BPF LSM Updates & Progress

KP Singh

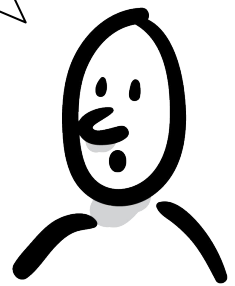
The Story...

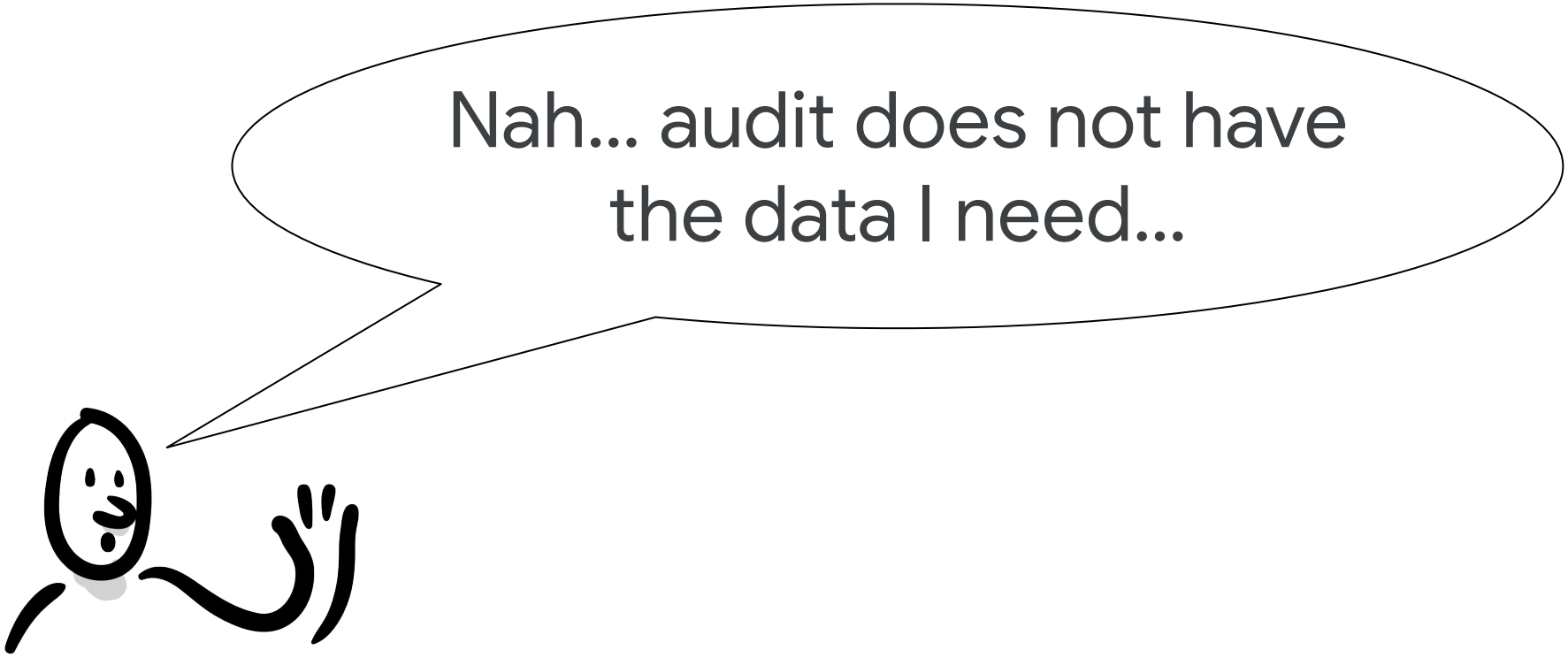
A long long time ago...

Actually, sometime back in 2019...


Hey! I need some audit logs...

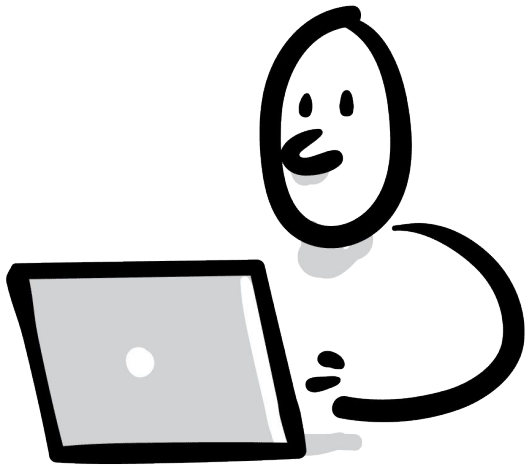
Can't you use Audit?



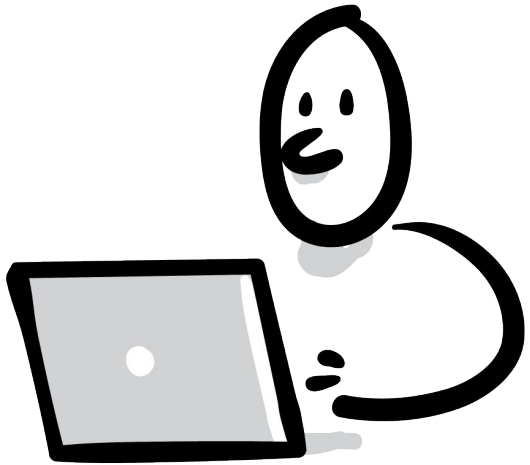

Nah... audit does not have
the data I need...


Okay... I will
modify audit...

Patch audit in the kernel...



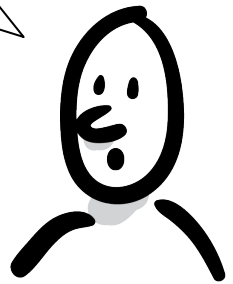
Update auditctl and userspace...



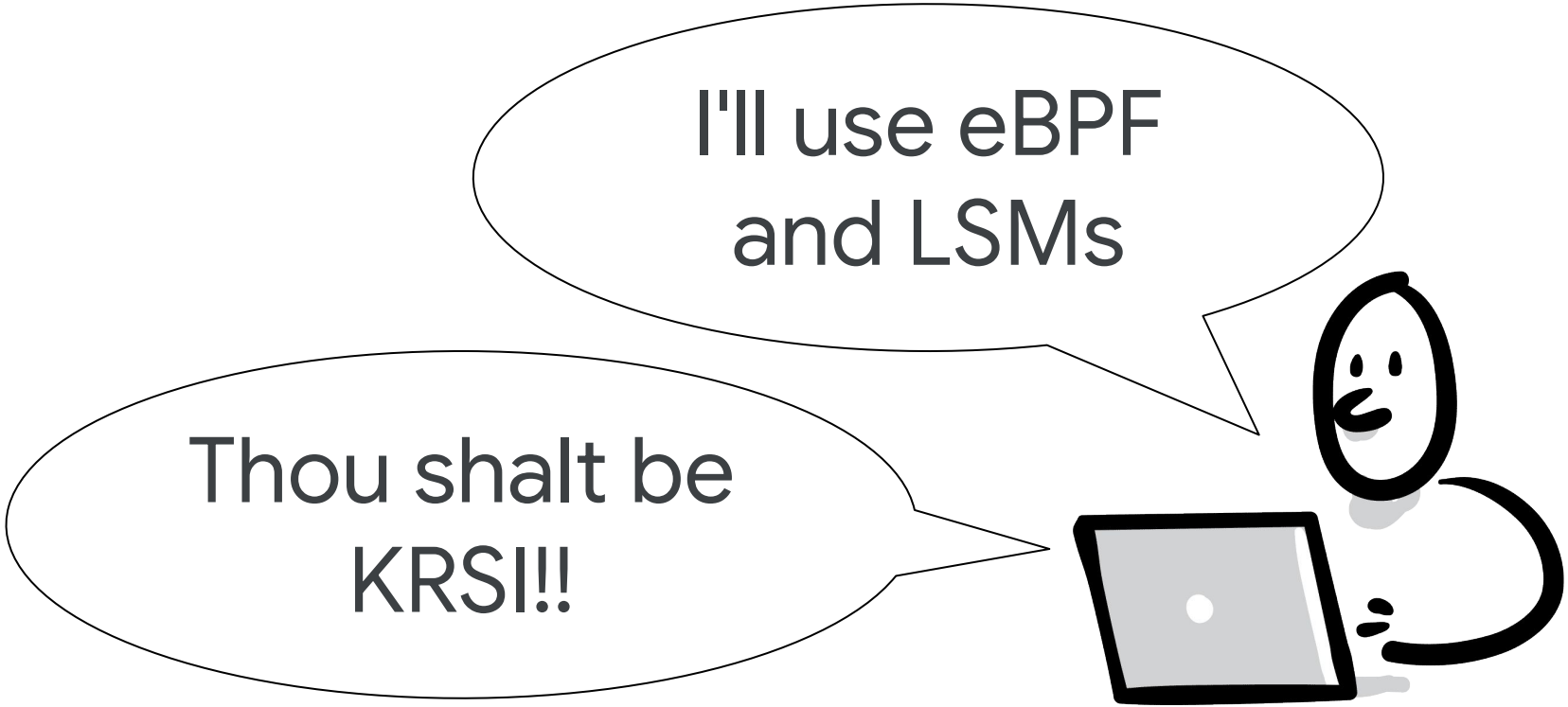

Erm...I want to use
this data to prevent
something...

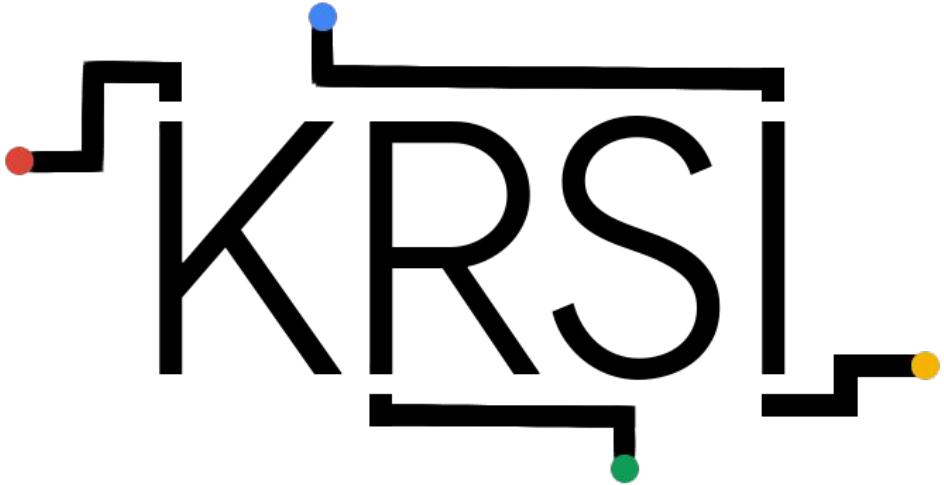
Rinse and Repeat for LSMs...

We just need a new way to do Security in Linux...




Okay, do it...


I'll use eBPF
and LSMs
Thou shalt be
KRSI!!

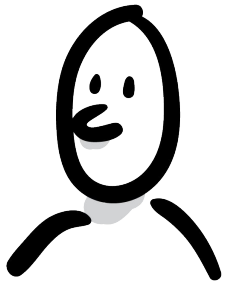


and

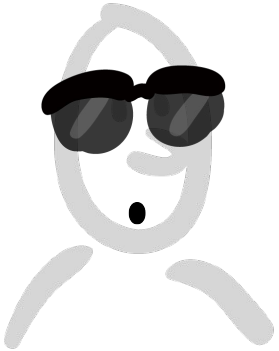
"Kernel Runtime Security Instrumentation"

was born..

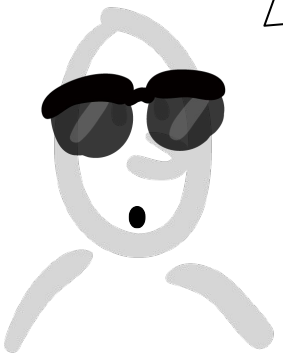
To LPC
Portugal!



The only way bpf-based LSM can land is both landlock and KRSI developers work together on a design that solves all use cases.

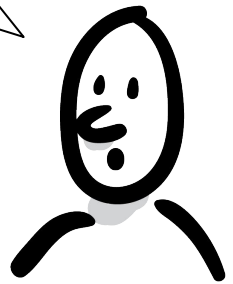


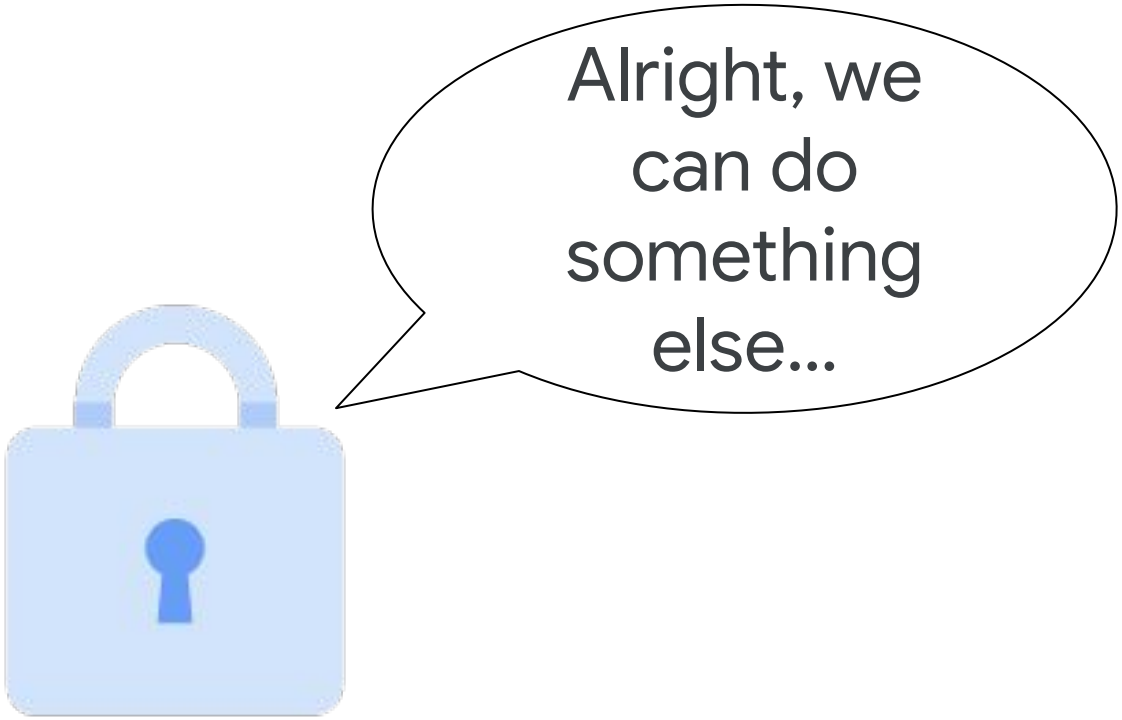
Oh and KRSI is a "great" name!



We want unprivileged Sandboxing

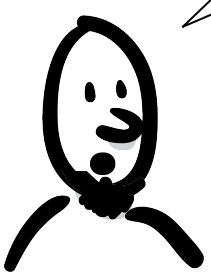
Unprivileged eBPF is still quite some-time away...




Alright, we can do something else...

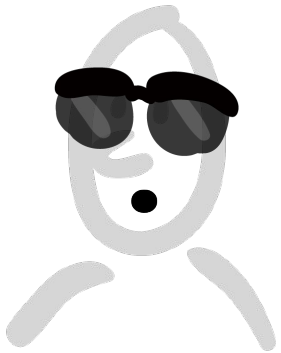
...and now we present security v/s eBPF...

The LSM mechanism
is not zero overhead.
It never has been.



I don't care!

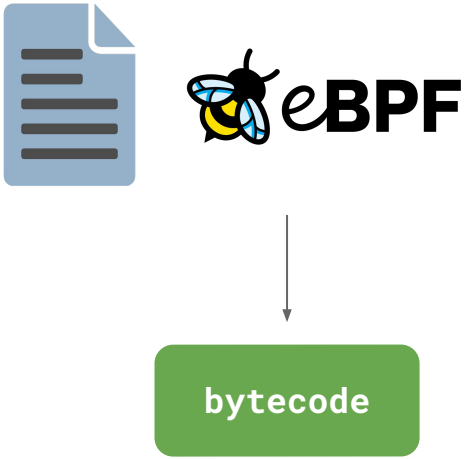
I think the key mistake we made is that we classified KRSI as LSM

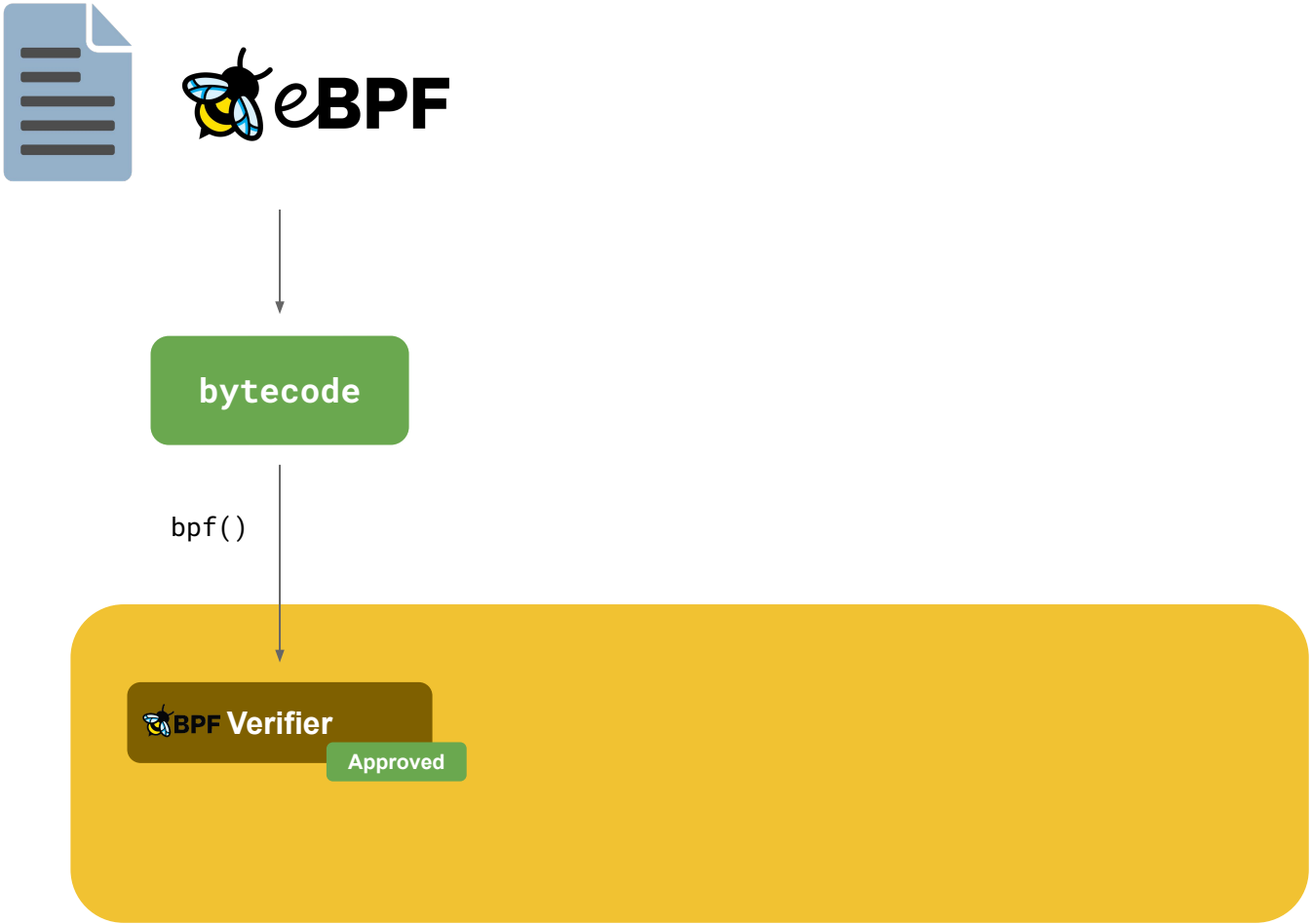


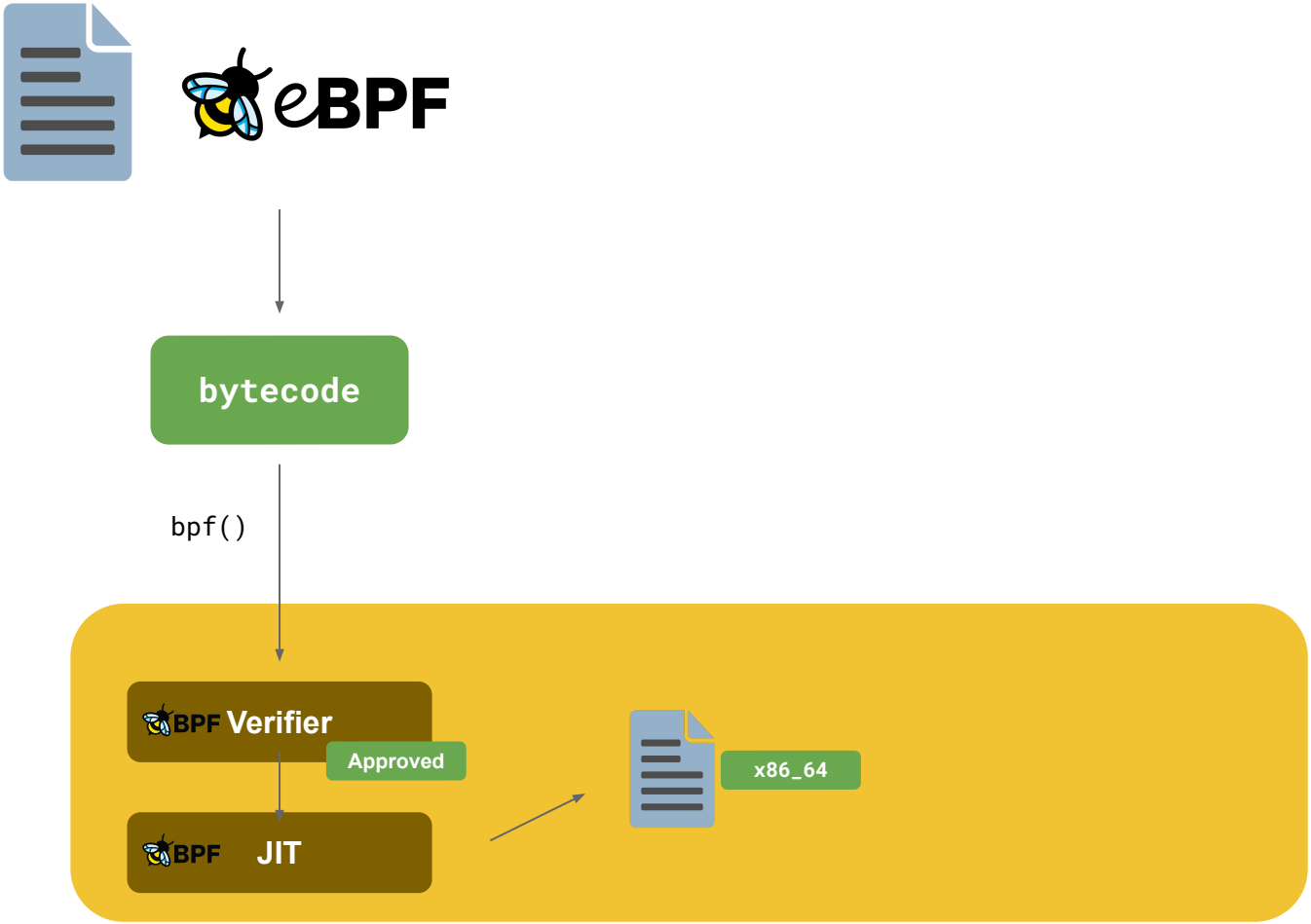
The Treaty of Impedance was signed...

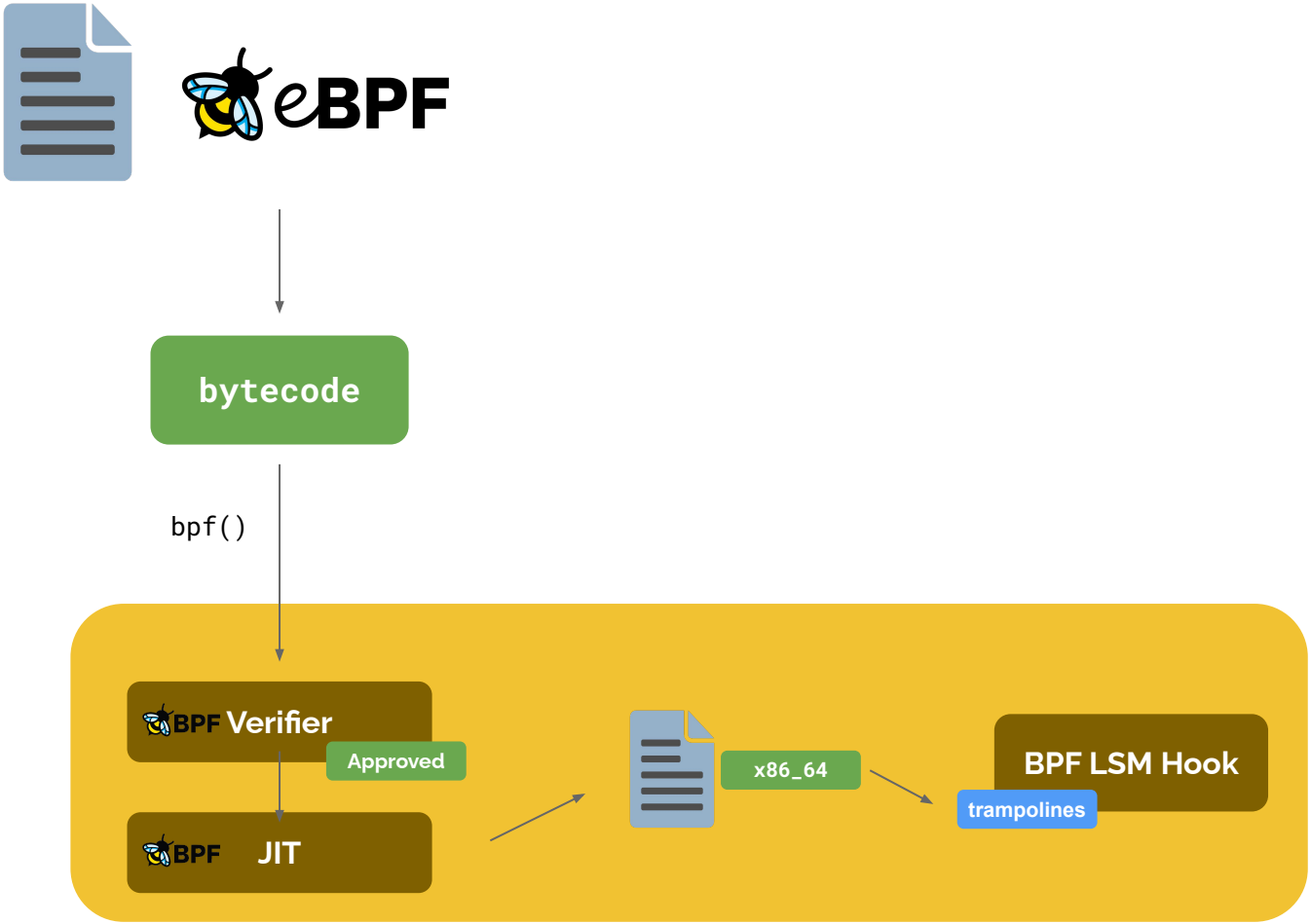
and KRSI was merged as BPF LSM!

How to use the BPF LSM?


eBPF
bytecode


eBPF
bytecode
bpf()
BPF Verifier
Approved


eBPF
bytecode
bpf()
BPF Verifier
Approved
BPF JIT
x86_64


eBPF
bytecode
bpf()
BPF Verifier
Approved
BPF JIT
x86_64
BPF LSM Hook
trampolines

Step 1: Hook into an appropriate LSM hook

```
SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{



        return 0;
}
```

Step 2: Use eBPF helpers

```
SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{
        __u32 pid = bpf_get_current_pid_tgid();


        return 0;
}
```

Step 3: Access structure fields with BTF

```
struct vm_area_struct {
        unsigned long vm_start;
} __attribute__((preserve_access_index));

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{
        __u32 pid = bpf_get_current_pid_tgid();
        unsigned long vm_start = vma->vm_start;

        return 0;
}
```

Step 4: Share variables with userspace

```
int mprotect_count = 0;

struct vm_area_struct {
        unsigned long vm_start, vm_end;
} __attribute__((preserve_access_index));

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{
        __u32 pid = bpf_get_current_pid_tgid();
        int vm_start = vma->vm_start;
        __sync_fetch_and_add(mprotect_count, 1);
        return 0;
}
```

Step 5: Allow or deny an operation

```
int mprotect_count = 0;

struct vm_area_struct {
        unsigned long vm_start, vm_end;
} __attribute__((preserve_access_index));

SEC("lsm/file_mprotect")
int BPF_PROG(mprotect_example, struct vm_area_struct *vma,
             unsigned long reqprot, unsigned long prot)
{
        __u32 pid = bpf_get_current_pid_tgid();
        int vm_start = vma->vm_start;
        __sync_fetch_and_add(mprotect_count, 1);
        return (mprotect_count > 100) ? -EPERM : 0;
}
```

Future work 1:
Improve performance of LSM hooks

Performance Impact 1: Indirect Calls

```
hlist_for_each_entry(P, &security_hook_heads.FUNC, list)    \
{                                                           \
    RC = P->hook.FUNC(__VA_ARGS__);                         \
    if (RC != 0)                                            \
        break;                                              \
}
```

Indirect calls = retpoline overhead

```
<security_bprm_check>:
  [...]
    mov     0x18(%rbx),%rax
    mov     %rbp,%rdi
    call    1812 <security_bprm_check+0x22>
        R_X86_64_PLT32     __x86_indirect_thunk_rax-0x4
    test    %eax,%eax
    jne     1820 <security_bprm_check+0x30>
    mov     (%rbx),%rbx
    test    %rbx,%rbx
    jne     1806 <security_bprm_check+0x16>
    xor     %eax,%eax
[...]
    ret
```

What's this thunk?

```
__x86_indirect_thunk_rax:

    call   setup_target

    capture_spec:

        pause
        lfence
        jmp    capture_spec

    setup_target:

        mov    %rax,(%rsp)
        ret
```

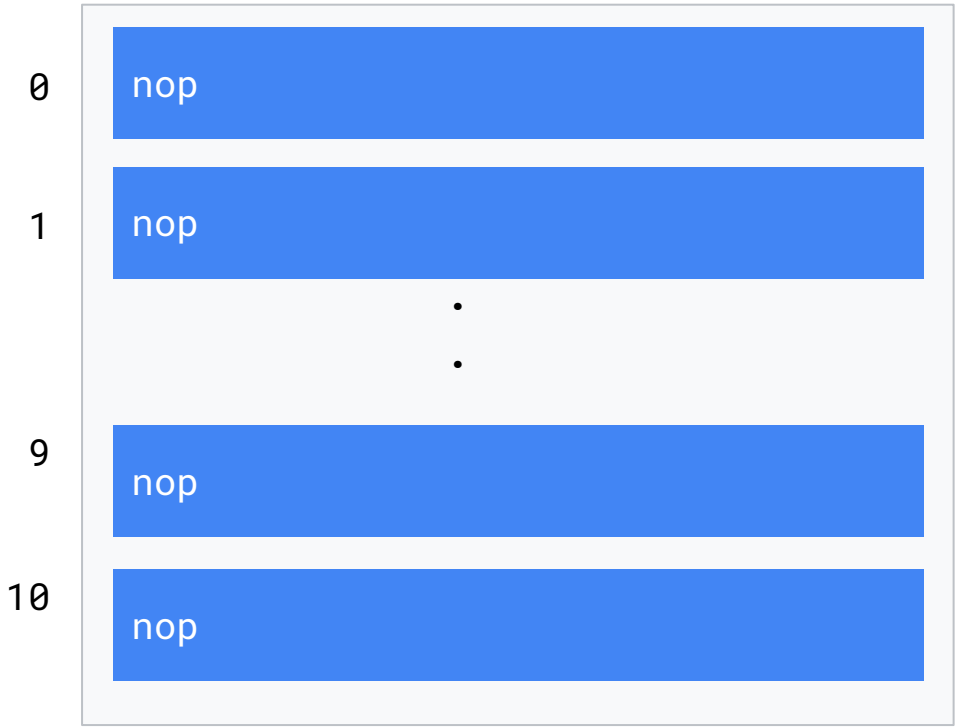
Performance impact 2: BPF LSM empty callbacks

```
int bpf_lsm_bprm_check_security(struct linux_binprm *bprm)
{
  return LSM_RET_DEFAULT(bprm_check_security);
}
```

called every time, even when there is no BPF program...

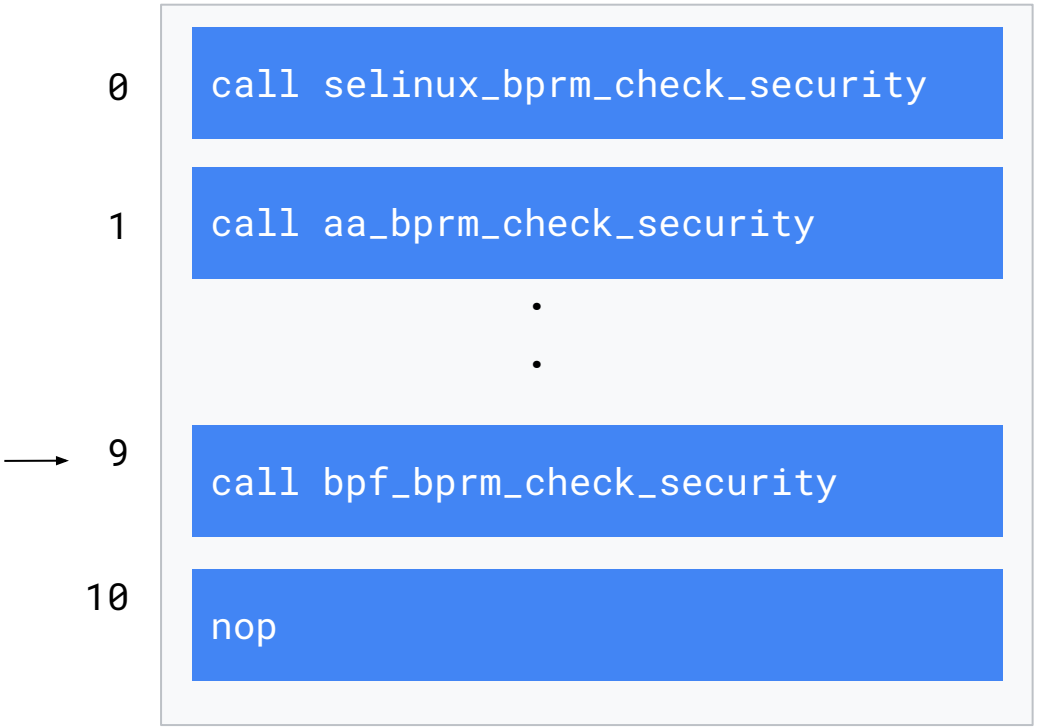
Proposed Improvement: Static calls

Slots for call instructions at compile time.



Proposed Improvement: Static calls

`bprm_check_security` call slots patched at `__init`, starting from the bottom



Implementation Details

```
struct security_hook_slot {
    struct static_call_key *key;
    void *trampoline;
    struct security_hook_list *hl;
    struct static_key *enabled_key;
};
```

Needed to define a STATIC_CALL

Information initialized by the LSM (using LSM_HOOK_INIT)

Static key to check if the slot is initialized.

Implementation Details

```
struct security_hook_list {
    struct security_hook_slot       *slots;
    union security_list_options     hook;
    const char                      *lsm;
    bool                            default_state;
};
```

Pointer to the slot assigned to the hook on LSM init

Hook implementation callback

State of the static key on init

Name of the owner LSM.

Implementation Details

```
#define LSM_HOOK_INIT(NAME, CALLBACK)          \
    {                                          \
        .slots = security_hook_slots.NAME,     \
        .hook = { .NAME = CALLBACK },          \
        .default_state = true                  \
    }

#define LSM_HOOK_INIT_DISABLED(NAME, CALLBACK)      \
    {                                               \
        .slots = security_hook_slots.NAME,          \
        .hook = { .NAME = CALLBACK },               \
        .default_state = false                      \
    }
```

```
#define __CALL_STATIC_INT(NUM, R, HOOK, ...)                                \
 if (static_branch_likely(&SECURITY_HOOK_ENABLED_KEY(HOOK, NUM))) {         \
   R = static_call(SECURITY_STATIC_SLOT(HOOK, NUM))(__VA_ARGS__);           \
   if (R != 0)                                                              \
      goto out;                                                             \
 }

#define call_int_hook(FUNC, IRC, ...) ({                                    \
  __label__ out;                                                            \
  int RC = IRC;                                                             \
  do {                                                                      \
    SECURITY_FOREACH_STATIC_SLOT(__CALL_STATIC_INT,RC, FUNC, __VA_ARGS__)   \
  } while(0);                                                               \
out:                                                                        \
  RC;
```

```
#define __CALL_STATIC_INT(NUM, R, HOOK, ...)                                \
 if (static_branch_likely(&SECURITY_HOOK_ENABLED_KEY(HOOK, NUM))) {         \
   R = static_call(SECURITY_STATIC_SLOT(HOOK, NUM))(__VA_ARGS__);           \
   if (R != 0)                                                              \
      goto out;                                                             \
 }

#define call_int_hook(FUNC, IRC, ...) ({                                      \
  __label__ out;                                                              \
  int RC = IRC;                                                               \
  do {                                                                        \
    SECURITY_FOREACH_STATIC_SLOT(__CALL_STATIC_INT,RC, FUNC, __VA_ARGS__)     \
  } while(0);                                                                 \
out:                                                                          \
  RC;
```

```
#define __CALL_STATIC_INT(NUM, R, HOOK, ...)                                \
 if (static_branch_likely(&SECURITY_HOOK_ENABLED_KEY(HOOK, NUM))) {         \
   R = static_call(SECURITY_STATIC_SLOT(HOOK, NUM))(__VA_ARGS__);           \
   if (R != 0)                                                              \
      goto out;                                                             \
 }

#define call_int_hook(FUNC, IRC, ...) ({                                      \
  __label__ out;                                                            \
  int RC = IRC;                                                             \
  do {                                                                      \
    SECURITY_FOREACH_STATIC_SLOT(__CALL_STATIC_INT,RC, FUNC, __VA_ARGS__)   \
  } while(0);                                                               \
out:                                                                        \
  RC;
```

```
<security_bprm_check>:
  [...]
    mov     %rdi,%rbp
    jmp     4c78 <security_bprm_check+0x18>

    call    4c70 <security_bprm_check+0x10>
        R_X86_64_PLT32     __SCT__security_static_slot_bprm_check_security_0-0x4

    test    %eax,%eax
    jne     4d15 <security_bprm_check+0xb5>
    jmp     4c8a <security_bprm_check+0x2a>
    mov     %rbp,%rdi

    call    4c82 <security_bprm_check+0x22>
        R_X86_64_PLT32     __SCT__security_static_slot_bprm_check_security_1-0x4
  [...]
```

Status

- Benchmarks results
  - 3% improvement in Redis
- Hooks that don't use call_{int, void}_hook are not covered yet
  - Can be covered with custom per-hook macros
  - No real hotpaths in these outlier hooks yet.

Future work 2:
Unintended side effects

```
security_inode_setxattr()
/*
 * SELinux and Smack integrate the cap call,
 * so assume that all LSMs supplying this call do so.
 */
ret = call_int_hook(inode_setxattr, 1, mnt_userns, dentry, name, value,
                    size, flags);
if (ret == 1)
            ret = cap_inode_setxattr(dentry, name, value, size, flags);
  if (ret)
   return ret;
  ret = ima_inode_setxattr(dentry, name, value, size);

  if (ret)
   return ret;
  return evm_inode_setxattr(mnt_userns, dentry, name, value, size);
}
```

bpf returns
LSM_RET_DEFAULT(inode_xattr)

capability check is skipped, security xattrs can be overwritten by unprivileged users

Fix side-effects...

- Default values returned by LSM_RET_DEFAULT can lead to subtle corner cases
- LSM programs may choose (in some cases, e.g. pure audit) to not make a decision
- Other side-effect issues:
  - `secid_to_secctx` broke Audit
  - `inode_copy_up_xattr` caused crashes

Ideas:

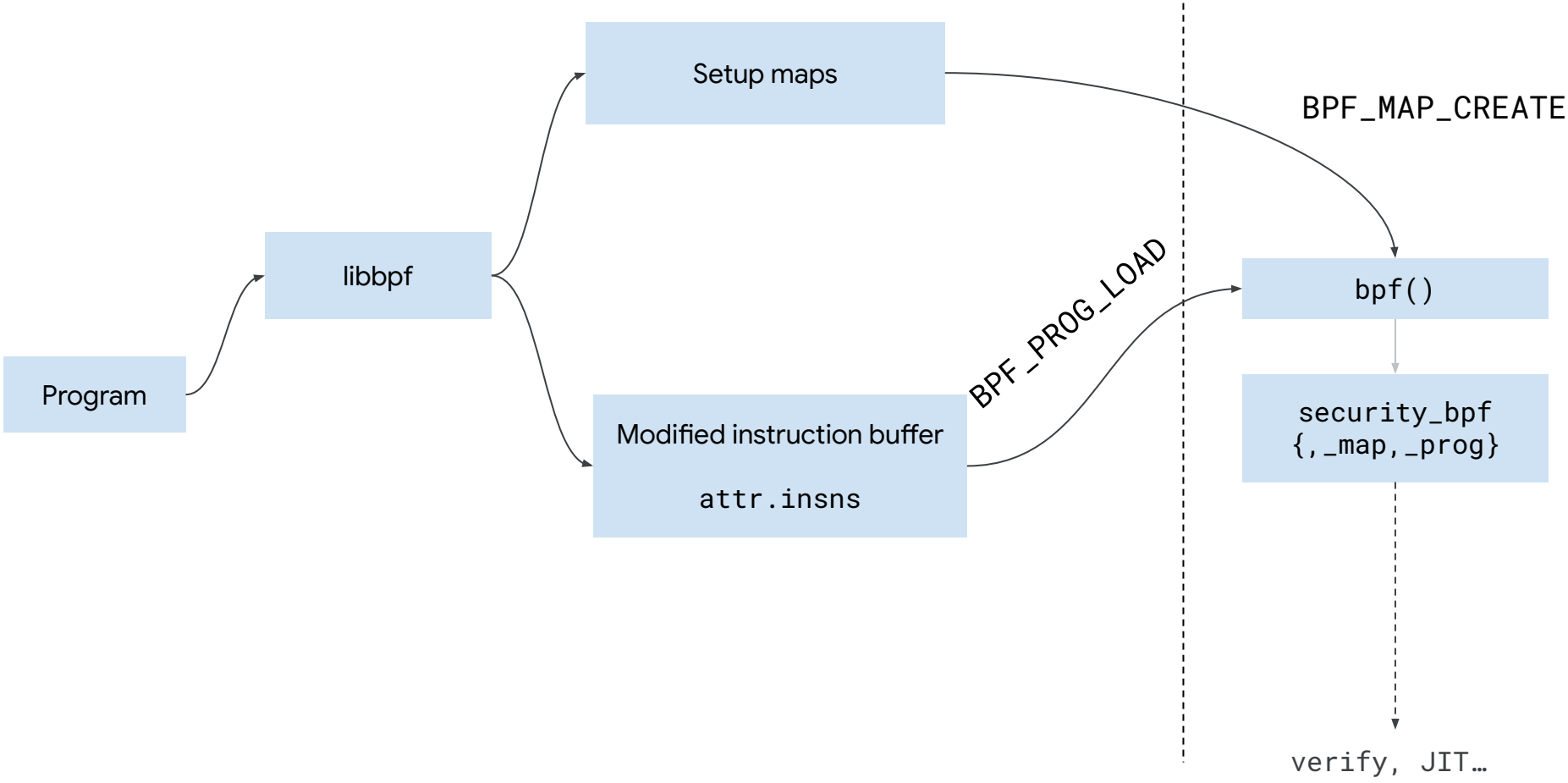
- A new error ENODECISION or use an existing error for no decision
- Static keys to enable / disable LSM hooks

Future work 3:
Program signatures

Goal

"This BPF program comes trustworthy source.."

Why can't the .o file be signed?



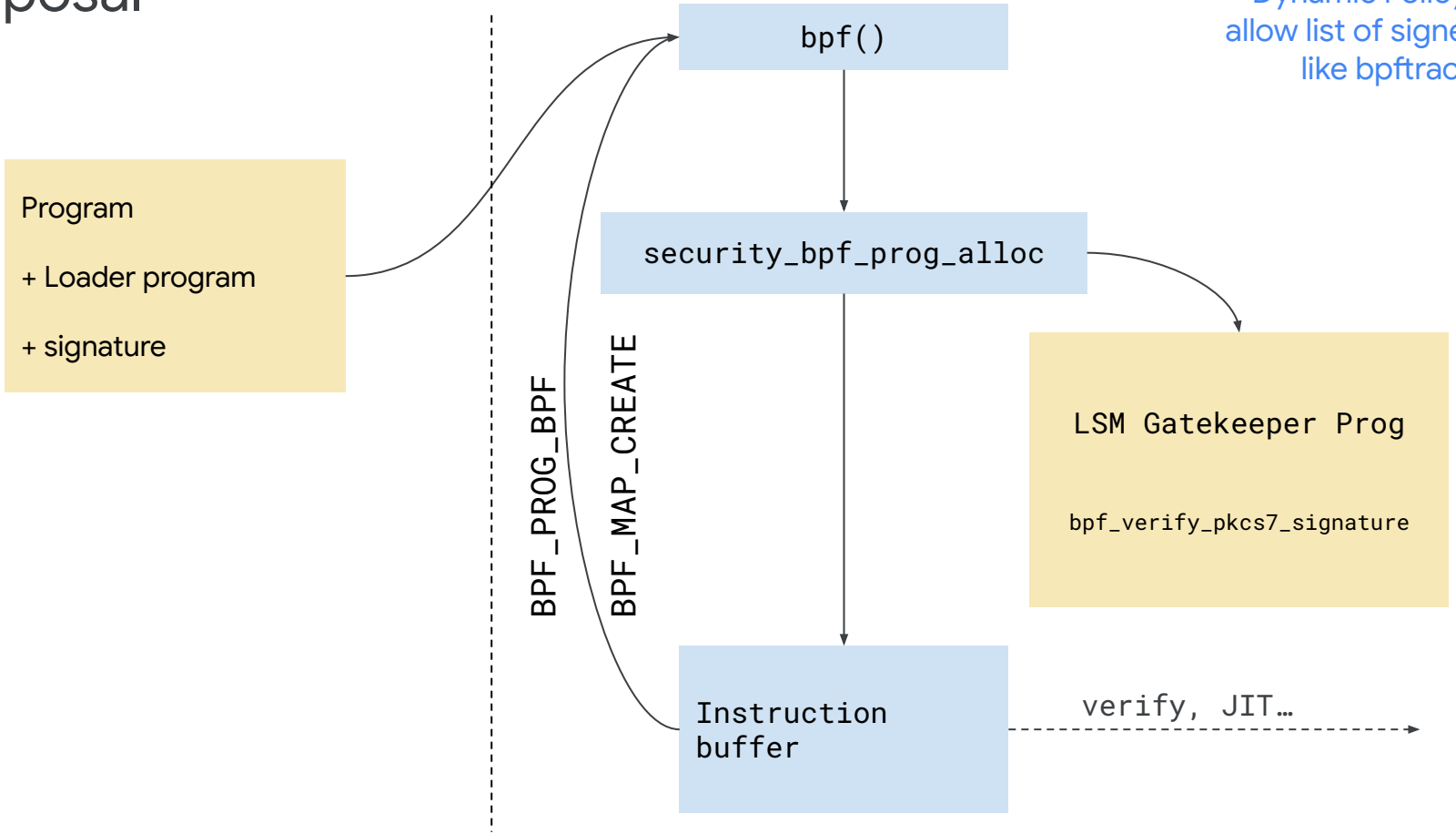
Loader Program

Trace of the operations performed by libbpf:

$$\textbf{Instructions}_{\textbf{loader}} = \text{Instructions}_{\text{syscall operations}} + \text{Instructions}_{\text{orignal}}$$

$\textbf{Instructions}_{\textbf{loader}}$ is stable and can be signed

Proposal


Dynamic Policy (e.g. allow list of signed tools like bpftrace)
Program
+ Loader program
+ signature
bpf()
security_bpf_prog_alloc
LSM Gatekeeper Prog
bpf_verify_pkcs7_signature
BPF_PROG_BPF
BPF_MAP_CREATE
Instruction buffer
verify, JIT…

Sample proposal: Example IMA

- IMA signs the programs (e.g. using `evmctl`)

- Signature blob is passed to the kernel via the `bpf` syscall like IMA wants it:

```
ima-ng sha1:9797edf8d0eed36b1cf92547816051c8af4e45ee
```

- IMA Gatekeeper program parses the signature

Future work 4:
xattr support

Simple prototype use case

“Just allow bpftrace to load BPF programs”

Simple domains

bpf: Can load BPF programs
xsetter: Can set xattrs

Simple domains

```
# xattr -l /usr/bin/bpftrace
security.domain: bpf
```

eBPF helper

```
ssize_t
bpf_getxattr(struct dentry *dentry,
             struct inode *inode,
             const char *name,
             void *value, int value__sz);
```

`bprm_committed_creds`

Get the xattr from the executable

Store security domain information in task blob

task_alloc

Transfer security domain information to child tasks

bpf_prog

Deny bpf syscalls for non-bpf domain tasks

`inode_setxattr`

Deny any attempt to set xattrs from outside xsetter domain

Contention

Can BPF LSM read all xattrs? only security.bpf or security.*

Future work 5:
LSM on ARM

Missing trampolines

- bpf_arch_text_poke
- Trampolines + FTrace
- Patches in flight...

Thank you!