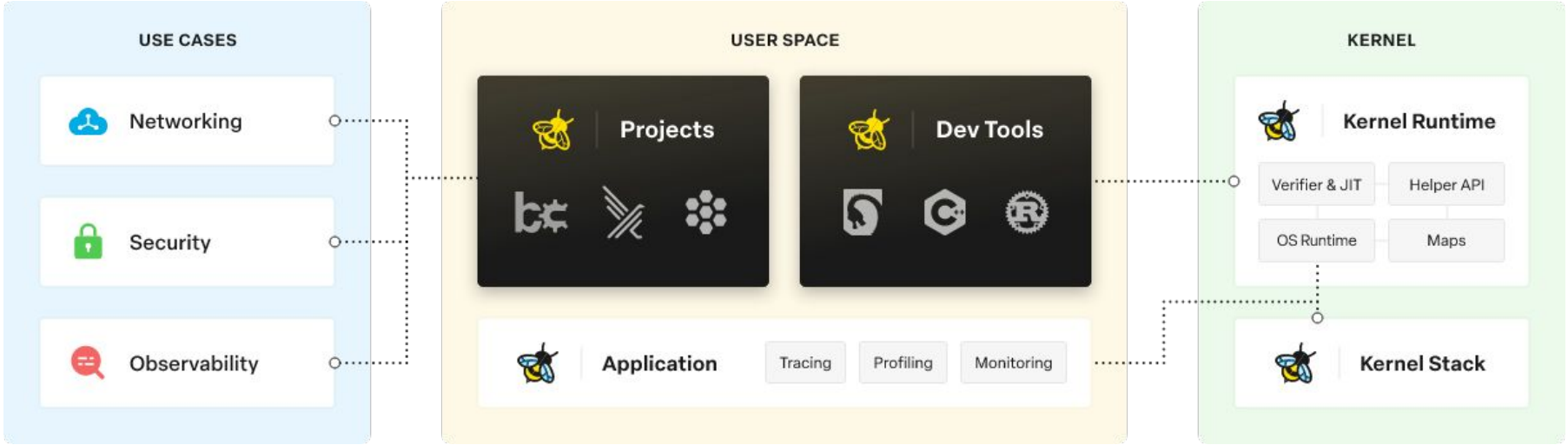
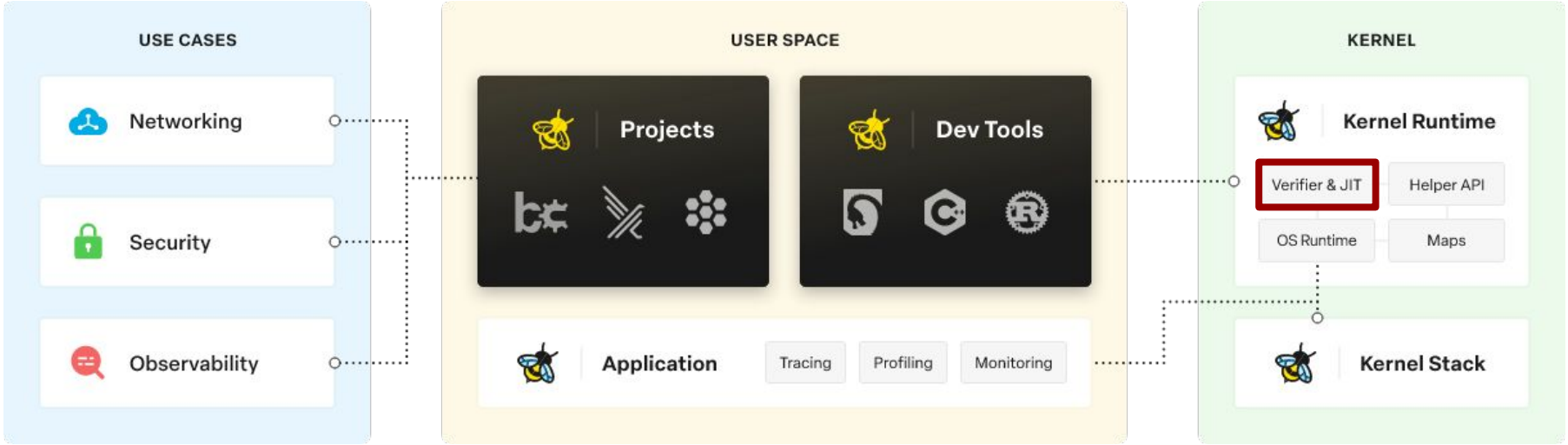
Russel Arbore

What is eBPF

- A bytecode program encoding that can be run in the kernel
- Can compile higher-level languages such as C into eBPF
- Kernel contains verifier to check that the eBPF bytecode will terminate and won’t crash
- Also contains a JIT compiler to convert to machine code for faster execution
- Sandboxing happens statically - once running, can monitor various aspects of the kernel at full performance


USE CASES
Networking
Security
Observability
USER SPACE
Projects
Dev Tools
Application
Tracing
Profiling
Monitoring
KERNEL
Kernel Runtime
Verifier & JIT
Helper API
OS Runtime
Maps
Kernel Stack


USE CASES
Networking
Security
Observability
USER SPACE
Projects
Dev Tools
Application
Tracing
Profiling
Monitoring
KERNEL
Kernel Runtime
Verifier & JIT
Helper API
OS Runtime
Maps
Kernel Stack

eBPF bytecode

- 10 GP registers
- Calling convention where each register is used as return values, function arguments, or callee saved
- Each instruction is 64 or 128 bits (second 64 bits is optional constant immediate value)
- First 64 bits broken down as:
  - Opcode: 8 bits
  - Src register: 4 bits
  - Dst register: 4 bits
  - Offset: 16 bits
  - Immediate: 32 bits
- If using 64 bit immediate, then last 32 bits of second 64 bits is concatenated with 32 bit immediate in first 64 bits

class	value	description	reference
BPF_LD	0x00	non-standard load operations	Load and store instructions
BPF_LDX	0x01	load into register operations	Load and store instructions
BPF_ST	0x02	store from immediate operations	Load and store instructions
BPF_STX	0x03	store from register operations	Load and store instructions
BPF_ALU	0x04	32-bit arithmetic operations	Arithmetic and jump instructions
BPF_JMP	0x05	64-bit jump operations	Arithmetic and jump instructions
BPF_JMP32	0x06	32-bit jump operations	Arithmetic and jump instructions
BPF_ALU64	0x07	64-bit arithmetic operations	Arithmetic and jump instructions

code	value	description
BPF_ADD	0x00	dst += src
BPF_SUB	0x10	dst -= src
BPF_MUL	0x20	dst *= src
BPF_DIV	0x30	dst = (src != 0) ? (dst / src) : 0
BPF_OR	0x40	dst |= src
BPF_AND	0x50	dst &= src
BPF_LSH	0x60	dst <<= (src & mask)
BPF_RSH	0x70	dst >>= (src & mask)
BPF_NEG	0x80	dst = ~src
BPF_MOD	0x90	dst = (src != 0) ? (dst % src) : dst
BPF_XOR	0xa0	dst ^= src
BPF_MOV	0xb0	dst = src
BPF_ARSH	0xc0	sign extending dst >>= (src & mask)
BPF_END	0xd0	byte swap operations (see Byte swap instructions below)

Interprogram Communication

1.5.2 Atomic operations

Atomic operations are operations that operate on memory and can not be interrupted or corrupted by other access to the same memory region by other eBPF programs or means outside of this specification.

All atomic operations supported by eBPF are encoded as store operations that use the `BPF_ATOMIC` mode modifier as follows:

- `BPF_ATOMIC | BPF_W | BPF_STX` for 32-bit operations
- `BPF_ATOMIC | BPF_DW | BPF_STX` for 64-bit operations
- 8-bit and 16-bit wide atomic operations are not supported.

The 'imm' field is used to encode the actual atomic operation. Simple atomic operation use a subset of the values defined to encode arithmetic operations in the 'imm' field to encode the atomic operation:

1.5.3.1 Maps

Maps are shared memory regions accessible by eBPF programs on some platforms. A map can have various semantics as defined in a separate document, and may or may not have a single contiguous memory region, but the 'map_val(map)' is currently only defined for maps that do have a single contiguous memory region.

Each map can have a file descriptor (fd) if supported by the platform, where 'map_by_fd(imm)' means to get the map with the specified file descriptor. Each BPF program can also be defined to use a set of maps associated with the program at load time, and 'map_by_idx(imm)' means to get the map with the given index in the set associated with the BPF program containing the instruction.

eBPF bytecode

- What is it?

eBPF bytecode

- What is it?
- It’s a low-level virtual assembly language.

eBPF Verifier

- Verify that the given eBPF program will not:
  - Crash
  - Perform malicious operations
  - Run forever
- The verifier accepts a turing-incomplete subset of eBPF bytecode
- Sound, incomplete

eBPF Verifier

```
bpf_mov R0 = R2
bpf_exit
```

```
bpf_mov R1 = 1
bpf_call foo
bpf_mov R0 = R1
bpf_exit
```

```
bpf_mov R1 = 1
bpf_mov R2 = 2
bpf_xadd *(u32 *)(R1 + 3) += R2
bpf_exit
```

eBPF Verifier

```
bpf_mov R0 = R2
bpf_exit
```

Read before write

```
bpf_mov R1 = 1
bpf_call foo
bpf_mov R0 = R1
bpf_exit
```

Not callee saved

```
bpf_mov R1 = 1
bpf_mov R2 = 2
bpf_xadd *(u32 *)(R1 + 3) += R2
bpf_exit
```

Not pointer type

eBPF Verifier

1. Construct CFG from bytecode, verify that it's a DAG (no loops)
2. Perform abstract interpretation from root instruction, tracking…
    a. Valid registers
    b. What registers point to
    c. Loads / stores done with correct alignment
    d. Loads only read from previously stored memory
    e. "bitmasks, smallest and largest possible values, and equivalence classes of values using identity-tracking"

eBPF Verifier

1. Construct CFG from bytecode, verify that it's a DAG (no loops)
2. Perform abstract interpretation from root instruction, tracking…
    a. Valid registers
    b. What registers point to
    c. Loads / stores done with correct alignment
    d. Loads only read from previously stored memory
    e. "bitmasks, smallest and largest possible values, and equivalence classes of values using identity-tracking"

Exponential run time?

eBPF Verifier

1. Construct CFG from bytecode, verify that it's a DAG (no loops)
2. Perform abstract interpretation from root instruction, tracking…
    a. Valid registers
    b. What registers point to
    c. Loads / stores done with correct alignment
    d. Loads only read from previously stored memory
    e. "bitmasks, smallest and largest possible values, and equivalence classes of values using identity-tracking"

Exponential run time? They do pruning?

eBPF Verifier

1. Construct CFG from bytecode, verify that it's a DAG (no loops)
2. Perform abstract interpretation from root instruction, tracking…
    a. Valid registers
    b. What registers point to
    c. Loads / stores done with correct alignment
    d. Loads only read from previously stored memory
    e. "bitmasks, smallest and largest possible values, and equivalence classes of values using identity-tracking"

Exponential run time? They do pruning?

Formally verified?

eBPF Verifier

1. Construct CFG from bytecode, verify that it's a DAG (no loops)
2. Perform abstract interpretation from root instruction, tracking…
    a. Valid registers
    b. What registers point to
    c. Loads / stores done with correct alignment
    d. Loads only read from previously stored memory
    e. "bitmasks, smallest and largest possible values, and equivalence classes of values using identity-tracking"

Exponential run time? They do pruning?

Formally verified? No…

Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions

Elazar Gershuni
Tel Aviv University, Israel and
VMware Research, USA
elazarg@gmail.com

Nadav Amit
VMware Research, USA
namit@vmware.com

Arie Gurfinkel
University of Waterloo, Canada
arie.gurfinkel@uwaterloo.ca

Nina Narodytska
VMware Research, USA
nnarodytska@vmware.com

Jorge A. Navas
SRI International, USA
jorge.navas@sri.com

Noam Rinetzky
Tel Aviv University, Israel
maon@cs.tau.ac.il

Leonid Ryzhyk
VMware Research, USA
lryzhyk@vmware.com

Mooly Sagiv
Tel Aviv University, Israel
msagiv@cs.tau.ac.il

Abstract

Extended Berkeley Packet Filter (eBPF) is a Linux subsystem that allows safely executing untrusted user-defined extensions inside the kernel. It relies on static analysis to protect the kernel against buggy and malicious extensions. As the eBPF ecosystem evolves to support more complex and diverse extensions, the limitations of its current verifier, including high rate of false positives, poor scalability, and lack of support for loops, have become a major barrier for developers.

We design a static analyzer for eBPF within the framework of abstract interpretation. Our choice of abstraction is based on common patterns found in many eBPF programs. We observed that eBPF programs manipulate memory in a rather disciplined way which permits analyzing them successfully with a scalable mixture of very-precise abstraction of certain bounded regions with coarser abstractions of other parts of the memory. We use the Zone domain, a simple domain that tracks differences between pairs of registers and offsets, to achieve precise and scalable analysis. We demonstrate

CCS Concepts • **Security and privacy → Operating systems security**; • **Software and its engineering → Automated static analysis**; *Software safety*.

Keywords ebpf, static analysis, linux, kernel extensions

ACM Reference Format:
Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314590

1 Introduction

We consider the problem of verifying *untrusted* kernel extensions. Modern operating systems achieve most of their functionality through dynamically loaded extensions that implement support for I/O devices, file systems, networking,

PLDI 2019 Verifier

- Pointers represented using abstract (region, offset) values
- Zone abstract domain for numerics
  - Supports predicates of the form X - Y <= C, X and Y are variables, C is a constant
- Supports loops
- Defines eBPFPL, a low level programming language that captures the essence of eBPF
  - For the purposes of the paper, eBPFPL is verified, not eBPF
- Define a formal semantics of eBPFPL
- Verifier available at: https://github.com/vbpf/ebpf-verifier

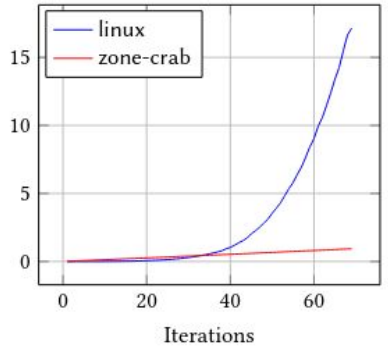
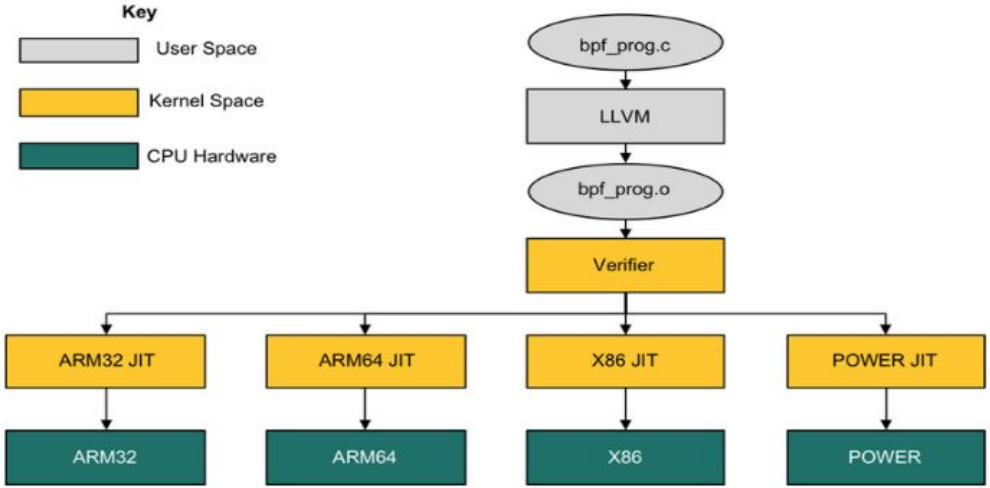


Figure 11. Execution Time (Sec) on Double `strncmp`

eBPF JIT


Key
User Space
Kernel Space
CPU Hardware
bpf_prog.c
LLVM
bpf_prog.o
Verifier
ARM32 JIT
ARM64 JIT
X86 JIT
POWER JIT
ARM32
ARM64
X86
POWER

eBPF JIT

1. Overestimate final memory usage
2. Build prologue
3. Emit instructions
    a. One-to-one mapping from eBPF instructions to target instructions
4. Build epilogue

eBPF JIT

1. Overestimate final memory usage
2. Build prologue
3. Emit instructions
    a. One-to-one mapping from eBPF instructions to target instructions
4. Build epilogue

Most of the compiler complexity goes into generating eBPF, not JITing it.

How to use eBPF?

Method 1: bpf(2) / libbpf

- Manually compile BPF program using clang, load into kernel and run with bpf(2)

Method 1: bpf(2) / libbpf

- Manually compile BPF program using clang, load into kernel and run with bpf(2)
- Really annoying

Method 2: bpftrace

- Quickly write trace programs, run them in one line
- No need to set up build system

Demo!