第三届 eBPF开发者大会

www.ebpftravel.com

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

WeChat

HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64

张 培 华

中国·西安

Outline

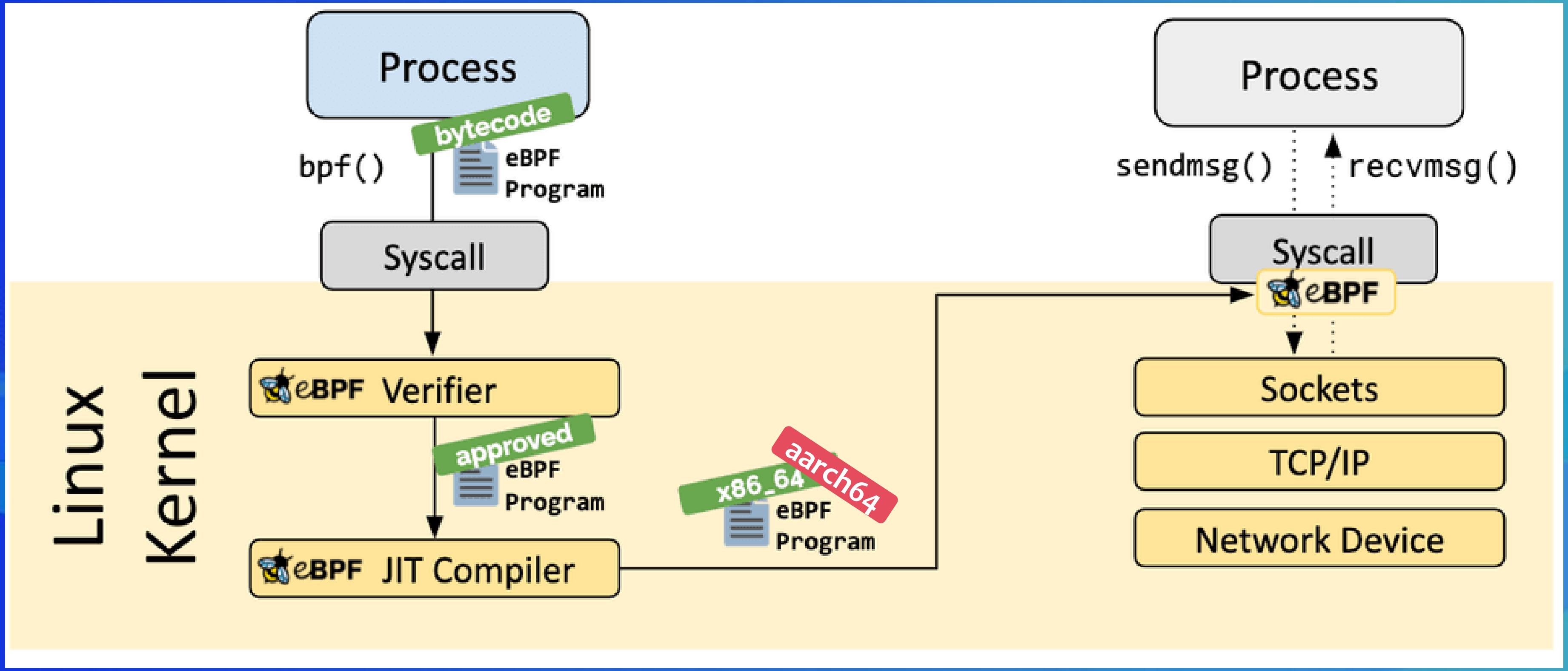
- What is eBPF?
- Verifier
- High-level Design
- Detailed Design
- Evaluation
- Conclusion

Outline

- What is eBPF?
- Verifier
- Conclusion
- High-level Design
- Evaluation
- Detailed Design

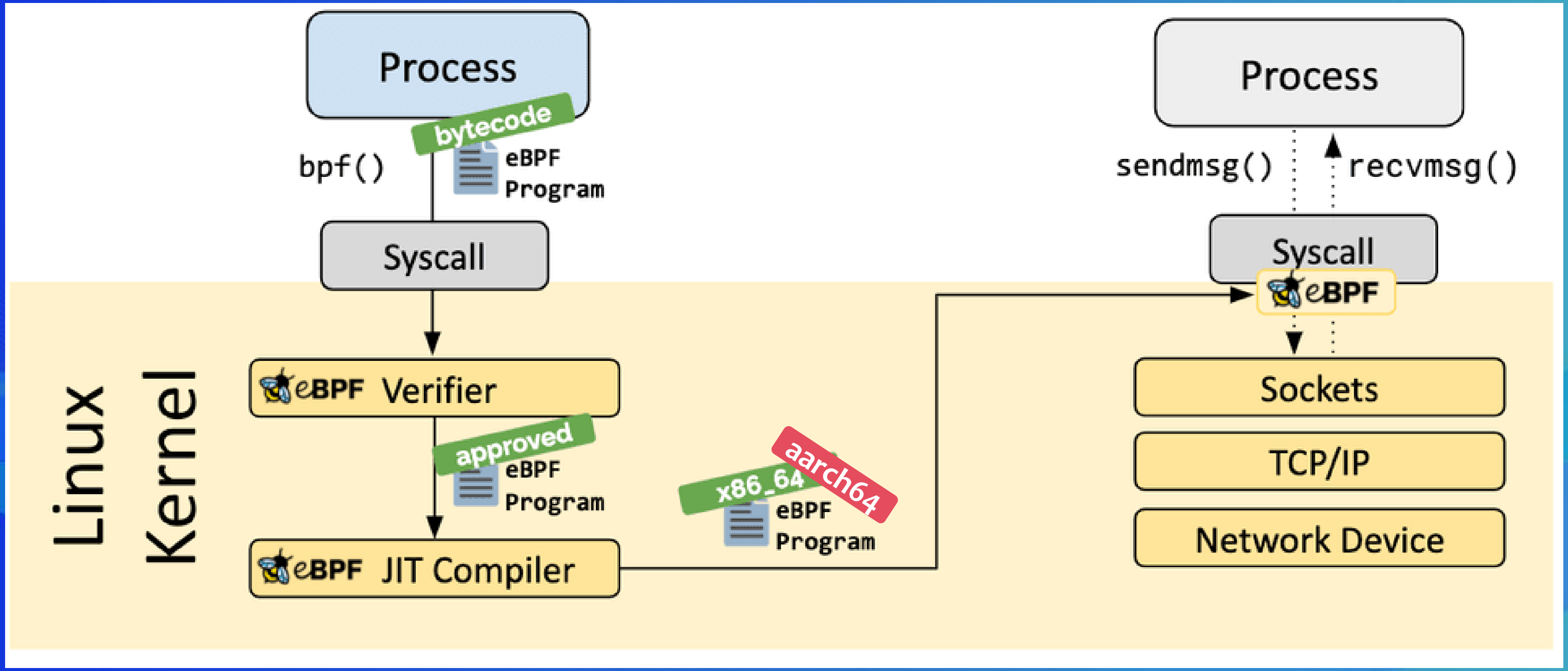
What is extended Berkeley Packet Filter (eBPF) ?

eBPF can be used to safely extend the kernel without changing kernel code or loading kernel modules



What is extended Berkeley Packet Filter (eBPF) ?

kernel provides an execution environment for eBPF



- Registers (r0-r10)
- Program Context
- Maps (key-value)
- Helper Functions
- BPF (Kernel) Stack

Outline

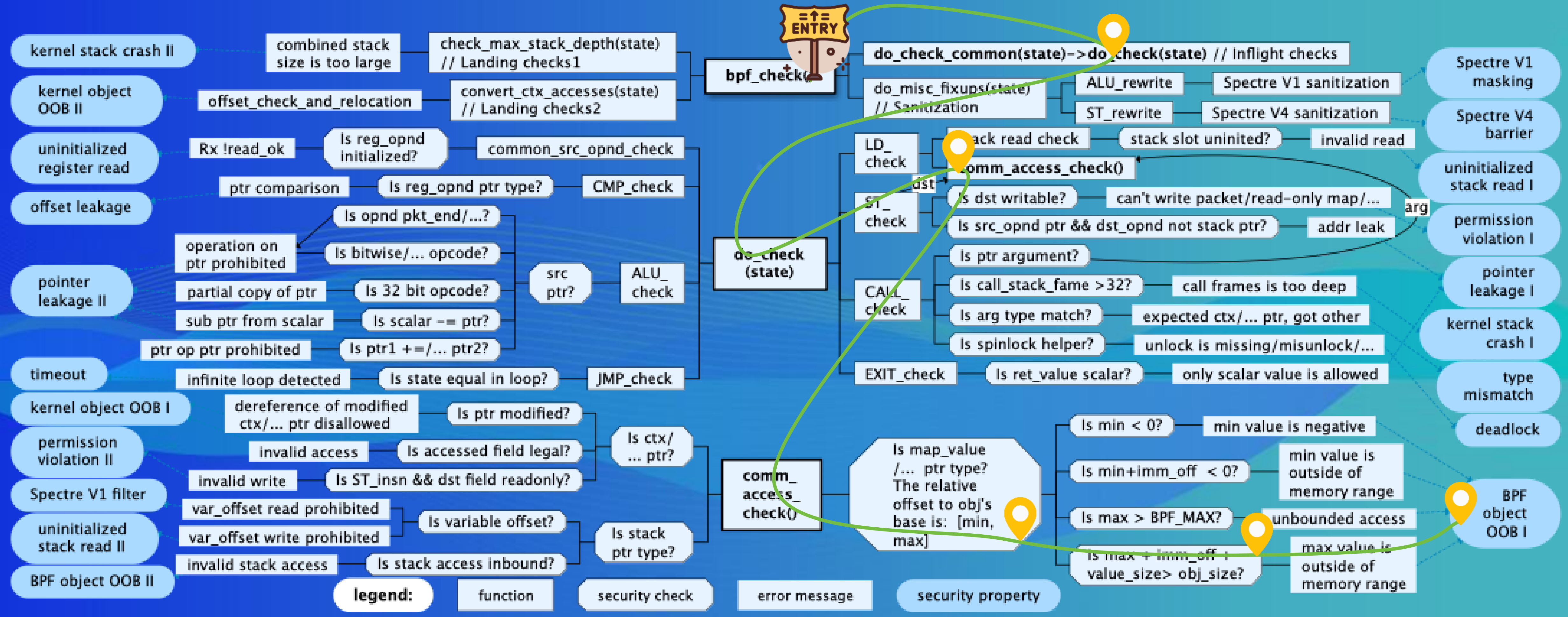
- What is eBPF?
- Verifier
- Conclusion
- High-level Design
- Evaluation
- Detailed Design

Outline


What is eBPF?
Verifier
Conclusion
High-level Design
Evaluation
Detailed Design

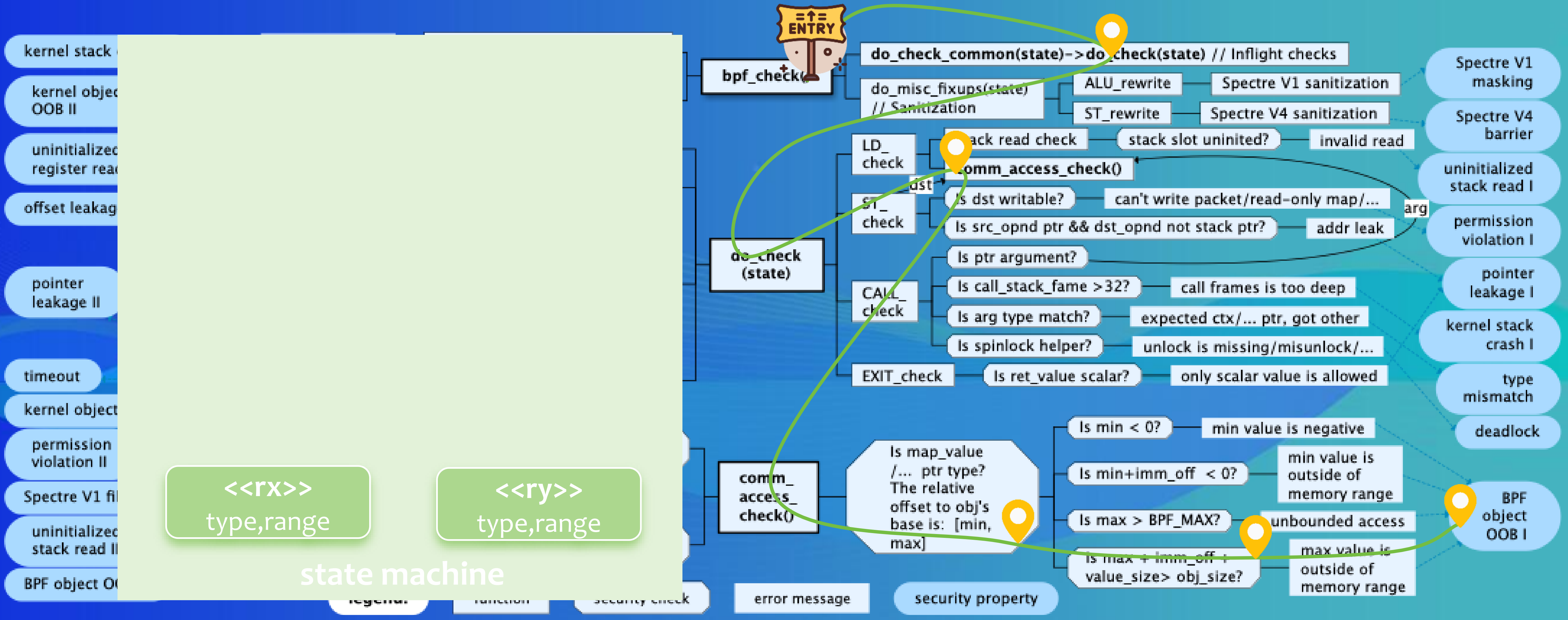
Security properties ensured in the verifier

We analyzed all (400+) security checks and summarized them into 20 security properties.



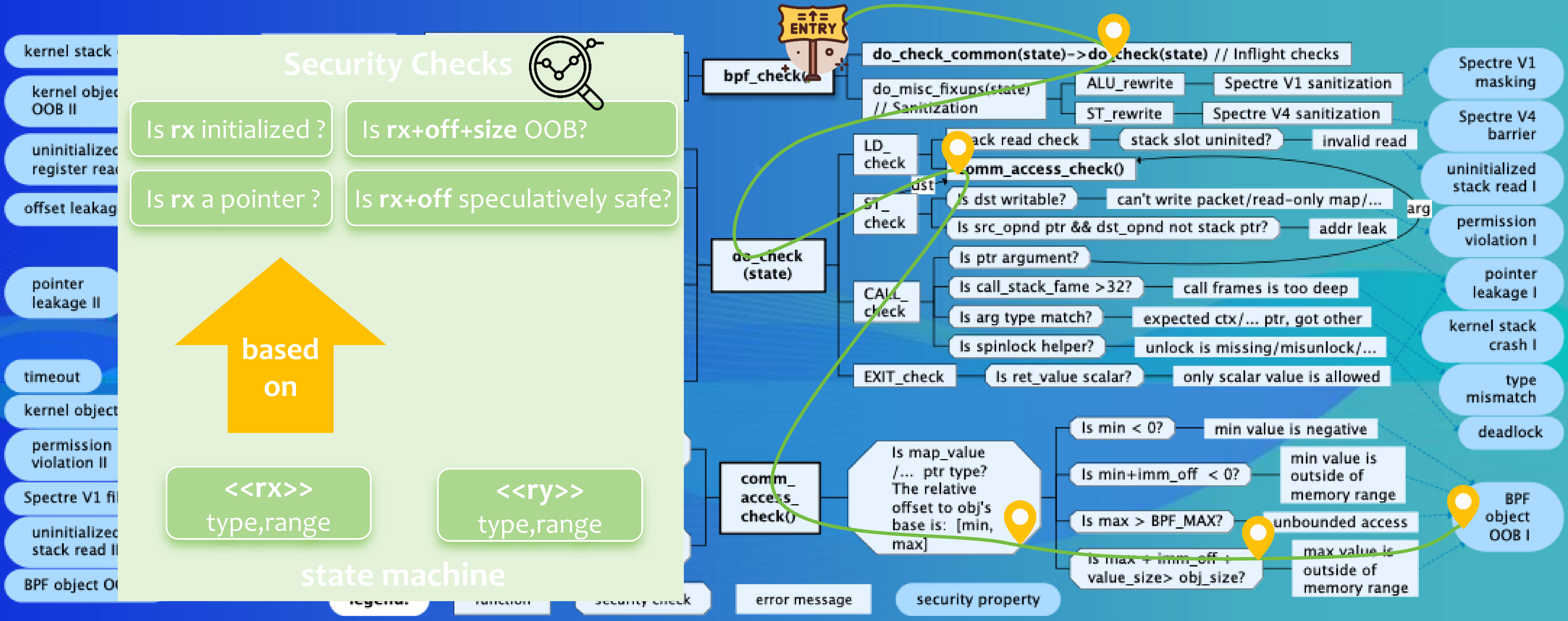
Security properties ensured in the verifier

We analyzed all (400+) security checks and summarized them into 20 security properties.



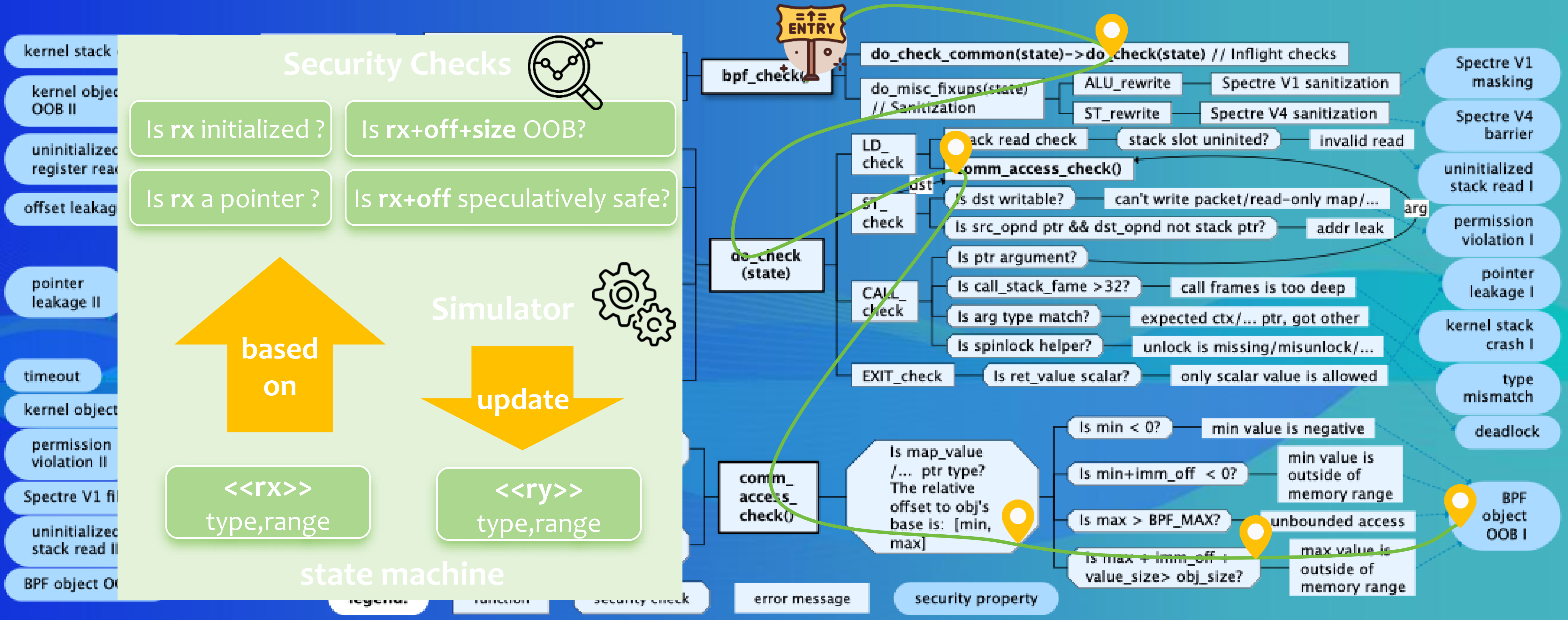
Security properties ensured in the verifier

We analyzed all (400+) security checks and summarized them into 20 security properties.



Security properties ensured in the verifier

We analyzed all (400+) security checks and summarized them into 20 security properties.



Security goals at design level

Integrity *Confidentiality* *Availability*

Three security goals: memory safety, information leakage prevention, and DoS prevention.

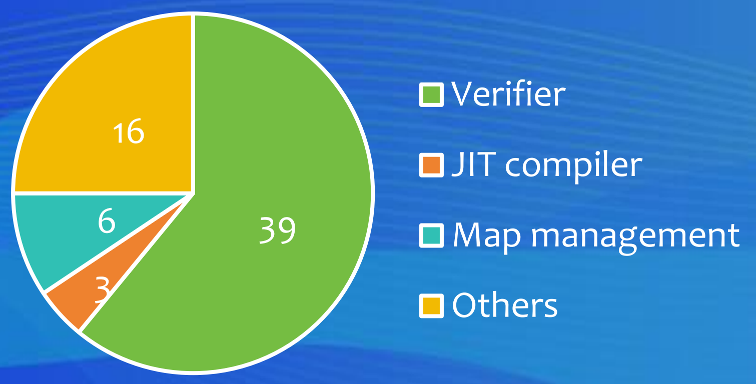
Security Goal	Description	Against Attacks	Corresponding Security Properties
SG-1: Memory Safety	Program can only access BPF memory, and specific kernel objects such as context.	OOB Access	BPF object OOB I/II, kernel object OOB I/II, permission violation I/II, *type mismatch*
SG-2: Information Leakage Prevention	Program cannot write pointers into maps, and calculation among pointers is not allowed.	Layout Leakage	pointer leakage I/II, offset leakage, *type mismatch*
	Program cannot read uninitialized information.	Uninitialized Read	uninit register read, uninit stack read I/II
	Program cannot speculatively access areas outside the BPF program's memory.	Spectre	Spectre V1 filter/masking, Spectre V4 barrier
SG-3: DoS Prevention	Program cannot execute for too long.	Denial-of-Service	time out, deadlock
	Program cannot crash while executing.	Crash Kernel	kernel stack crash I, kernel stack crash II

Dilemma of Static Analysis

The verification-based method has become the bottleneck of eBPF.

Correctness dilemma:
unsafe programs can pass the verification

Capability dilemma:
complex programs can not pass the verification



Verifier contributes the most of CVEs

State Explosion

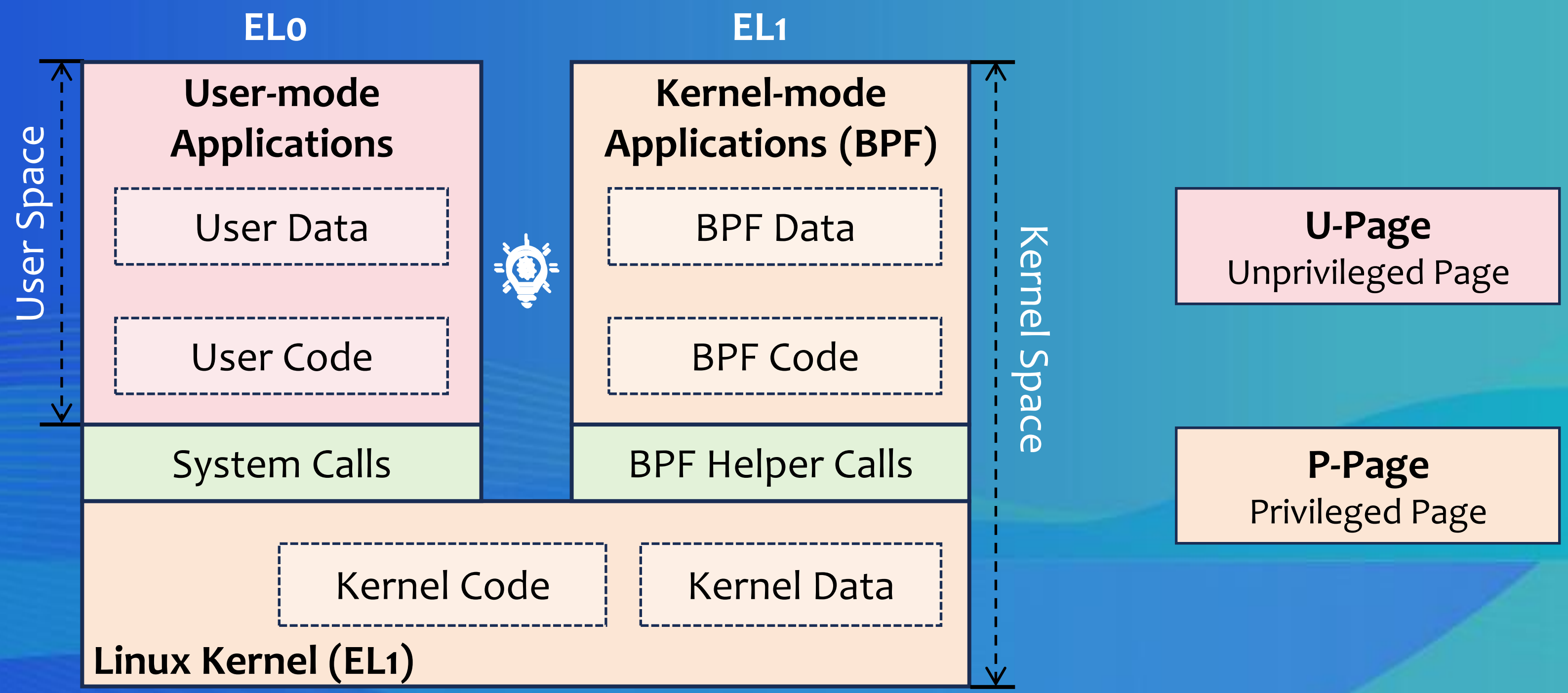
Outline

- What is eBPF?
- Verifier
- High-level Design
- Detailed Design
- Evaluation
- Conclusion

Outline

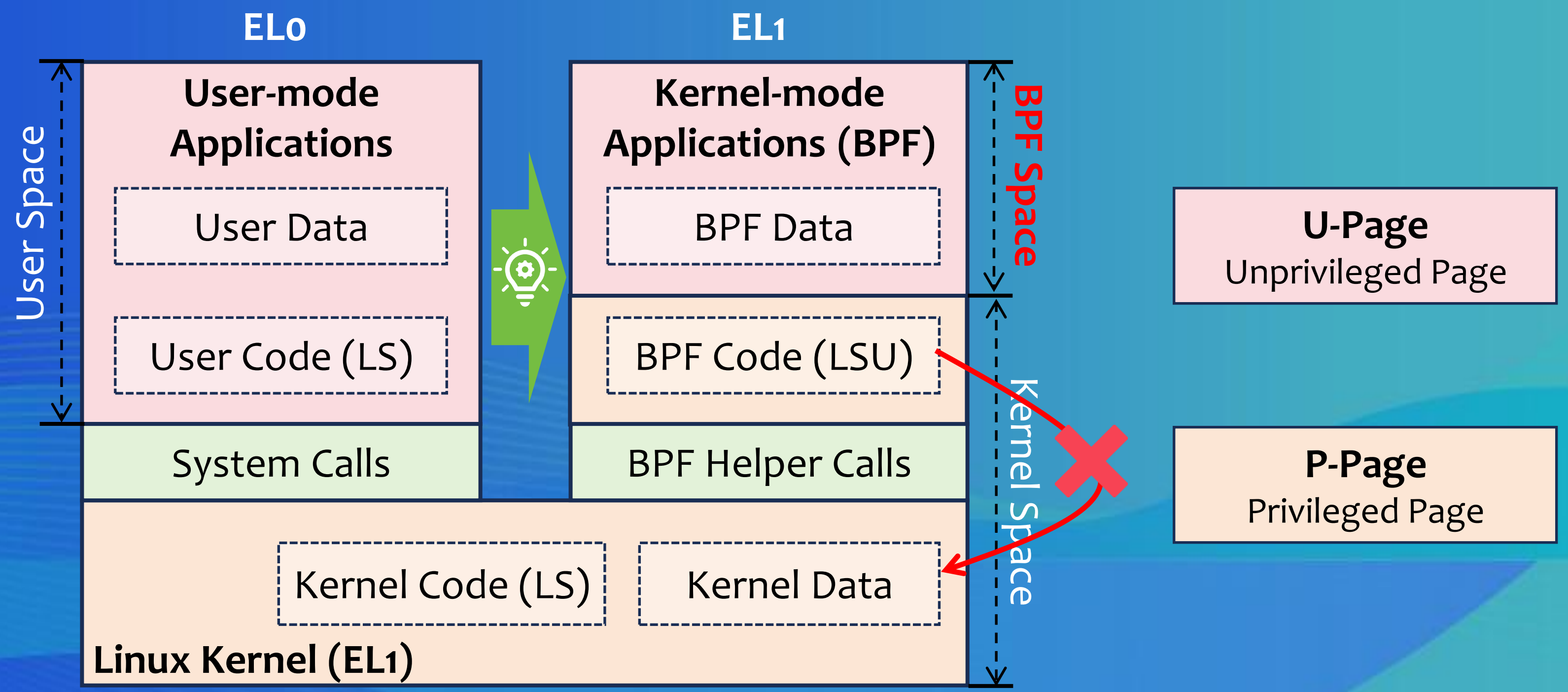
- What is eBPF?
- Verifier
- Conclusion
- High-level Design
- Evaluation
- Detailed Design

Our Key Idea: Build an isolation environment



Our Key Idea: Build an isolation environment -- HIVE

SG-1: EL-based memory isolation with LSU to de-privilege BPF programs,
SG-2: Independent BPF address space, and SG-3: Exception roll-back



* **Unprivileged load/store (LSU) instructions** are treated as at EL0, no matter which EL they are running on.

Challenges——BPF programs are highly coupled to Linux kernel

- ***BPF objects* require object-grained isolation.**
  - Metadata (e.g., pointers) is embedded in BPF objects and cannot be accessed.
  - EL-based memory isolation cannot provide such sub-page protection.
- ***Kernel objects* need to be accessed securely.**
  - BPF programs can directly access specific (discontinuous) fields of kernel objects.
  - EL-based memory isolation prevents such access and cannot provide such fine-grained protection.
- … …

Outline

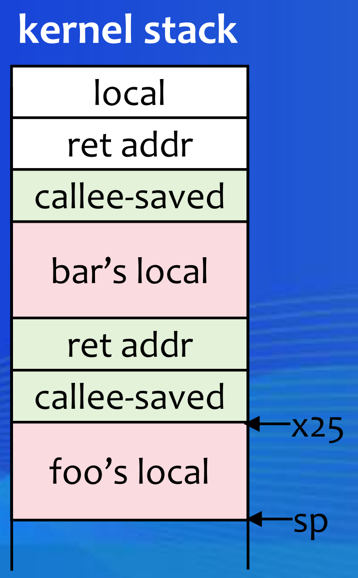
- What is eBPF?
- Verifier
- Conclusion
- High-level Design
- Evaluation
- Detailed Design

Outline

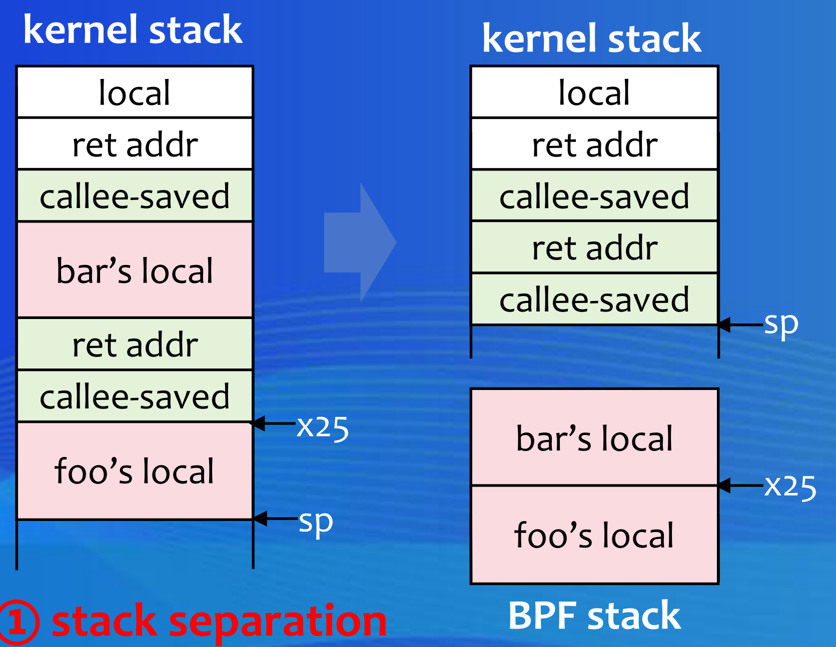

What is eBPF?
Verifier
Conclusion
High-level Design
Evaluation
Detailed Design

Handling BPF objects

BPF objects contain BPF-inaccessible metadata

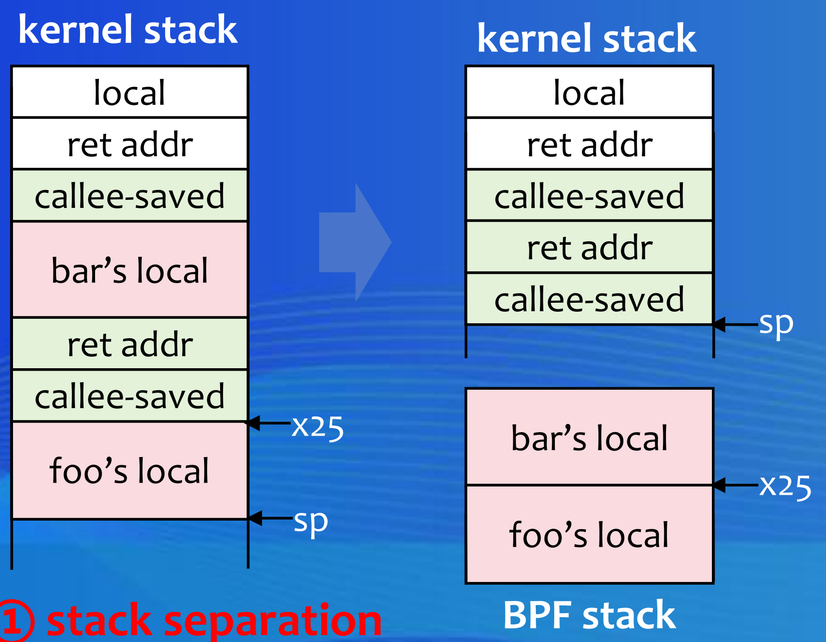


Handling BPF objects -- Compartmentalization

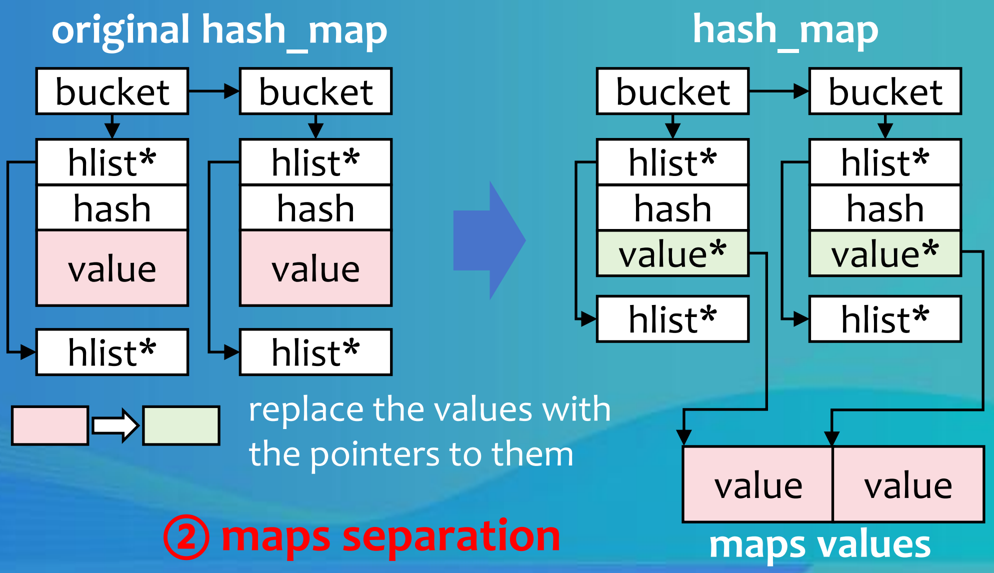


① stack separation

Handling BPF objects -- Compartmentalization



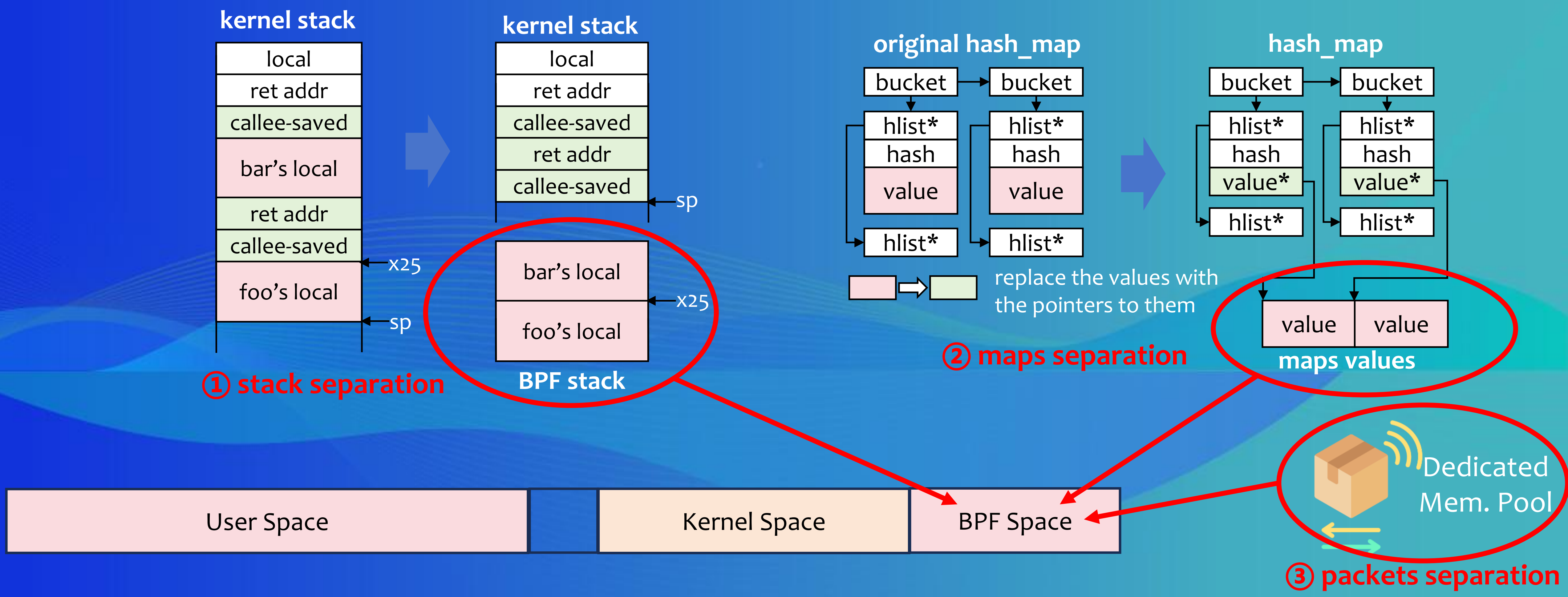
① stack separation



② maps separation

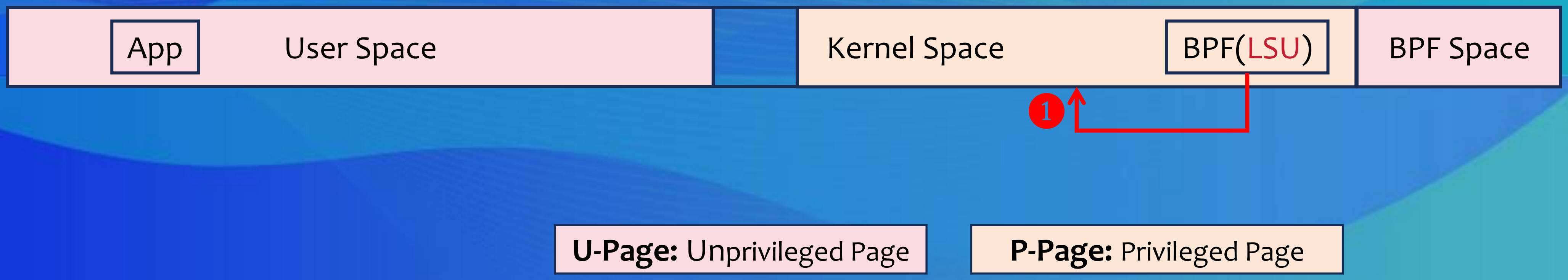
③ packets separation

Handling BPF objects -- Compartmentalization



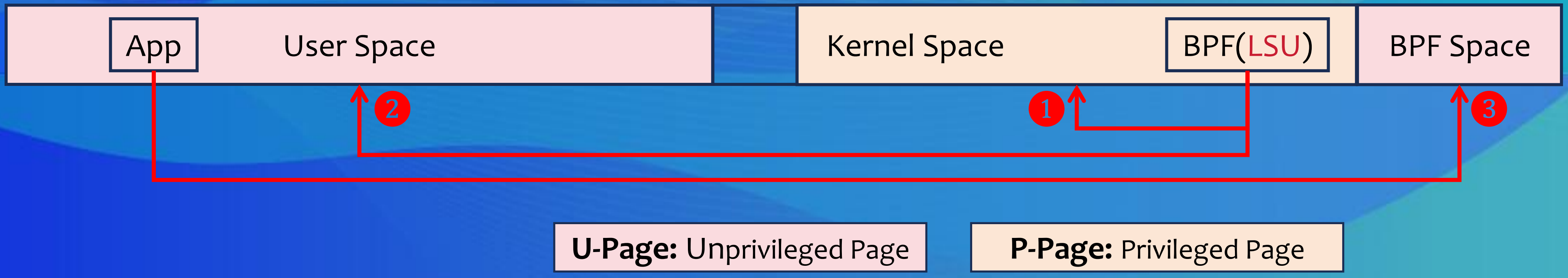
Direct Memory Access Isolation (SG-1)

1. **BPF program cannot access the kernel space.**
   - due to LSU cannot access P-pages



Direct Memory Access Isolation (SG-1)

1. **BPF program cannot access the kernel space.**
   - due to LSU cannot access P-pages
2. **BPF program cannot access the user space.**
   - E0PD0 forbids unprivileged access to lower half space
3. **User program cannot access the BPF space**
   - E0PD1 forbids unprivileged access to higher half space



Preventing Info. Leak (SG-2)

Independent address space (SG-2.1)
BPF space does not contain kernel layout information.

mov x_n, xzr

Use after initialization (SG-2.2)
BPF space is Initialized during BPF program loading. All BPF-used registers are cleared when helper returns.

Convert Spectre to Meltdown (SG-2.3)
The CSV3 patch forbids the speculatively loaded data with a permission fault to be used to form an address.

Secure and Passive DoS Prevention (SG-3)

Exceptions Capturing	Execution Timing
HIVE passively captures all triggered exceptions, rolls back the state to the entry point of the program, and unloads it.	HIVE maintains a timetable for each executing BPF program to track their execution time.

preventing kernel crash

preventing execution without terminating

Handling Kernel Pointers in BPF Program -- Our Insight

New solution for SG-1 and SG-2

1. These kernel pointers **cannot be modified.**
2. De-referenced points must be **exclusive.**
3. Accessing **privileged-pages.**

ARM Pointer Authentication (PA)
can ensure the pointer integrity.

\+

Regular Load/Store Instruction
can access the kernel space normally.

How do we identify memory access to kernel objects?
How do we prevent attacks against PA (e.g., replace, Spectre)?
How do we prevent signed pointers from being leaked?

Please read the paper if you are interested.

Outline

What is eBPF?

Verifier

High-level Design

Detailed Design

Evaluation

Conclusion

Outline

- What is eBPF?
- Verifier
- Conclusion
- High-level Design
- Evaluation
- Detailed Design

Security Evaluation

Real attacks against the security properties.

CVE ID	Root Cause	Target Property	Status[1]
2020-27194	Incorrect bound of `OR` insn.	dead loop	●
2021-3490	Incorrect 32-bit bound of bitwise.	BPF obj OOB	●
2021-31440	Incorrect bounds of 32-64 convert.	pointer leakage	●
2022-23222	Mischeck of `*_OR_NULL` Pointer.	kernel obj OOB	●
2020-8835	Incorrect 32-bit Bound.	kernel stack crash	●
2021-4204	Improper input validation.	offset leakage	●
2023-2163	Incorrect branch pruning.	type mismatch	●
2021-34866	Lack map pointer validation.	permission violation	●
2021-33624	Mispredicted branch speculation.	Spectre V1	❍

[1] ●: the attack is mitigated by HIVE, ❍: CVE is confirmed but lacks exploit.

Performance Evaluation

We selected 161 BPF programs from BCC and Tracee.

Table 7: The experimental results of real-world applications when running BPF programs with and w/o HIVE.

App.	config	baseline		eBPF-Tracee			eBPF-BCC			HIVE-Tracee			HIVE-BCC			HIVE/eBPF-O/H[4]		exe_cnt/req[5]	
		THRU[1]	%CPU[2]	THRU[1]	O/H[3]	%CPU[2]	THRU[1]	O/H[3]	%CPU[2]	THRU[1]	O/H[3]	%CPU[2]	THRU[1]	O/H[3]	%CPU[2]	Tracee	BCC	Tracee	BCC
Apache	**32KB**	18.50	98.6	10.48	76.6	98.4	6.17	199.9	99.1	10.11	82.9	98.6	6.03	206.9	99.1	3.48	2.28	555.1	568.8
	64KB	16.17	98.9	8.80	83.8	99.0	5.32	203.9	98.9	8.54	89.5	98.9	5.27	206.9	98.6	3.02	0.99	654.1	693.3
	128KB	12.52	99.0	6.65	88.3	99.0	3.60	248.1	99.1	6.42	95.0	99.4	3.46	262.2	98.4	3.46	3.90	809.6	1028.6
	256KB	7.70	99.6	4.41	74.6	98.5	2.01	282.2	98.1	4.26	80.8	98.5	2.01	282.8	98.1	3.44	0.16	1171.5	1749.5
	Geomean	-	-	-	80.6	-	-	231.1	-	-	86.9	-	-	237.4	-	**3.34**	**1.08**	**766.1**	**917.9**
Nginx	**32KB**	27.25	99.0	13.94	95.5	99.3	5.52	393.8	100.0	13.41	103.3	99.3	5.42	402.7	99.9	3.82	1.77	481.3	701.7
	64KB	23.96	99.0	12.34	94.1	99.5	4.48	434.8	99.9	11.86	102.1	99.8	4.40	444.8	99.8	3.95	1.83	584.6	823.9
	128KB	19.95	99.4	9.07	119.9	99.5	3.30	505.3	99.6	8.67	130.0	99.5	3.25	513.1	99.8	4.37	1.28	761.9	704.6
	256KB	12.98	93.4	5.85	121.8	99.5	2.26	474.9	98.0	5.58	132.5	99.0	2.19	492.5	99.5	4.60	2.97	1089.0	1912.4
	Geomean	-	-	-	107.1	-	-	450.2	-	-	116.1	-	-	461.2	-	**4.18**	**1.87**	**695.1**	**939.5**
Memc-ached	**32B**	1584.39	98.5	941.77	68.2	99.3	471.06	236.3	99.9	907.77	74.5	99.4	459.56	244.8	99.9	3.61	2.44	8595.7	13117.5
	64B	1583.11	98.6	939.88	68.4	99.3	467.08	238.9	99.9	906.88	74.6	99.4	458.95	244.9	99.8	3.51	1.74	8602.8	13110.0
	128B	1577.85	98.4	938.74	68.1	99.8	464.41	239.8	99.8	906.19	74.1	99.5	452.39	248.8	99.5	3.47	2.59	8647.7	13119.9
	256B	1551.61	98.6	923.09	68.1	99.5	461.82	236.0	99.6	883.12	75.7	99.3	455.12	240.9	99.6	4.33	1.45	8685.5	13115.6
	Geomean	-	-	-	68.2	-	-	237.7	-	-	74.7	-	-	244.8	-	**3.71**	**2.00**	**8632.9**	**13115.8**
Redis	**32B**	1342.35	88.7	861.30	55.9	90.0	698.98	92.0	66.7	836.33	60.5	81.0	689.23	94.8	67.9	2.90	1.39	975.9	1088.0
	64B	1304.76	100.0	861.96	51.4	81.7	663.63	96.6	65.7	836.59	56.0	82.0	659.54	97.8	64.6	2.94	0.62	1028.6	1399.3
	128B	1300.93	90.0	858.71	51.5	82.0	664.15	95.9	66.1	827.77	57.2	79.3	657.55	97.8	69.8	3.60	0.99	1020.9	1398.1
	256B	1292.59	90.0	855.05	51.2	90.0	656.88	96.8	70.0	821.67	57.3	80.0	652.03	98.2	68.0	3.90	0.74	1015.0	1408.2
	Geomean	-	-	-	52.4	-	-	95.3	-	-	57.7	-	-	97.2	-	**3.31**	**0.89**	**1009.9**	**1315.8**

[1] The application's throughput (thousands of requests per second). [2] The CPU utilization (%). [3] The overhead (%) of vanilla eBPF and HIVE compared to baseline which does not load BPF programs.
[4] The overhead (%) of HIVE compared to the vanilla eBPF, which is calculated using the throughput directly. [5] The average number of times BPF programs are executed per request.

Complexity Evaluation

The ultimate goal of eBPF is to “replace kernel modules as the de-facto means of extending the kernel”.

	BPF	HIVE		eBPF			KLEE				
Kernel Module	**#insn**	**exec time**	**load time**	**rejected cause**	**Ainsn**	**Astate**	**Ainsn**	**Astate**	**Icov**	**Bcov**	**exporing time**
polynomial	126	0.5μs	1.0ms	loop	1M	9K	10.5M	16.9K	99	75	4h 54min
crc-ccitt	134	0.1μs	1.1ms	loop	1M	9.5K	79.9K	2K	61	67	2min 27s
libarc4	265	8.1μs	1.7ms	loop	1M	34.5K	1.7M	21.5K	100	100	21h 25min
prime_numbers	378	0.6μs	2.4ms	branch	141K	1.9K	45.7M	23.9K	71	56	4h 54min
ghash	734	6.7μs	7.9ms	loop	1M	9.7K	21.5M	4.1K	50	55	17h 16min
sha3	1028	32.9μs	11.8ms	loop	1M	1.2K	158.5M	587	98	91	8h 3min
xxhash	1158	1.3μs	7.2ms	pointer ALU	38	1	26M	49.5K	40	39	7h 27min
libchacha	1421	4.4μs	2.9ms	loop	1M	2.6K	79.6M	131.1K	94	83	12h 6min
libsha256	1445	16.7μs	13.6ms	loop	1M	9.5K	50.6M	2.1K	91	85	12min 1s
des	1751	5.2μs	26.4ms	pointer ALU	39	1	7.4M	1K	100	95	1min 15s

Outline

- What is eBPF?
- Verifier
- High-level Design
- Detailed Design
- Evaluation
- Conclusion

Outline

- What is eBPF?
- Verifier
- High-level Design
- Detailed Design
- Evaluation
- Conclusion

Conclusion

- **Verification-based method has become the bottleneck of eBPF.**

- **We provide a hardware-backed isolation environment – Hive.**
  - De-priviledged and decoupled BPF.
  - Special design for accessing kernel objects.

- **Hive can provide the same security guarantees with low runtime overhead.**
- **Also addressed the capbility issue.**
  - Now BPF programs can be as complex as they want.

Thanks

patphzhang@tencent.com; https://patrickphzhang.github.io/
wangzhe12@ict.ac.cn

Isolation for the BPF Space

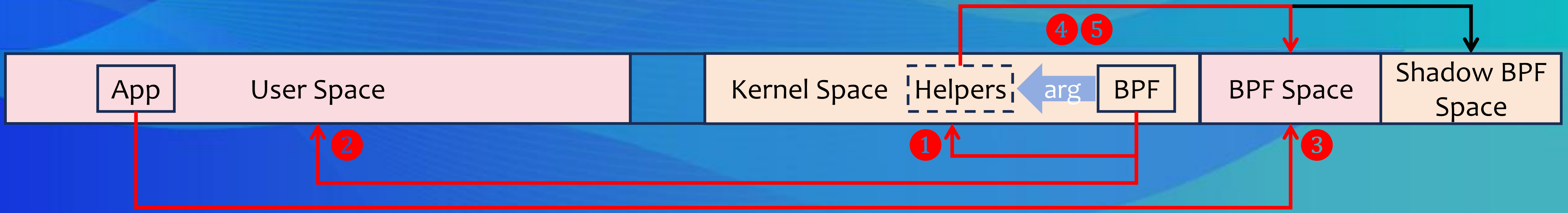
Isolation of direct memory access

1. **BPF program cannot access the kernel space.**
   - due to LSU cannot access P-pages
2. **BPF program cannot access the user space.**
   - E0PD0 forbids unprivileged access to lower half space
3. **User program cannot access the BPF space**
   - E0PD1 forbids unprivileged access to higher half space

Sanitization of helpers' parameters

4. **Helpers cannot be abused to access the kernel space.**
   - pointer parameters are masked when calling helpers
5. **Helpers can access unprivileged BPF space transparently.**
   - pointers are redirected to the shadow BPF space

Only need 1 instruction: *orr x_n, $mask_{1TB}$*



U-Page: Unprivileged Page

P-Page: Privileged Page

eBPF Pointer Types: *Inclusive* and *Exclusive* Types

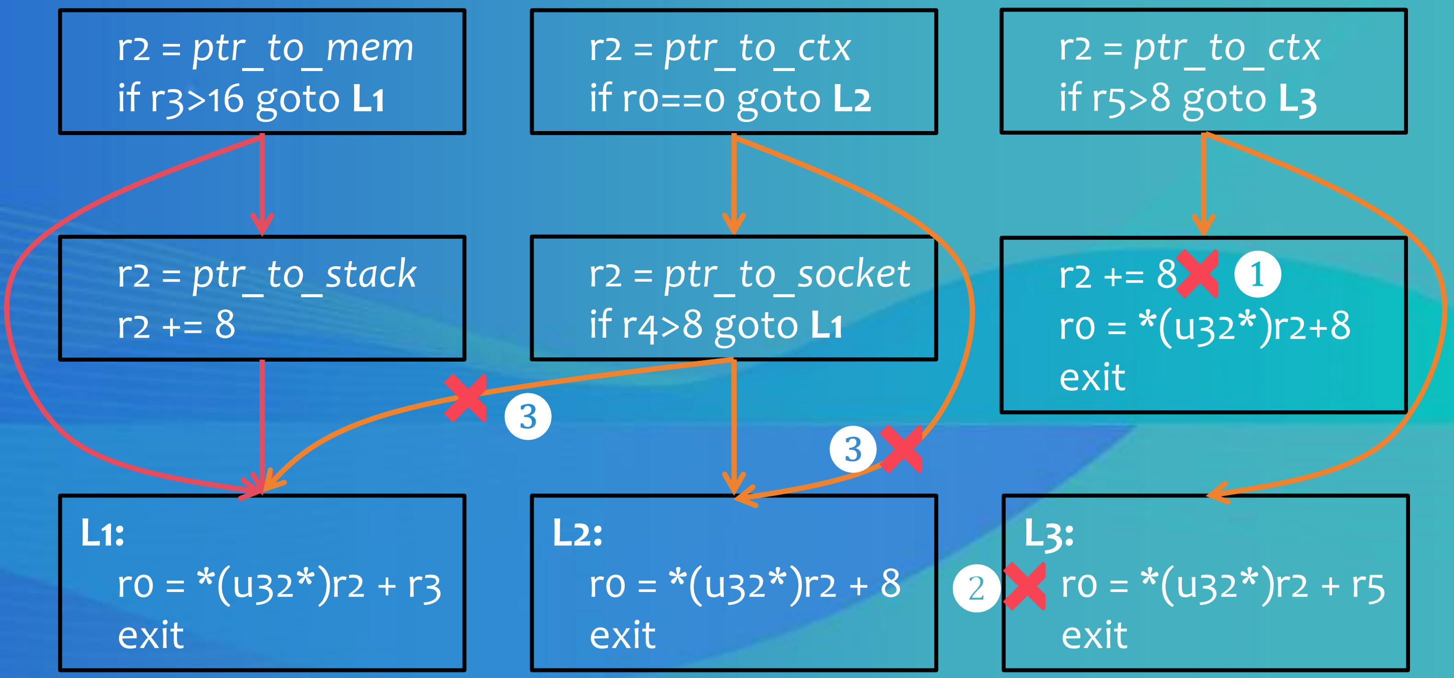
inclusive_type (10)

ptr_to_buf *ptr_to_stack* *ptr_to_packet*
ptr_to_mem *ptr_to_packet_meta*
ptr_to_tp_buffer *ptr_to_packet_end*
ptr_to_flow_keys *ptr_to_map_value*
ptr_to_map_key

exclusive_type (8)

ptr_to_tcp_sock *ptr_to_socket*
ptr_to_sock_common *ptr_to_xdp_sock*
ptr_to_ctx *ptr_to_btf_id*
ptr_to_map *ptr_to_func*

Types	Point to	Can be Modified	De-reference Access form	De-reference Pinned Loc.
Inclusive	BPF object	✓	Arbitrary form	✗
Exclusive	Kernel object	1 ✗	2 constant offset	3 ✓



eBPF Pointer Types: *Inclusive* and *Exclusive* Types

inclusive_type (10)

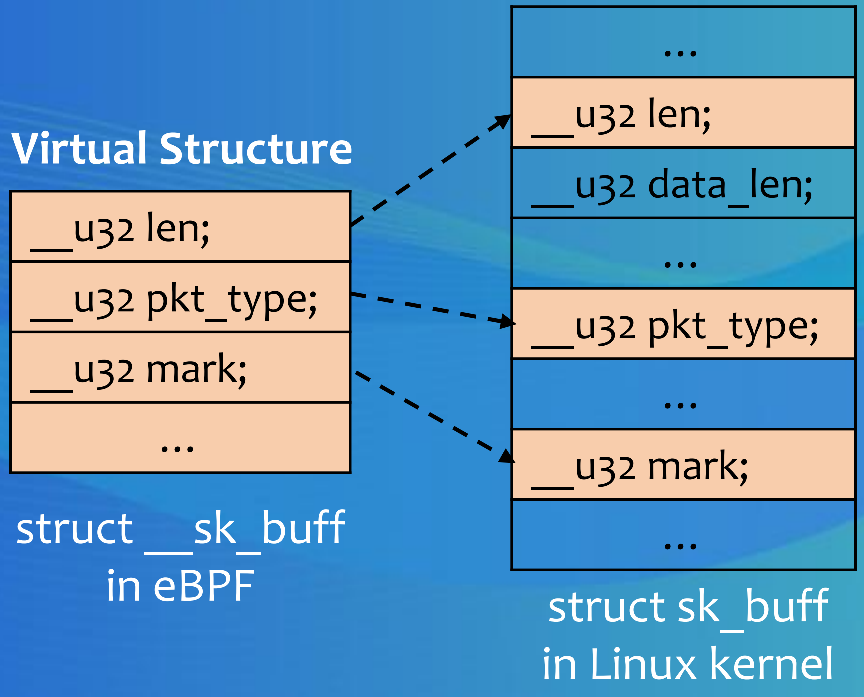
ptr_to_buf *ptr_to_stack* *ptr_to_packet*
ptr_to_mem *ptr_to_packet_meta*
ptr_to_tp_buffer *ptr_to_packet_end*
ptr_to_flow_keys *ptr_to_map_value*
ptr_to_map_key

exclusive_type (8)

ptr_to_tcp_sock *ptr_to_socket*
ptr_to_sock_common *ptr_to_xdp_sock*
ptr_to_ctx *ptr_to_btf_id*
ptr_to_map *ptr_to_func*

Types	Point to	Can be Modified	De-reference Access form	De-reference Pinned Loc.
Inclusive	BPF object	✓	Arbitrary form	✗
Exclusive	Kernel object	1 ✗	2 constant offset	3 ✓

Why the exclusive pointer type has so much constraints?



Used for RELOCATION

1. for accurate pointer tracking.
2. for relocating the offset.
3. for enforcing access control.

Handling Exclusive Pointer Types——Point-of-use Probing (SG-1)

context obj:

struct sk_buff

```
... ...
int len
... ...

... ...
```

BPF program

```
... ...
/* R1 stores ctx_ptr
   load sk_buff.len */
R2 = *(u32*)(R1 + 8)
... ...
```

JIT

Native Code

```
/* 1.sign context pointer at
   the program's entry */
/* 1.1 load exe_cnt to x24 */
mov x5, 8
mov x6, &exe_cnt
ldadd x5, x24, x6
/* 1.2 create the modifier */
orr x6, x24, ctx_type
/* 1.3 sign the ctx_ptr */
pacda x0, x6
... ...
/* 2. load sk_buff.len */
/* 2.1 placeholder instr */
nop
nop
nop
/* 2.2 load sk_buff.len */
ldtr w1, x0, 8  ## LSU
```

1

HIVE exception handler

2

```
/* 1. filter the exception */
if exception instr's opcode is not ldtr then
   goto label_illegal;
endif
/* 2. find the type and do security check */
foreach obj_type of exclusive_types do
   if auth(%x0, %x24 ∧ obj_type) pass then
     /* perform the security check */
     if access(base:%x5, offset:8, size:4) \
     is in whitelist(program_type, obj_type);
        then  patch the auth code; return;
     else goto label_illegal;
     endif
   endif
end foreach
/* 3. uninstall the bpf program */
label_illegal: uninstall_bpf(this_program);
```

3 4 5

Patch 6

generated code

```
mov x5, x0 ##not corrupt x0
orr x6, x24, ctx_type
autda x5, x6
ldr w1, x5, 8  ## regular LS
```

Security Method
Trust on the first access to kernel object

Create Unique Modifier
to avoid the pointer substitution attacks.

Trigger Permission Fault
when access the kernel space via LSU.

Patch Generated Code
to bind the access to the kernel object.

Check Legality
to lock the access
to the target object.