BeeBox

Hardening BPF against Transient Execution Attacks

Di Jin Alexander J. Gaidis Vasileios P. Kemerlis

Secure Systems Laboratory (SSL)
Department of Computer Science
Brown University

Overview

- The separation between the OS and applications can be at odds with modern computing infrastructure's performance requirements
- **BPF** bends the line of separation, reducing context switching, data copying, and unnecessary software stack traversals

✖ ... but it undermines kernel security, especially in the presence of **transient execution attacks**

✔ BeeBox is here to save the day!

BROWN

Background

Berkeley Packet Filter (BPF)

- User apps ➜ *Safely* delegate computations to the OS kernel[1]
- A small virtual architecture with a RISC-like instruction set
- Many applications
  - Packet filtering
  - Networking (Cilium, Katran, ...)
  - System call filtering (Android, Docker, Chrome, OpenSSH, Tor, ...)
  - Kernel profiling
  - FUSE (Filesystem in Userspace)[2]
  - High-performance storage[3]
  - ...

[1] *The BSD Pkt. Filter: A New Arch. for User-level Pkt. Capture.* USENIX Winter 1993
[2] *Extension Framework for File Systems in User space.* USENIX ATC 2019
[3] *XRP: In-Kernel Storage Functions with eBPF.* OSDI 2022

BROWN

BPF Features

- Basic instructions ➜ ALU, memory access, conditional branching
- Safety ➜ Statically verified (termination, non-leaking behavior)
- JIT-compilation
- Runtime environment
  - BPF helpers ➜ Pre-defined native kernel funcs. (invoked from BPF code)
  - BPF context ➜ Data passed to BPF program by the kernel
  - BPF stack ➜ Scratch space used by BPF programs
  - BPF maps ➜ A collection of data structures used to store aggregated results (array, hash table, bloomfilter, *etc.*)

BROWN

Transient Execution

- Pipeline, slow memory ➜ Out-of-order/speculative/... execution
- Being speculative ➜ Need to revert instructions
  - mis-prediction: conditional branch target, indirect branch target, memory access address, ...
  - exception: page fault, permission check, ...
- Transient instructions := the instructions that are squashed[4]

[4] *A Systematic Evaluation of Transient Execution Attacks and Defenses.* USENIX SEC 2019

BROWN

Transient Execution Vulnerabilities

(a) Transient execution **may break** the semantics intended by the developer

- Permission restrictions
- Sanity checks
- Other invariants

(b) Transient execution cannot be fully reverted, leaving the door open for **side-channel attacks**

(a) + (b) ⇒ Information leakage from a **correct** program

BROWN

BPF and Transient Execution Attacks

BPF is an attractive attack vector for transient execution attacks

- BPF JIT ⇒ Power to "push" native code in kernel space
- BPF operates on user data ⇒ Complete control over the execution trace
- BPF works on hot paths ⇒ Side channels are less noisy
- BPF lives in the kernel ⇒ Bypasses cross-domain defenses

BROWN

Existing Solutions and their Limitations

LPM (Linux Provisional Mitigations) ≔ Existing defenses in Linux BPF

- Analysis over impossible branch/value combinations ➜ Prevents unsafe behavior in speculation
- Speculation barriers ➜ Prevents speculative store bypass

LPM limitations

- Compatibility ➜ Rejects legal programs
  - Forbids some pointer arithmetics due to uncertain speculative behavior
  - Rejects programs due to analysis-time explosion
- Performance ➜ Significant overhead for complex BPF programs
- Scope ➜ Does not protect helper functions

BROWN

BeeBox

BeeBox Design

BeeBox goals:

- Defend against Spectre-PHT (v1) and Spectre-STL (v4)
  - Similar to LPM
  - Other transient execution attacks are handled by generic defenses
- Maintain full compatibility
- Remain performant

Key idea:

- **Allow any (mis-)speculation**
- ... but **sandbox** all memory accesses

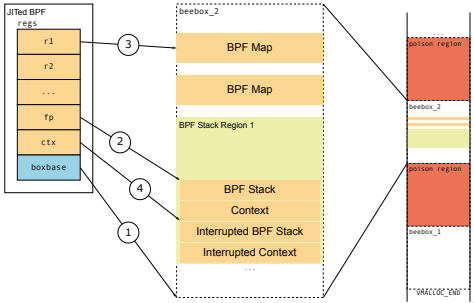
$\Rightarrow$ No sensitive data (kernel data) is exposed during speculation

BROWN

BeeBox Design

Inspired by Software-Fault Isolation (SFI)[5] techniques

- Sandbox region per user (4GB)
- Dedicated boxbase register (①)
- Transform BPF pointers
- Isolating BPF's data
  - BPF stack
  - BPF maps
  - BPF context



[5] *Efficient software-based fault isolation.* OSDI 1993

BeeBox Design: Instrumentation

```
...
r2 = $map_ptr

r3 = r2 + r3

r4 = *(u64 *) r3

...
```

```
...
mov rsi, 0xffffc90000001000

add rcx, rsi

mov rdx, qword ptr [rcx]

...
```

```
mov r12, 0xffffc90000000000
...
mov rsi, 0x1000

add rcx, rsi

mov ecx, ecx
mov rdx, qword ptr [r12 + rcx]
...
```

Example **BPF program**, its **vanilla JIT-ed code**, and its **BeeBox JIT-ed code**

- If array index (`r3` and `rcx`) is **speculatively** out-of-bound

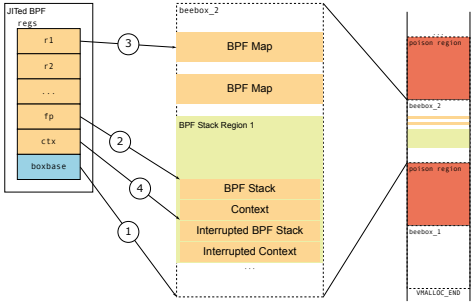
$\Rightarrow$ *Out-of-bound access in vanilla*

$\Rightarrow$ **Sandbox-ed access in BeeBox**

BROWN

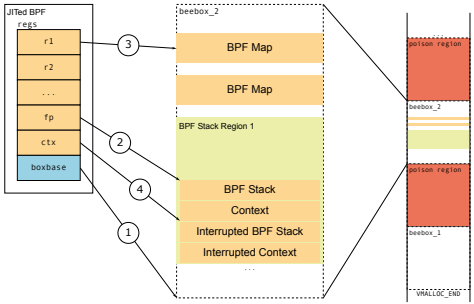
BeeBox Design: BPF Stack

- BPF stack (②) ➜ local variables
  - Only used by BPF
  - Allocated per execution
- Pre-allocate per-CPU region
  - Reduce memory allocation



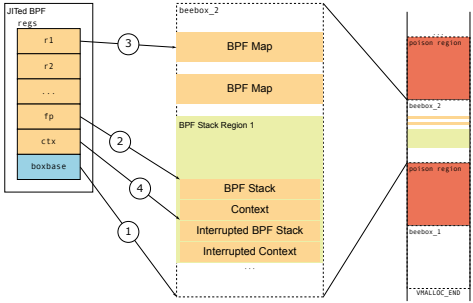
BeeBox Design: BPF Maps

- BPF map (③) ➜ data structures storing aggregated results
  - Used by BPF and the kernel
  - Persistent allocation
- Split BPF maps
  - Kernel metadata ➜ Outside
  - BPF (meta)data ➜ Inside



BeeBox Design: BPF Context

- BPF context (④) ➜ data passed in by the kernel
  - Used by BPF and the kernel
  - Allocated by the kernel
- Copy to BPF stack region



Context-copying Optimizations

Copying context can be *costly*

- **BeeBox-RC** (reduce copy)
  - ➜ Only copy fields that are used $\rightsquigarrow$ via static analysis
  - ✔ Generally applicable

BROWN

Context-copying Optimizations

Copying context can be *costly*

- **BeeBox-RC** (reduce copy)
  - ➜ Only copy fields that are used $\rightsquigarrow$ via static analysis
  - ✔ Generally applicable
- **BeeBox-RB** (ring buffer)
  - ➜ When the context's entire page is accessible, map it into the sandbox
  - ✔ Applicable for device-level BPF (*e.g.*, XDP)

BROWN

Context-copying Optimizations

Copying context can be *costly*

- **BeeBox-RC** (reduce copy)
  - ➜ Only copy fields that are used $\rightsquigarrow$ via static analysis
  - ✔ Generally applicable
- **BeeBox-RB** (ring buffer)
  - ➜ When the context's entire page is accessible, map it into the sandbox
  - ✔ Applicable for device-level BPF (*e.g.*, XDP)
- **BeeBox-CP** (clean pointer)
  - ➜ Avoid instrumenting context pointers when they cannot be speculatively hijacked $\rightsquigarrow$ via static analysis
    - Applicable to BPF programs with simple context usage (*e.g.*, packet filter and cBPF)

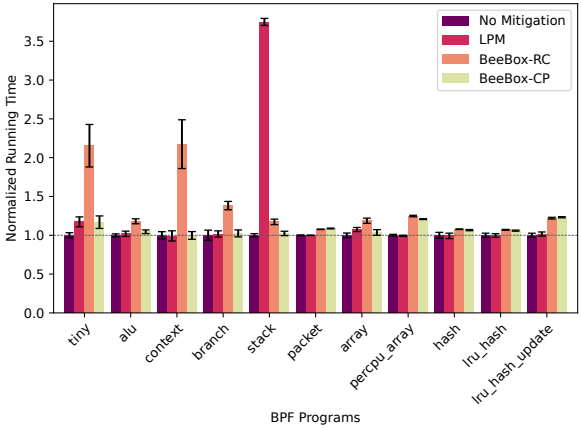
Other Interesting Features

- Helper instrumentation is guided by annotations and static analysis
- `boxbase` register reservation in BPF's runtime
- Return addresses are separated from the BPF stack
- Support for interrupts and re-entrant BPF programs

BROWN

Evaluation

Evaluation: Synthetic Micro-benchmarks

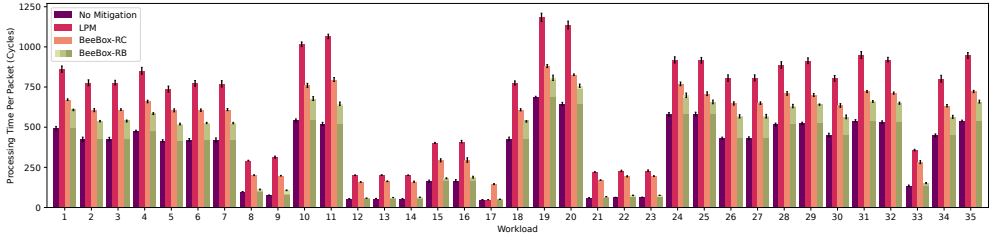


Manually-written BPF programs that stress different BPF access patterns

➜ 0%–23% overhead

BROWN

Evaluation: Katran



Katran's performance test suite for load balancing

➜ ≈20% overhead (vs. ≈112% of LPM)

BROWN

Evaluation: Packet Filtering and `seccomp-BPF`

Benchmark	No Mitigation	BeeBox
Nginx	0.81% (± 1.09%)	0.32% (± 1.47%)
Redis	0.98% (± 0.44%)	0.84% (± 0.74%)

Throughput decrease of `seccomp-BPF` (95% CIs)

Filter	No Mitigation	BeeBox-CP	%-Chg
`bpf1`	325927 (± 3611)	327778 (± 3006)	+0.57%
`bpf2`	324615 (± 3960)	323374 (± 5375)	−0.38%
`bpf3`	324114 (± 3977)	323834 (± 5088)	−0.09%
`bpf4`	328610 (± 4827)	325568 (± 7818)	−0.93%
`bpf5`	328072 (± 3883)	325395 (± 7352)	−0.82%
`bpf6`	314801 (± 2025)	313618 (± 2650)	−0.38%

Packet filtering performance in pkts/sec (95% CIs)

BROWN

Conclusion

- BPF is important, but prone to transient execution attacks
- BeeBox mitigates the problem by sandboxing the BPF runtime, restricting memory access targets under speculation
- BeeBox is **efficient** and more **secure** than LPM, while maintaining **compatibility** with vanilla BPF

✪ Artifact ⇝ https://gitlab.com/brown-ssl/beebox

BROWN

Backup Slides

Securing Helper Functions

Helper functions need to change accordingly:

- Compatibility $\rightsquigarrow$ different pointer representation
- Security $\rightsquigarrow$ untrusted pointer value from BPF

To secure a helper function:

- ✔ We instrument sandbox-ed pointer dereferences
- ✔ We verify that native pointers are safe to access
  - If not, we fall back to inserting speculation barriers

BROWN

Securing Helper Functions: Instrumentation

```
static void *array_map_lookup_elem(struct bpf_array *array, void *key)
{
        u32 index = *(u32 *)key;

        if (index >= array->map.max_entries)
                return NULL;

        return array->value + (u64)array->elem_size * index;
}
```

Securing Helper Functions: Instrumentation

```
static __beebox void *array_map_lookup_elem(struct bpf_array __beebox *array,
                                            void __beebox *key)
{
        u32 index = *(u32 *) unbox (key);

        if (index >= *unbox ( & array->max_entries))
                return NULL;

        return array->value + (u64) *unbox ( & array->elem_size) * index;
}
```

- `__beebox` annotation ➜ Avoid normal dereference
- `unbox` macro ➜ Apply instrumentation and dereference
- Checking is achieved via the static analysis tool `sparse`[6]

[6] https://www.kernel.org/doc/html/latest/dev-tools/sparse.html

BROWN

Securing Helper Functions: Clean Pointers

- Clean pointers :=
  - Embedded immediate
  - Pointers loaded through clean pointers
- Safety requirement
  - No spilling into sandbox
  - No leaking into BPF runtime

✔ Can be identified by static analysis

```
bpf_get_current_uid_gid:
  mov    rax, QWORD PTR gs:0x1ad00  # struct task_struct
  test   rax, rax
  je     1f
  mov    rax, QWORD PTR [rax+0x638] # struct cred
  mov    rax, QWORD PTR [rax+0x4]   # uid, gid
  ret
1:
  mov    rax, 0xffffffffffffffea     # -EINVAL
  ret
```

BROWN

Memory Usage

Experiment	Vanilla Usage	BeeBox Usage	Overhead
At rest	176MB (178MB)	180MB (180MB)	2.4%
Packet filter	182MB (183MB)	186MB (188MB)	2.3%
Katran	580MB (582MB)	592MB (592MB)	2.0%
Nginx (seccomp)	189MB (190MB)	196MB (197MB)	3.5%
Redis (seccomp)	212MB (213MB)	218MB (221MB)	3.0%

Memory usage of BeeBox compared to vanilla Linux. 'At rest' means no workload is running. The reported numbers are formatted as *avg (max)*.

BROWN

Scope and Compatibility

Category	Feature	LPM	retpoline	IBRS	KPTI	BeeBox
Security	Block Spectre-PHT in BPF code	✓				✓
	Block Spectre-PHT in BPF helpers					✓
	Block Spectre-STL in BPF code	✓				✓
	Block Spectre-STL in BPF helpers					✓
	Block Spectre-BTB		✓	✓		
	Block Meltdown				✓	
Compatibility	Allow conditional ptr. arithmetic in unpriv. BPF					✓
	Avoid verifier state explosion in unpriv. BPF					✓

Existing Linux kernel defenses and BeeBox's coverage over transient execution attacks and compatibility features.

BROWN

Scope and Compatibility

Category	Feature	LPM	retpoline	IBRS	KPTI	BeeBox
Security	Block Spectre-PHT in BPF code	✓				✓
	Block Spectre-PHT in BPF helpers					✓
	Block Spectre-STL in BPF code	✓				✓
	Block Spectre-STL in BPF helpers					✓
	Block Spectre-BTB		✓	✓		
	Block Meltdown				✓	
Compatibility	Allow conditional ptr. arithmetic in unpriv. BPF					✓
	Avoid verifier state explosion in unpriv. BPF					✓

Existing Linux kernel defenses and BeeBox's coverage over transient execution attacks and compatibility features.

BROWN

Scope and Compatibility

Category	Feature	LPM	retpoline	IBRS	KPTI	BeeBox
Security	Block Spectre-PHT in BPF code	✓				✓
	Block Spectre-PHT in BPF helpers					✓
	Block Spectre-STL in BPF code	✓				✓
	Block Spectre-STL in BPF helpers					✓
	Block Spectre-BTB		✓	✓		
	Block Meltdown				✓	
Compatibility	Allow conditional ptr. arithmetic in unpriv. BPF					✓
	Avoid verifier state explosion in unpriv. BPF					✓

Existing Linux kernel defenses and BeeBox's coverage over transient execution attacks and compatibility features.

BROWN