ePass: A Framework for Enhancing Flexibility and Runtime Safety of eBPF Programs

Yiming Xiang, Wanning He, Mehbubul Hasan Al-Quvi, Emmett Witchel, Ryan Huang

December 11, 2025

Overview

1. Introduction
2. Background
3. Design
   3.1 Core
   3.2 IR
   3.3 Pass
   3.4 Supervisor
   3.5 Sample Passes
4. Implementation
5. Evaluation
6. Conclusion

eBPF Introduction

Extended Berkeley Packet Filter (eBPF) is a kernel technology to extend the Linux kernel, and is widely used in big companies (Google, Meta, Microsoft, etc.).

eBPF Introduction

Extended Berkeley Packet Filter (eBPF) is a kernel technology to extend the Linux kernel, and is widely used in big companies (Google, Meta, Microsoft, etc.).

Benefits:

- Adding kernel features without modifying or recompiling the kernel
- Safe and efficient
- Wide use cases

eBPF Introduction

Extended Berkeley Packet Filter (eBPF) is a kernel technology to extend the Linux kernel, and is widely used in big companies (Google, Meta, Microsoft, etc.).

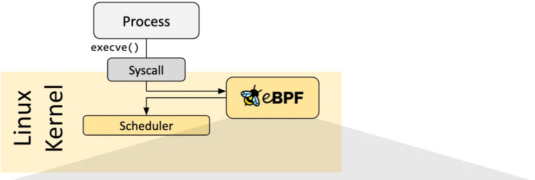
Benefits:

- Adding kernel features without modifying or recompiling the kernel
- Safe and efficient
- Wide use cases

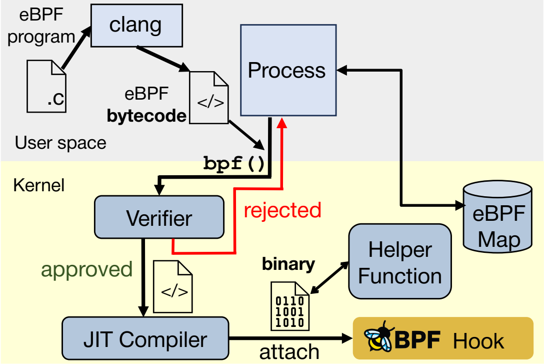
Use cases:

- Network packet filtering (where it originated)
- Observability
- Security
- Efficient networking
- Ongoing research: scheduling, storage, databases, distributed protocols...

eBPF Architecture



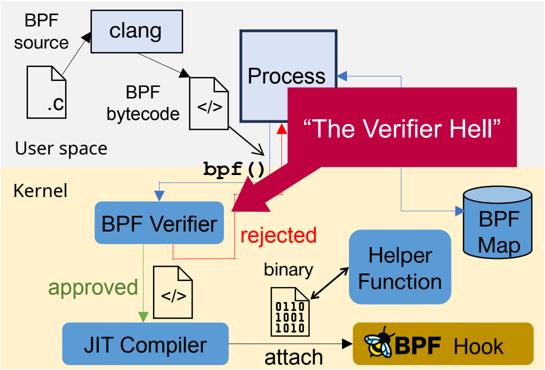

```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```



eBPF.io authors. "What is eBPF?"

The Problem: The Verifier Hell

The current Linux eBPF verifier uses the symbolic execution approach to exhaustively check the safety of eBPF programs against several rules.
This static-only approach causes **inflexibility** and **vulnerability**.



Limitation 1: Flexible Data Structure

- Advanced use cases (e.g., CPU schedulers, cache extensions) require richer data structures like **linked lists** and **trees**.
- But **dynamic allocation and pointer chasing** are restricted in eBPF.
- Recent systems (*e.g.*, KFlex, cache extension) work around this by **adding custom helpers/kfuncs** to support complex structures.

Limitation 2: False Rejections

Many researchers and developers have reported that the verifier is too strict and rejects many programs that are actually safe.

Stack overflow:

BPF verifier says program exceeds 1M instruction

Asked 2 years, 5 months ago Modified 2 years ago Viewed 2k times

3

For the following program I get an error from the verifier saying that it exceeds 1M instructions, even though it shouldn't. The program finds the hostname of a HTTP packet.

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
```

```
static __always_inline void bpf_barrier(void)
{
	/* Workaround to avoid verifier complaint:
	 * "dereference of modified ctx ptr R5 off=48+0, ctx+const is allowed,
	 *         ctx+const+const is not"
	 */
	barrier();
}
```

Real-world Applications (Cilium and Falco):

```
/* We need the unroll here otherwise the verifier complains about back-edges *.
#pragma unroll
	for(int i = 0; i < MAX_NUM_COMPONENTS; i++)
```

```
/* verifier workaround (dereference of modified ctx ptr) */
if (!revalidate_data_pull(ctx, &data, &data_end, &ip4))
	return DROP_INVALID;
```

```
#ifdef BPF_SUPPORTS_RAW_TRACEPOINTS
	call_filler(ctx, ctx, evt_type, drop_flags, socketcall_syscall_id);
#else
	/* Duplicated here to avoid verifier madness */
	struct sys_enter_args stack_ctx;
```

```
/* LLVM tends to generate code that verifier doesn't understand,
 * so force it the way we want it in order to open up a range
 * on the reg.
 */
asm volatile("r1 = *(u32 *)(%[ctx] +0)\n\t"
```

```
SEC("tp_btf/sched_process_exit")
int BPF_PROG(sched_proc_exit, struct task_struct *task)
{
	/* NOTE: this is a fixed-size event and so we should use the `ringbuf-approach`.
	 * Unfortunately we are hitting a sort of complexity limit in some kernel versions (<5.10)
	 * It seems like the verifier is not able to recognize the `ringbuf` pointer as a real pointer
```

Limitation 3: Dynamic security issues

There are security properties that are explicitly outside the verifier's domain:

- Helper functions/kfuncs
- JIT
- Dynamic tail calls

They may cause CVEs.

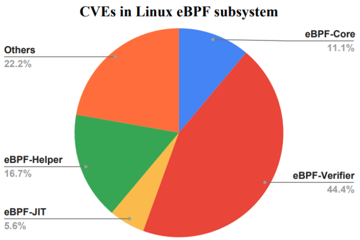
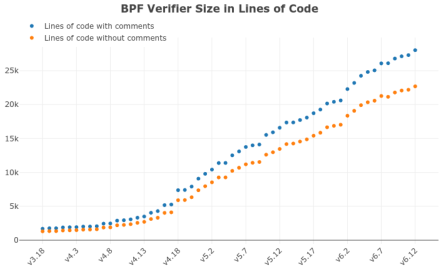


Figure 1: Categories of vulnerabilities in Linux eBPF subsystem.

Mohamed et al., "Understanding the Security of Linux eBPF Subsystem"

Limitation 3: Dynamic security issues

Moreover, the verifier can also incorrectly allow unsafe programs to run due to its analysis capability or bugs in its implementation.



Paul Chaignon, "Complexity of the BPF Verifier"

Vulnerabilities/Bugs	Total	Helper	Verifier
Arbitrary read/write	3	1	2
Deadlock/Hang	2	1	1
Integer overflow/underflow	2	2	0
Kernel pointer leak	5	0	5
Memory leak	2	0	2
Null-pointer dereference	7	6	1
Out-of-bound access	7	1	6
Reference count leak	1	1	0
Use-after-free	2	1	1
Misc	9	5	4
Total	40	18	22

Table 1: Bug statistics in eBPF helper functions and verifier in years of 2021 and 2022. Instead of searching CVEs (which are not embraced by the Linux community [28]), we searched commit logs for security-related bug fixes and confirmed them manually.

Jinghao et al., "Kernel extension verification is untenable"

Existing Approaches

- Improve static analysis

Existing Approaches

- Improve static analysis
  - Simple and precise static analysis of untrusted Linux kernel extensions. PLDI ’19.
  - Validating the eBPF Verifier via State Embedding. OSDI ’24.
  - VEP: A Two-stage Verification Toolchain for Full eBPF Programmability. NSDI ’25.
  - Prove It to the Kernel: Precise Extension Analysis via Proof-Guided Abstraction Refinement. SOSP ’25.

Existing Approaches

- Improve static analysis
  - Simple and precise static analysis of untrusted Linux kernel extensions. PLDI '19.
  - Validating the eBPF Verifier via State Embedding. OSDI '24.
  - VEP: A Two-stage Verification Toolchain for Full eBPF Programmability. NSDI '25.
  - Prove It to the Kernel: Precise Extension Analysis via Proof-Guided Abstraction Refinement. SOSP '25.
- Add runtime mechanisms
  - Fast, Flexible, and Practical Kernel Extensions. SOSP '24.
  - Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. eBPF '23.
  - SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. CCSW '24.

Existing Approaches

- Improve static analysis
  - Simple and precise static analysis of untrusted Linux kernel extensions. PLDI ’19.
  - Validating the eBPF Verifier via State Embedding. OSDI ’24.
  - VEP: A Two-stage Verification Toolchain for Full eBPF Programmability. NSDI ’25.
  - Prove It to the Kernel: Precise Extension Analysis via Proof-Guided Abstraction Refinement. SOSP ’25.
- Add runtime mechanisms
  - Fast, Flexible, and Practical Kernel Extensions. SOSP ’24.
  - Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. eBPF ’23.
  - SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. CCSW ’24.

They only focus on a specific use case, such as supporting dynamic loop and sandboxing the memory accesses.

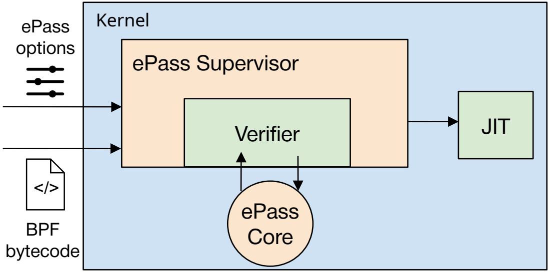
Existing Approaches

- Improve static analysis
  - Simple and precise static analysis of untrusted Linux kernel extensions. PLDI ’19.
  - Validating the eBPF Verifier via State Embedding. OSDI ’24.
  - VEP: A Two-stage Verification Toolchain for Full eBPF Programmability. NSDI ’25.
  - Prove It to the Kernel: Precise Extension Analysis via Proof-Guided Abstraction Refinement. SOSP ’25.
- Add runtime mechanisms
  - Fast, Flexible, and Practical Kernel Extensions. SOSP ’24.
  - Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. eBPF ’23.
  - SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. CCSW ’24.

They only focus on a specific use case, such as supporting dynamic loop and sandboxing the memory accesses.

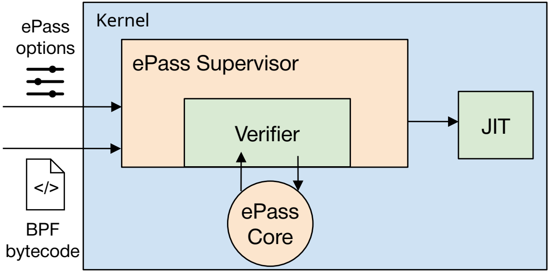
We need a more general and flexible approach to improve both flexibility and safety!

Design Overview



EPASS is implemented as an in-kernel framework that operates before JIT.

Design Overview



EPASS is implemented as an in-kernel framework that operates before JIT.

Design goals:

- **Flexible** Allow developers to conveniently analyze and manipulate the eBPF bytecode (*e.g.*, implementing a runtime check).
- **Safe** Enhance the safety while minimizing the attack surface.
- **Verifier-cooperative** EPASS can use the verifier's analysis results and the verifier can call EPASS.
- **High-performance** Minimize overhead (both static and runtime).

Safety Concerns

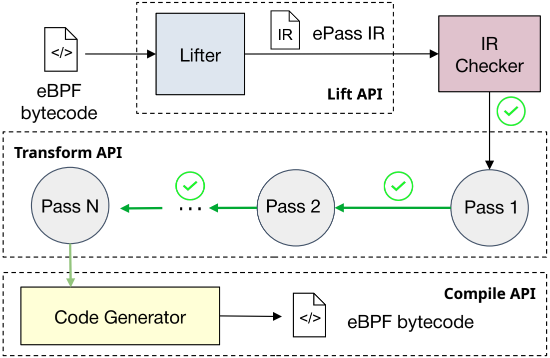
Does EPASS increase huge TCB?

Safety Concerns

Does ePass increase huge TCB?

- We do not modify the original eBPF components including the verifier and the JIT compiler. ePass only collects information from the verifier (*e.g.*, getting a range of a register).
- All the instrumented eBPF programs still go through the verifier to ensure eBPF safety properties.

Core Overview



A small compiler framework that provides functionalities to manipulate the eBPF bytecode and is decoupled from the kernel. EPASS Core first lifts the bytecode to an Intermediate Representation (IR) and transform it. Finally it compiles the IR back to eBPF bytecode.

Sample Workflow

1

```
r6 = 1000
call fun_5#5
r6 += -1
if r6 == 0 goto pc+1
goto pc-4
r0 = 2
exit
```

eBPF bytecode

Lift

2

```
b0:
b1:
  %0 = phi <b0 -> 1000> <b2 -> %2>
  %1 = call helper_func_5()
  %2 = add %0, -1
  jeq %2, 0, b2/b3
b2:
  ja b1
b3:
  ret 0x2
```

ePass IR

Transform

3

```
b0:
  %0 = alloc u32
  store %0, 0
b1:
  %1 = phi <b0 -> 1000> <b2 -> %3>
  %2 = call helper_func_5()
  %3 = add %1, -1
  jeq %3, 0, b2/b5
b2:
  %4 = load %0
  %5 = add %4, __BB_INSN_CNT__
  store %0, %5
  jgt %5, 1000000, b3/b4
b3:
  ja b1
b4:
  throw
b5:
  ret 0x2
```

ePass IR

Compile

4

```
r6 = 0
r7 = 1000
call fun_5#5
r7 += -1
if r7 == 0 goto pc+5
r6 += 5
if r6 > 1000000 goto pc+1
goto pc-6
r0 = 0
exit
r0 = 2
exit
```

eBPF bytecode

EPASS IR API Examples

Categories	API	Description
Instruction Manipulation	`ir_insn *epass_create_ja_insn(epass_env *env, ir_insn *pos_insn, ir_bb *to_bb, enum insert_position pos)` `void epass_ir_remove_insn(epass_env *env, ir_insn *insn)`	Create a direct jump (JA) instruction jumping to `to_bb` before or after (defined by `pos`) `pos_insn`. Remove an instruction from IR.
BB Manipulation	`ir_bb *epass_ir_split_bb(epass_env *env, ir_function *fun, ir_insn *insn, enum insert_position insert_pos)`	Split the BB into two BBs at a given position. This is often used to insert conditional jump instructions.
Verifier Integration	`ir_insn* epass_ir_find_ir_insn_by_rawpos(ir_function *fun, u32 rawpos)` `vi_entry *get_vi_entry(epass_env *env, u32 idx)`	Get the IR instruction that the the bytecode at `rawpos` corresponds to. Get the VI map entry from the instruction index `idx`.
Miscellaneous	`array epass_ir_get_operands(epass_env *env, ir_insn *insn)` `void epass_ir_replace_all_uses(epass_env *env, ir_insn *insn, ir_value rep)`	Get all the operands of a instruction. Replace all the uses of a VR with another value.

Table: Examples of EPASS IR APIs.

IR Overview

EPASS IR is a simple yet expressive abstraction of eBPF bytecode that facilitates program transformations.

It is inspired by the well-established designs of LLVM IR. Our IR provides several similar features that make analysis and optimization more convenient:

IR Overview

ePass IR is a simple yet expressive abstraction of eBPF bytecode that facilitates program transformations.

It is inspired by the well-established designs of LLVM IR. Our IR provides several similar features that make analysis and optimization more convenient:

- **SSA Form.** SSA simplifies optimization and analysis by providing a clear def-use chain.

IR Overview

EPASS IR is a simple yet expressive abstraction of eBPF bytecode that facilitates program transformations.
It is inspired by the well-established designs of LLVM IR. Our IR provides several similar features that make analysis and optimization more convenient:

- **SSA Form**. SSA simplifies optimization and analysis by providing a clear def-use chain.
- **Virtual Registers (VR)**. This abstraction frees developers from worrying about physical registers or low-level constraints.

IR Overview

EPASS IR is a simple yet expressive abstraction of eBPF bytecode that facilitates program transformations.
It is inspired by the well-established designs of LLVM IR. Our IR provides several similar features that make analysis and optimization more convenient:

- **SSA Form.** SSA simplifies optimization and analysis by providing a clear def-use chain.
- **Virtual Registers (VR).** This abstraction frees developers from worrying about physical registers or low-level constraints.
- **Explicit Control Flow Graph (CFG).** This block-based structure makes control-flow analysis straightforward and efficient.

IR Overview

EPASS IR is a simple yet expressive abstraction of eBPF bytecode that facilitates program transformations.

It is inspired by the well-established designs of LLVM IR. Our IR provides several similar features that make analysis and optimization more convenient:

- **SSA Form.** SSA simplifies optimization and analysis by providing a clear def-use chain.
- **Virtual Registers (VR).** This abstraction frees developers from worrying about physical registers or low-level constraints.
- **Explicit Control Flow Graph (CFG).** This block-based structure makes control-flow analysis straightforward and efficient.
- **Explicit Phi Instruction.** Phi instruction simplifies dataflow analysis and variable management.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **EPASS Instruction Set** Comprise three major categories:

ePass IR Design Goals

Though inspired from LLVM IR, ePass IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **ePass Instruction Set** Comprise three major categories:
  - Instructions that directly correspond to the eBPF instructions, constructed by the lifter.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **EPASS Instruction Set** Comprise three major categories:
  - Instructions that directly correspond to the eBPF instructions, constructed by the lifter.
  - General instructions used during transformations, such as `alloc`, `store`, `getelemptr`.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **EPASS Instruction Set** Comprise three major categories:
  - Instructions that directly correspond to the eBPF instructions, constructed by the lifter.
  - General instructions used during transformations, such as `alloc, store, getelemptr`.
  - eBPF-specific instructions not in the eBPF ISA. *e.g.*, the `throw` instruction that terminates the program safely, and `eCall`: a virtual instruction to support complex operations in eBPF.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **EPASS Instruction Set** Comprise three major categories:
  - Instructions that directly correspond to the eBPF instructions, constructed by the lifter.
  - General instructions used during transformations, such as `alloc, store, getelemptr`.
  - eBPF-specific instructions not in the eBPF ISA. *e.g.*, the `throw` instruction that terminates the program safely, and `eCall`: a virtual instruction to support complex operations in eBPF.
- **Untyped virtual registers** Currently do not have a full-featured type system, but differentiates value sizes for correct compilation, accompanied with an IR checker.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **EPASS Instruction Set** Comprise three major categories:
  - Instructions that directly correspond to the eBPF instructions, constructed by the lifter.
  - General instructions used during transformations, such as `alloc, store, getelemptr`.
  - eBPF-specific instructions not in the eBPF ISA. *e.g.*, the `throw` instruction that terminates the program safely, and `eCall`: a virtual instruction to support complex operations in eBPF.
- **Untyped virtual registers** Currently do not have a full-featured type system, but differentiates value sizes for correct compilation, accompanied with an IR checker.
- **Raw memory load and store** The memory load and store instructions in eBPF ISA are mapped to two IR instructions–`loadraw` and `storeraw`. They are separated from the EPASS `load` and `store` instructions to preserve the stack layout.

EPASS IR Design Goals

Though inspired from LLVM IR, EPASS IR is designed to be more lightweight and specifically tailored to the eBPF bytecode and our design goals.

- **EPASS Instruction Set** Comprise three major categories:
  - Instructions that directly correspond to the eBPF instructions, constructed by the lifter.
  - General instructions used during transformations, such as `alloc, store, getelemptr`.
  - eBPF-specific instructions not in the eBPF ISA. *e.g.*, the `throw` instruction that terminates the program safely, and `eCall`: a virtual instruction to support complex operations in eBPF.
- **Untyped virtual registers** Currently do not have a full-featured type system, but differentiates value sizes for correct compilation, accompanied with an IR checker.
- **Raw memory load and store** The memory load and store instructions in eBPF ISA are mapped to two IR instructions–`loadraw` and `storeraw`. They are separated from the EPASS `load` and `store` instructions to preserve the stack layout.
- **Annotated IR** The IR is *annotated* with eBPF bytecode information during lifting.

Annotated IR Example

Log from a sample program:

```
b0:
b1:
  %0 = phi <b0 -> 0x1f4(32)[0.imm]> <b2 -> %2[2.insn]>
  %1 = call __built_in_func_5() // [1.insn]
  %2 = add(64) %0[2.dst], 0xffffffff(32)[2.imm],  // [2.insn]
  jeq(64) %2[3.dst], 0x0(32)[3.imm], b2/b3 // [3.insn]
b2:
  ja b1 // [4.insn]
b3:
  ret 0x2(32)[5.imm] // [6.insn]
```

EPASS Instruction Examples

Syntax	Description
VR = `alloc` T	Allocate space on stack. Code generator may optimize it to registers.
`store` T V_1 V_2	Store value V_2 to V_1 where V_1 should be an alloc instruction.
VR = `loadraw` T AV	Load a raw memory data of size T at address AV.
VR = `add` V_1 V_2	Add V_1 and V_2.
`jeq` V_1 V_2 B_1 B_2	Jump to B_2 if $V_1 = V_2$, to B_1 if not.
VR = `phi` $<B_i \rightarrow V_i> \cdots$	The Phi (φ) instruction. The runtime value of VR is V_i if coming from B_i.
`throw`	Terminate the program safely.
VR = `allocarray` T s	Allocate an array of length s and size of each element T on stack.
VR = `getelemptr` V_1 V_2	Get the pointer to element $V_2[V_1]$ where V_2 is a pointer to an array.

Table: EPASS instruction examples. *VR* is a virtual register, *V* is a value which may be either a VR or a constant, *T* is a size type (*e.g.*, 8B), *AV* is an address value which represents a value with a constant offset, *s* is a number, and *B* is a basic block.

Pass Design

A pass can manipulate the IR for a variety of purposes, such as optimizing code or inserting runtime checks.
We design it to be **flexible** and **highly configurable**.

Pass Design

A pass can manipulate the IR for a variety of purposes, such as optimizing code or inserting runtime checks.

We design it to be **flexible** and **highly configurable**.

- Each pass may have its own parameters.

Pass Design

A pass can manipulate the IR for a variety of purposes, such as optimizing code or inserting runtime checks.

We design it to be **flexible** and **highly configurable**.

- Each pass may have its own parameters.
- In-Verifier (IV) and Post-Verifier (PV) passes.

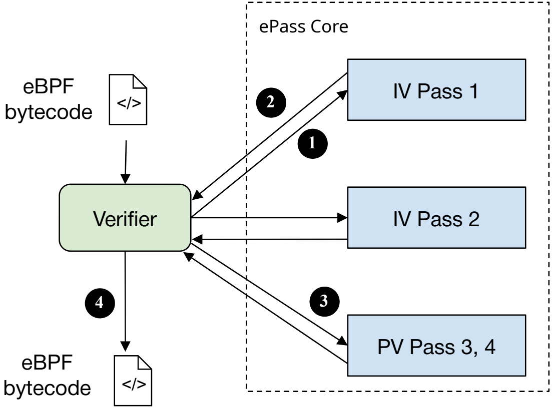
Pass Design

A pass can manipulate the IR for a variety of purposes, such as optimizing code or inserting runtime checks.

We design it to be **flexible** and **highly configurable**.

- Each pass may have its own parameters.
- In-Verifier (IV) and Post-Verifier (PV) passes.
- IR checker. The pass runner invokes the IR checker before and after running each pass to ensure the transformed IRs are valid.

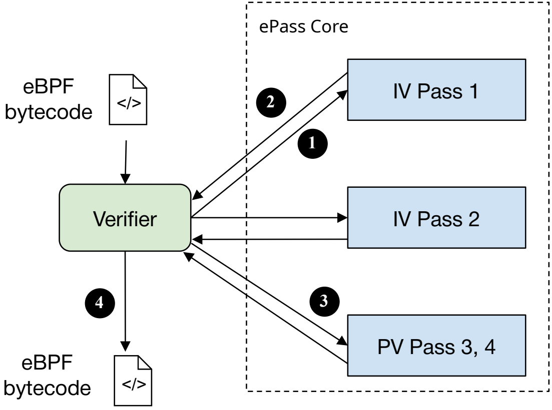
In-Verifier (IV) and Post-Verifier (PV) Passes



Passes with different purposes require different ways, stages to run.

- Addressing verifier error passes: probably need to run multiple times.
- Optimization and some runtime checks: only need to run once at last.

In-Verifier (IV) and Post-Verifier (PV) Passes



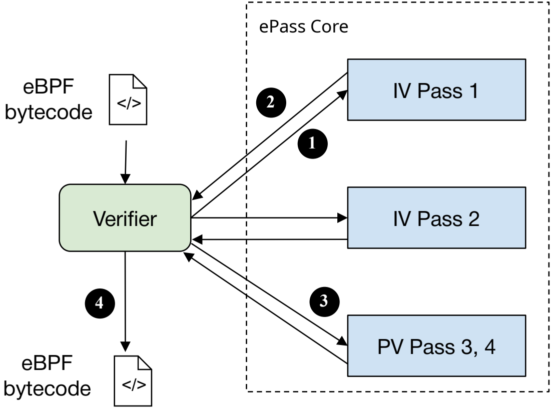
Passes with different purposes require different ways, stages to run.

- Addressing verifier error passes: probably need to run multiple times.
- Optimization and some runtime checks: only need to run once at last.

We categorize all passes to two types:

- **In-verifier (IV) Pass** Address verifier-reported errors during the static verification. Usually bound to a specific verifier error.
- **Post-verifier (PV) Pass** Executed only once after the static verification and serve broader purposes.

EPASS Supervisor



The supervisor will build an execution strategy to run all the passes.

Verifier Information (VI)

PV passes run after the verification, and lack direct access to the verifier's intermediate state.

Some PV passes require verifier analysis results to perform checks.

To solve this problem, we provide *Verifier Information map (VI map)*.

Verifier Information (VI)

PV passes run after the verification, and lack direct access to the verifier's intermediate state.

Some PV passes require verifier analysis results to perform checks.

To solve this problem, we provide *Verifier Information map (VI map)*.

VI Map

A mapping that maps instruction indices (raw positions) to their corresponding analysis results.

Verifier Information (VI)

PV passes run after the verification, and lack direct access to the verifier's intermediate state.
Some PV passes require verifier analysis results to perform checks.
To solve this problem, we provide *Verifier Information map (VI map)*.

VI Map

A mapping that maps instruction indices (raw positions) to their corresponding analysis results.

With the function `epass_ir_find_ir_insn_by_rawpos`, we could obtain the verifier information for all VRs in the IR.

Verifier Information (VI)

PV passes run after the verification, and lack direct access to the verifier's intermediate state.
Some PV passes require verifier analysis results to perform checks.
To solve this problem, we provide *Verifier Information map (VI map)*.

VI Map

A mapping that maps instruction indices (raw positions) to their corresponding analysis results.

With the function `epass_ir_find_ir_insn_by_rawpos`, we could obtain the verifier information for all VRs in the IR.
Developers could add any verifier data to this map on their demand.

Builtin Passes Overview

We have implemented 12 major passes covering different purposes:

- Flexibility: 6
- Security enhancement: 3
- Optimization: 3

Heap Pass

- To introduce advanced data structures, we need a mechanism to allocate memory dynamically.

Heap Pass

- To introduce advanced data structures, we need a mechanism to allocate memory dynamically.
- We emulate heap memory by using a eBPF map as the backing store.

Heap Pass

- To introduce advanced data structures, we need a mechanism to allocate memory dynamically.
- We emulate heap memory by using a eBPF map as the backing store.
- However, pointer chasing is banned by the verifier.

Heap Pass

- To introduce advanced data structures, we need a mechanism to allocate memory dynamically.
- We emulate heap memory by using a eBPF map as the backing store.
- However, pointer chasing is banned by the verifier.
- EPASS could transform `storeraw` and `loadraw` to use the fake heap!

Heap Pass

- To introduce advanced data structures, we need a mechanism to allocate memory dynamically.
- We emulate heap memory by using a eBPF map as the backing store.
- However, pointer chasing is banned by the verifier.
- ePass could transform `storeraw` and `loadraw` to use the fake heap!
- Need to provide eCall instructions `malloc` and `free` doing the real allocation and deallocation.

Heap Pass

- To introduce advanced data structures, we need a mechanism to allocate memory dynamically.
- We emulate heap memory by using a eBPF map as the backing store.
- However, pointer chasing is banned by the verifier.
- EPASS could transform `storeraw` and `loadraw` to use the fake heap!
- Need to provide eCall instructions `malloc` and `free` doing the real allocation and deallocation.

Our heap pass supports:

- Default fixed-size block based allocator. (Heap is for a single program, and in many cases users can predict the size of the block to avoid fragmentation.)
- Allow users to define their own allocators in the eBPF program and only uses the heap pass to transform memory accesses (to resolve the pointer chasing issue).

Heap Pass

epass_helper.h

```
struct … meta SEC(".maps");
struct … data SEC(".maps");
__noinline void* malloc(__u64 size){…
asm volatile(".byte 0x85, 0x62, 0,0,0,0,0,0\n"
: : "r"(r1), "r"(r2) :);}
__noinline void free(void* ptr){...}
```

```
#include "epass_helper.h"
…
struct ll {
  __u64 a;
  struct test_struct *next;
};

…
struct ll *a = malloc(sizeof(struct ll));
struct ll *b = malloc(sizeof(struct ll));
a->next = b
…
free(a);
free(b);
```

eBPF C Program

```
[0] (b7) r1 = 16
[1] (85) call efun#0 (#arg: 1)
[2] (bf) r2 = r0
[3] (85) call efun#0 (#arg: 1)
[4] (7b) *(u64 *)(r2 +8) = r0
…
```

eBPF Bytecode

ePass IR

```
…
 %4 = ecall efun#0(0x10(32)[0.imm]) // [1.insn]
 %5 = ecall efun#0(0x10(32)[0.imm]) // [3.insn]
 storeraw u64 %4[1.dst]+8 %5[14.src] // [4.insn]
…
```

Heap Pass

```
 storeraw u32 R10[1.dst]+-12(+off) 0x0(32)[1.src] // [1.insn]
 %0 = add(64) R10[3.dst], 0xfffffff4(32)[3.imm],  // [3.insn]
 %1 = loadimm(imm64) 0x0 // [4.insn]
 %2 = call __built_in_func_1(%1, %0[3.insn]) // [6.insn]
…
 %7 = … (malloc implementation, omit here)
 %8 = …
 if %7 < 0 or %7 >= 0xfef goto error
 %9 = add(64) %2, %7,
 storeraw u64 %9+8 %8[14.src] // [4.insn]
…
```

(Simplified) ePass IR

Heap Pass Demo

Loop Counter Pass

Many developers have encountered the verifier's instruction limit rejection. The verifier's static limit calculation is very inaccurate as it only counts how many instructions it processed. Developers need to manually use `bpf_for` or `bpf_repeat` to express loops, which makes the code less readable and less maintainable.

Loop Counter Pass

Many developers have encountered the verifier's instruction limit rejection. The verifier's static limit calculation is very inaccurate as it only counts how many instructions it processed. Developers need to manually use `bpf_for` or `bpf_repeat` to express loops, which makes the code less readable and less maintainable.
We implement a loop counter IV pass that adds a runtime instruction counter to the loops causing verifier rejection, and derive the best bound for each loop from the verifier's static analysis results.

Helper Validation Pass

Verifier has no way to ensure that the helper functions are correct.
This PV pass will add runtime checks for the return value of helper functions to make sure that they fall within the verifier's expected range.

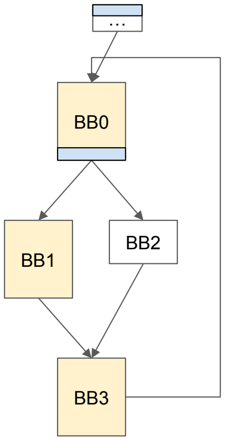
Taillcall Instruction Counter Pass

Tailcall is not tracked by the verifier for instruction limit checking. One can construct a long chain of tailcalls to bypass the original limit to do DoS attacks. Implementation bug may also lead to infinite tailcalls (CVE-2024-47794).

Taillcall Instruction Counter Pass

Tailcall is not tracked by the verifier for instruction limit checking. One can construct a long chain of tailcalls to bypass the original limit to do DoS attacks. Implementation bug may also lead to infinite tailcalls (CVE-2024-47794). We deign:

- A PV pass that inserts a dynamic instruction counter to track executed instructions.
- The counter is stored in a per-CPU map and passed across tail calls.
- Each program reads and accumulates the counter at entry.
- Execution stops once the total counter exceeds 1M.



Instruction counter pass

MSan Pass

- Verifier does not track uninitialized stack memory in default mode.

MSan Pass

- Verifier does not track uninitialized stack memory in default mode.
- In non-root mode, verifier tracks uninitialized stack memory inaccurately. It only tracks the deepest used offset and assumes all bytes above it are initialized.

MSan Pass

- Verifier does not track uninitialized stack memory in default mode.
- In non-root mode, verifier tracks uninitialized stack memory inaccurately. It only tracks the deepest used offset and assumes all bytes above it are initialized.
- A eBPF version of the code snippet in the Clang MemorySanitizer examples can access uninitialized stack memory. This is a typical memory access bug in real-world programs.

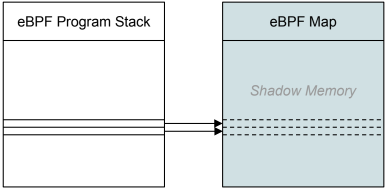
```c
int a[10];
a[9] = 0;
if (a[x]) // x is a valid index
  bpf_printk("%d", a[x]);
```

Uninitialized memory example

MSan Pass

A PV pass.
Uses the *Shadow Memory* technique.
MSan pass will:

- Allocate shadow memory in a eBPF map with the same size of the original stack (512 bytes).
- Add instructions after all `storeraw` and before all `loadraw` instructions to set/query the shadow byte (1=initialized, 0=uninitialized).



Masking Pass

There is a gap between the LLVM compiler and the verifier. The LLVM compiler can generate programs that the verifier rejects.

```
int id = bpf_ktime_get_ns() % 10;
static int arr[10];
for (int i = 0; i < 10; ++i)
  arr[i] = i;
bpf_printk("%d", arr[id]);
```

Masking Pass

There is a gap between the LLVM compiler and the verifier. The LLVM compiler can generate programs that the verifier rejects.

```c
int id = bpf_ktime_get_ns() % 10;
static int arr[10];
for (int i = 0; i < 10; ++i)
  arr[i] = i;
bpf_printk("%d", arr[id]);
```

```c
int id = bpf_ktime_get_ns() % 10;
static int arr[10];
for (int i = 0; i < 10; ++i)
  arr[i] = i;
if (id >= 10 || id < 0)
  return 0;
bpf_printk("%d", arr[id]);
```

Masking Pass

There is a gap between the LLVM compiler and the verifier. The LLVM compiler can generate programs that the verifier rejects.

```c
int id = bpf_ktime_get_ns() % 10;
static int arr[10];
for (int i = 0; i < 10; ++i)
  arr[i] = i;
bpf_printk("%d", arr[id]);
```

```c
int id = bpf_ktime_get_ns() % 10;
static int arr[10];
for (int i = 0; i < 10; ++i)
  arr[i] = i;
if (id >= 10 || id < 0)
  return 0;
bpf_printk("%d", arr[id]);
```

All Rejected!

Developers need to manually insert an `AND` instruction to mask the index to the array size. However, this workaround is strange and makes the code less readable.

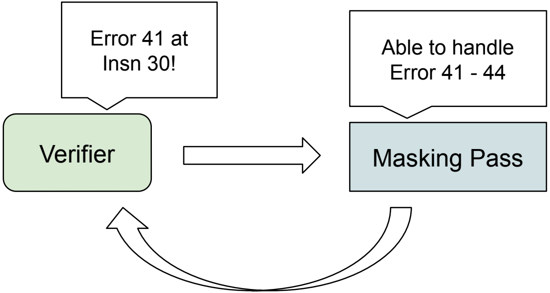
Masking Pass

An IV pass.

If the verifier reports an error that the masking pass could possibly resolve, it will run.

Masking pass will:

- Find why and where the verifier thinks the access is out of bound.
- Insert a boundary check at the appropriate position.



Runtime Validation Pass: Exploits due to verifier bugs

A common exploit pattern involves passing invalid arguments to a helper function.

[1] Full exploit code: https://github.com/tr3ee/CVE-2022-23222/blob/master/exploit.c

Runtime Validation Pass: Exploits due to verifier bugs

A common exploit pattern involves passing invalid arguments to a helper function. CVE-2022-23222 creates a kernel pointer address leakage and finally obtains the root privilege by passing invalid arguments to the `bpf_skb_load_bytes` helper function [1].

[1] Full exploit code: https://github.com/tr3ee/CVE-2022-23222/blob/master/exploit.c

Runtime Validation Pass: Exploits due to verifier bugs

A common exploit pattern involves passing invalid arguments to a helper function. CVE-2022-23222 creates a kernel pointer address leakage and finally obtains the root privilege by passing invalid arguments to the `bpf_skb_load_bytes` helper function [1].

Root cause: verifier's incorrect inference of an argument's value (`OR_NULL` arithmetic).

```
r9 = r1
r0 = bpf_lookup_elem(ctx->comm_fd, 0)
if (r0 == NULL) exit(1)
r8 = r0
r0 = bpf_ringbuf_reserve(ctx->ringbuf_fd, PAGE_SIZE, 0)
r1 = r0
r1 += 1
if (r0 != NULL) { ringbuf_discard(r0, 1); exit(2); }
// verifier believes r0 = 0 and r1 = 0.
// However, r0 = 0 and r1 = 1 on runtime.
r7 = r1 + 8
// verifier believes r7 = 8, but r7 = 9 actually.
r0 = bpf_skb_load_bytes_relative(r9, 0, r8, r7, 0)
```

[1] Full exploit code: https://github.com/tr3ee/CVE-2022-23222/blob/master/exploit.c

Runtime Validation Pass

To capture this CVE, we could add a runtime check for the helper function `bpf_skb_load_bytes_relative`:

```
+ if r8 + r7 is outside the packet, throw exception
r0 = bpf_skb_load_bytes_relative(r9, 0, r8, r7, 0)
```

This technique is called dynamic parameter auditing (DPA) from MOAT (Hongyi et al.).

Runtime Validation Pass

To capture this CVE, we could add a runtime check for the helper function `bpf_skb_load_bytes_relative`:

```
+ if r8 + r7 is outside the packet, throw exception
r0 = bpf_skb_load_bytes_relative(r9, 0, r8, r7, 0)
```

This technique is called dynamic parameter auditing (DPA) from MOAT (Hongyi et al.). We generalize this technique as a PV pass that can be applied to any helper function. We perform runtime checks for every argument of the helper function call based on the verifier's expected value ranges. This pass can mitigate many similar exploits caused by verifier derivation bugs before causing any side effects.

Implementation Overview

EPASS is implemented in 12K lines of C code, including core (11K), supervisor, user space tools, and twelve passes.

Our implementation guidelines are keeping things **flexible**, **simple** and **easy-to-test**.

Therefore,

- EPASS supervisor that manges EPASS and the verifier is designed to be minimal (112 lines).
- EPASS core can also be compiled and used in user space. However, passes that require verifier information will not work.

User-space Tool

```
Usage: epass <command> [options] <file>

Commands:
  read              Read (lift, transform and compile) the specified file
  print             Print the specified file

Options:
  --pass-only, -P       Skip compilation
  --gopt <arg>  Specify global (general) option
  --popt <arg>  Specify pass option
  --sec, -s <arg>       Specify ELF section manually
  -F <arg>      Output format. Available formats: sec, log (default)
  -o <arg>      Output the modified program
```

Sample uses:

- Run MSan pass with default parameters on `prog.o`:

  `epass -m read -p prog.o --gopt "verbose=3" --popt msan`

- Run msan and heap pass with heap size 10MB and block size 128B on `prog.o`:

  `epass -m read -p prog.o --gopt "verbose=3" --popt "msan,heap(size=10MB,block=128B)"`

User-space Tool

We also provide `libbpf` and `bpftool` with EPASS support. Any applications depend on `libbpf` can easily enable EPASS by setting environment variables.

Example:

```
LIBBPF_ENABLE_EPASS=1 LIBBPF_EPASS_GOPT="verbose=3" LIBBPF_EPASS_POPT="msan" bpftool
prog load ...
```

User-friendly Error Messages

```
In BB 1,
  %3 = add(64) %1[2.dst], 0xffffffff(32)[2.imm],  // [2.insn]
  %4 = load %0         <--- Operand defined here
  %5 = add(64) %4, __BB_INSN_CRITICAL_CNT__,
In BB 1,
  %4 = load %0
  %5 = add(64) %4, __BB_INSN_CRITICAL_CNT__,          <--- Instruction that uses the operand
  store %0, %5
Error: ePass/core/aux/prog_check.c:195 <check_insn_operand> Instruction not found in the operand's users
```

Evaluation Overview

We show some results of our experiments regarding:

- Flexibility
- Security
- Latency and Throughput
- Runtime Overhead

Flexibility

Op.	EPASS	KM
Insert	18	39 (2.2x)
Lookup	175k	213k (1.2x)
Delete	8	37 (4.6x)

(a) Linked list.

EPASS	KM
262	351 (1.3x)
95	183 (1.9x)
102	254 (2.5x)

(b) Binary Search Tree.

Table: Latency of operations in flexible data structures. Numbers are in nanoseconds. Slower ratios are relative to EPASS.

Flexibility

Category	Root Cause	Collected Programs	Resolved by EPASS
C1	Lack of type information	4	4
C2	Limit calc. is inaccurate	10	8
C3	Arithmetic derivation failure	6	6
C4	Range analysis	3	3

Table: Number and cause of false-rejected programs we collect from real-world user reports.

Security

Type	Passes	CVE ID
Invalid memory access	MS, AS, RV	**2021-3490**, 2021-45402, 2021-33200, **2021-31440**, 2021-3444, 2023-39191, 2023-52452, 2024-35905, 2024-43910, 2024-45020
Helper abuse	RV	**2022-23222**, **2021-4204**, 2024-26589
Kernel DoS	CP	2024-47794, 2024-42072

Table: eBPF CVEs mitigated by EPASS. MS: MSan pass, AS: ASan pass, RV: runtime validation pass, CP: counter pass.

Latency and Throughout

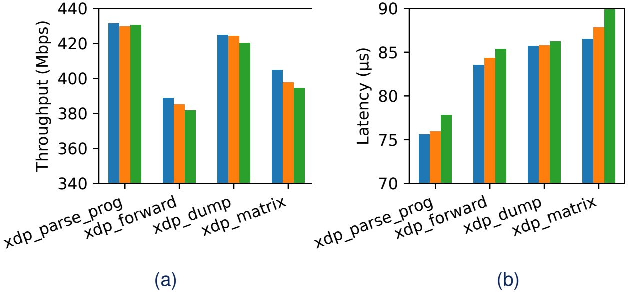


Figure: Throughput and latency of four XDP programs. Blue bar (left) is without EPASS, orange bar (middle) is with counter pass, green bar (right) is with MSan pass.

Runtime Overhead

Test	Vanilla	Falco		
		Raw	Counter	MSan
open/close	2.26	4.18	4.23 (1.2%)	4.26 (1.9%)
stat	1.20	1.81	1.85 (2.2%)	1.88 (3.8%)
fork process	308.52	347.48	352.61 (1.5%)	355.10 (2.2%)
exec process	981.17	1133.40	1162.60 (2.6%)	1139.60 (0.5%)
shell process	1882.89	2118.33	2122.66 (0.2%)	2136.66 (0.8%)
AF_UNIX	14.36	19.52	19.89 (1.8%)	20.03 (2.6%)
Postmark	42.42	55.23	55.86 (1.2%)	56.45 (2.2%)

Table: Application benchmark of EPASS on Falco. The first six are lmbench tests (in microseconds). The last is Postmark (in seconds). Vanilla is without Falco, and Raw is Falco without EPASS. Numbers in the parentheses indicate EPASS's overhead percentage.

Conclusion

Main Contributions

- Provide a new IR for eBPF, and a pass-based compiler framework.
- Allow developers to extend the expressiveness of eBPF language, add runtime checks and optimizations easily, safely, and with minimal overhead.
- Enhance flexibility and safety of eBPF programs.

Thanks for the attention!

EPASS is under development at https://github.com/OrderLab/ePass.
We welcome any feedback and suggestions!

Yiming Xiang
yiming@utexas.edu