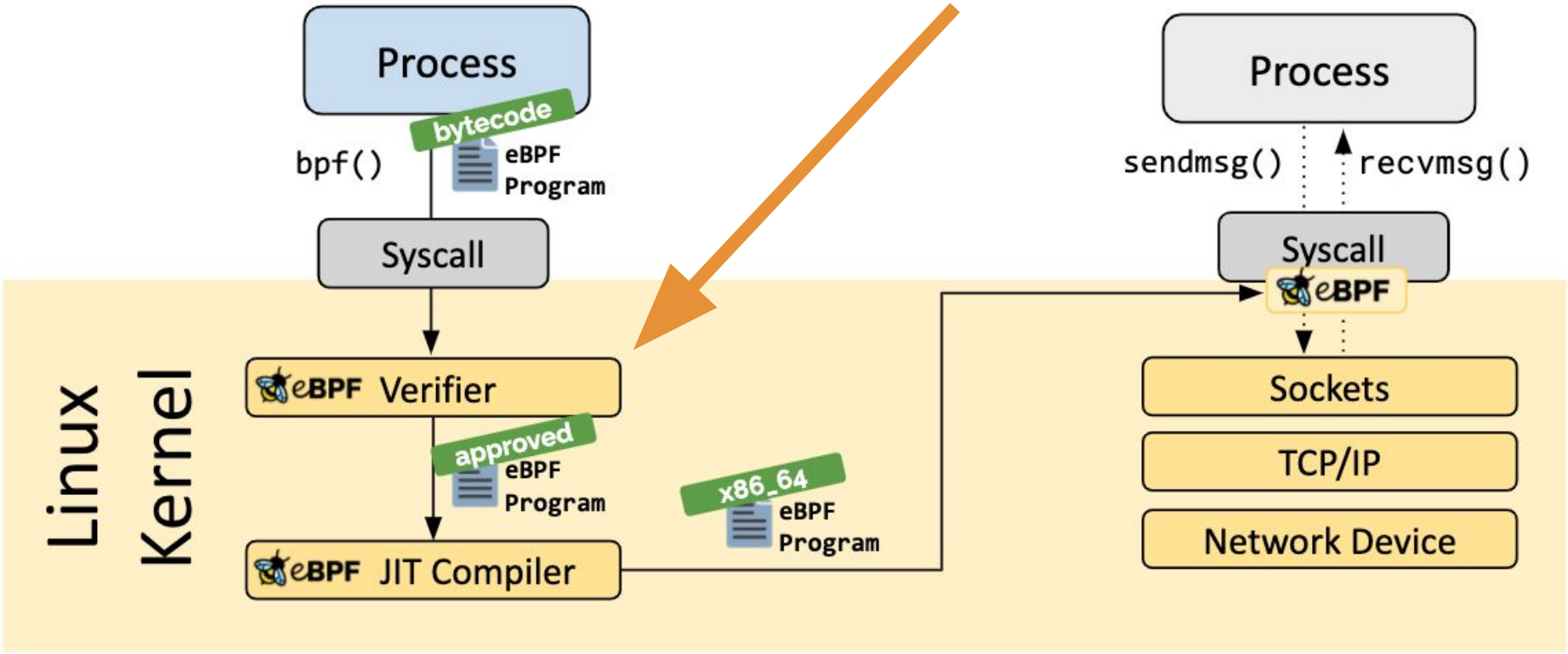
@shunghsiyu

What is the verifier?

*The only thing sitting between our eBPF programs and a **deep dark chasm of destruction** is the eBPF verifier*

— David Miller (netdev maintainer)

What is the verifier?



How does it works?

How does it works?

Type & Value

Example 1

```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Drops all packets */
    return 0;
}
```

Example 1

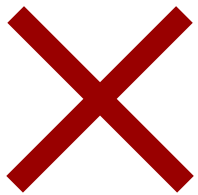
```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Drops all packets */
    return 0;
}   int == typeof(0)
```

Example 1.1

```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Leaking kernel pointer is a security risk */
    return skb;
}
```

Example 1.1

```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Leaking kernel pointer is a security risk */
    return skb;
}   int != struct __sk_buff *
```



What about compilers?

What about compilers?

They **trust** the users too much

Example 1.2

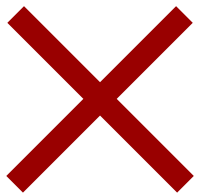
```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Lying gets you pass the compiler,
     * but not the verifier. */
    return        skb;
}
```

Example 1.2

```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Lying gets you pass the compiler,
     * but not the verifier. */
    return (long) skb;
}
```

Example 1.2

```
SEC("socket1")
int bpf_prog(struct __sk_buff *skb)
{
    /* Lying gets you pass the compiler,
     * but not the verifier. */
    return (long) skb;
}
```



What verifier sees

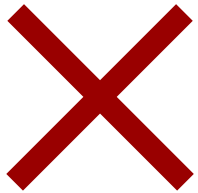
```
SEC("socket1")
    bpf_prog(                    skb)
{
    /* Lying gets you pass the compiler,
     * but not the verifier. */
    return        skb;
}
```

Verifier’s type systems

```
   SEC("socket1")
SCALAR bpf_prog( PTR_TO_CTX        skb)
   {
      /* Lying gets you pass the compiler,
       * but not the verifier. */
      return        skb;
   }
```

Verifier’s type systems

```
   SEC("socket1")
SCALAR bpf_prog( PTR_TO_CTX        skb)
   {
      /* Lying gets you pass the compiler,
       * but not the verifier. */
      return         skb;
   }  SCALAR   !=   PTR_TO_CTX
```



How does it works?

Type & Value

Example 2

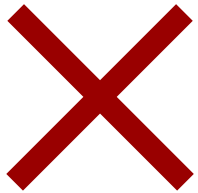
```
unsigned int i; /* externally initialized */
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{
    /* Verifier does not allow you to do this */
    return arr[i];
}
```

Example 2

```
unsigned int i; /* externally initialized */
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{
    /* Verifier does not allow you to do this */
    return arr[i];
}
```



Why?

Why?

The verifier is pessimistic

Half empty or half full?

Glass icons created by Freepik - Flaticon

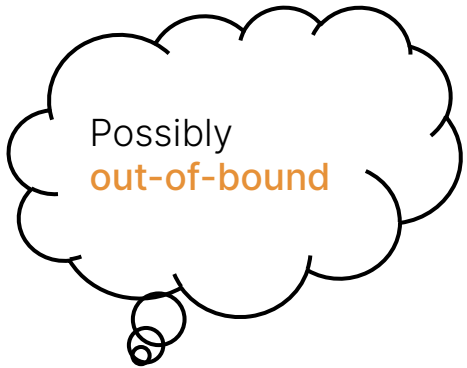
Out-of or within bound?

```
arr[i]
```

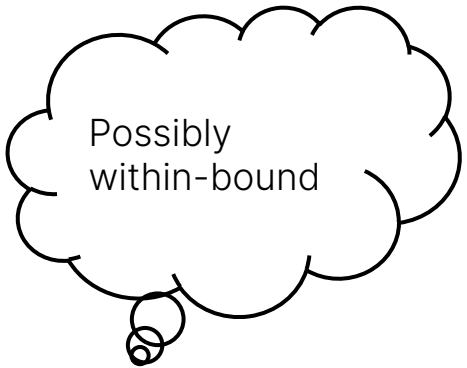
verifier

compiler

Out-of or within bound?




```
arr[i]
```



verifier

compiler

How the verifier reasons

```
unsigned int i;
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{

    return arr[i];
}
```

How the verifier reasons

```
unsigned int i; /* externally initialized */
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{

    return arr[i];
}
```

How the verifier reasons

```
unsigned int i; /* externally initialized */
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    return arr[i];
}
```

How the verifier reasons

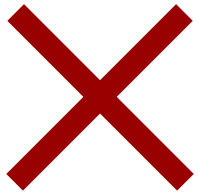
```
unsigned int i; /* externally initialized */
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    return arr[i]; /* arr[2^32-1] is out-of-bound! */
}
```

How the verifier reasons

```
unsigned int i; /* externally initialized */
char arr[4] = { ... };

SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    return arr[i]; /* arr[2^32-1] is out-of-bound! */
}
```



Making the verifier happy

```
SEC("sock_filter")
int bpf_prog(void)
{

    if (i <= 3)

        return arr[i];
    else

        return 0;
}
```

Making the verifier happy

```
SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    if (i <= 3)

        return arr[i];
    else

        return 0;
}
```

Making the verifier happy

```
SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    if (i <= 3)
        /* i <= 3 == true, so 0 ≤ i ≤ 3 */
        return arr[i];
    else

        return 0;
}
```

Making the verifier happy

```
SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    if (i <= 3)
        /* i <= 3 == true, so 0 ≤ i ≤ 3 */
        return arr[i]; /* arr[i] always within-bound */
    else

        return 0;
}
```

Making the verifier happy

```
SEC("sock_filter")
int bpf_prog(void)
{
    /* i can be any unsigned int, 0 ≤ i ≤ 2^32-1 */
    if (i <= 3)
        /* i <= 3 == true, so 0 ≤ i ≤ 3 */
        return arr[i]; /* arr[i] always within-bound */
    else
        /* i <= 3 == false, so 4 ≤ i ≤ 2^32-1 */
        return 0;
}
```

Making the verifier happy (unconditionally)

```
SEC("sock_filter")
int bpf_prog(void)
{
    unsigned int i2 = i;
    /* i2 can be any unsigned int, 0 ≤ i2 ≤ 2^32-1 */

    i2 &= 0x3; /* apply mask to i2, now 0 ≤ i2 ≤ 3 */

    return arr[i2]; /* arr[i2] always within-bound */
}
```

Confession

Confession

That's not how verifier **really** works

eBPF bytecodes & registers

```
unsigned int i2 = i;

i2 &= 0x3;

return arr[i2];
```

```
r1 = 0 ll (&i);
r2 = *(u32 *)(r1 + 0);

r2 = r2 & 0x3

r3 = 0 ll (&arr);
r3 = r3 + r2;
r0 = *(u32 *)(r3 + 0);
exit;
```

Conceptually

It’s close enough

eBPF bytecodes & registers

```
unsigned int i2 = i;

i2 &= 0x3;

return arr[i2];
```

```
r1 = 0 ll (&i);
r2 = *(u32 *)(r1 + 0);

r2 = r2 & 0x3

r3 = 0 ll (&arr);
r3 = r3 + r2;
r0 = *(u32 *)(r3 + 0);
exit;
```

eBPF bytecodes & registers

```
unsigned int i2 = i;

i2 &= 0x3;

return arr[i2];
```

```
r1 = 0 ll (&i);
r2 = *(u32 *)(r1 + 0);

r2 = r2 & 0x3

r3 = 0 ll (&arr);
r3 = r3 + r2;
r0 = *(u32 *)(r3 + 0);
exit;
```

Summary

Type & Value

Thank you for listening 

Other things that the verifiers does

Make sure eBPF program terminates properly

Modifies eBPF programs to

- Replace map file-descriptor with map pointer
- Make division-by-zero impossible
- Dead-code sanitization
- Spectre mitigation
- Bound limiting

Resources

“Peeking into the BPF verifier” slide deck — COSCUP 2022

BPF Verifier Overview — XDP Newbie

Safe Programs The Foundation of BPF — eBPF Summit 2020

BPF and Spectre: Mitigating transient execution attacks — eBPF Summit 2021

CVE-2020-8835: Linux Kernel Privilege Escalation via Improper Verification

Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers