PeX: A Permission Check Analysis Framework for Linux Kernel

Tong Zhang[*] Virginia Tech
Wenbo Shen[†] Zhejiang University
Dongyoon Lee Stony Brook University
Changhee Jung Purdue University
Ahmed M. Azab[‡] Samsung Research America
Ruowen Wang[‡] Samsung Research America

Abstract

Permission checks play an essential role in operating system security by providing access control to privileged functionalities. However, it is particularly challenging for kernel developers to correctly apply new permission checks and to scalably verify the soundness of existing checks due to the large code base and complexity of the kernel. In fact, Linux kernel contains millions of lines of code with hundreds of permission checks, and even worse its complexity is fast-growing.

This paper presents PeX, a static Permission check error detector for LinuX, which takes as input a kernel source code and reports any missing, inconsistent, and redundant permission checks. PeX uses KIRIN (Kernel InteRface based Indirect call aNalysis), a novel, precise, and scalable indirect call analysis technique, leveraging the common programming paradigm used in kernel abstraction interfaces. Over the inter-procedural control flow graph built by KIRIN, PeX automatically identifies all permission checks and infers the mappings between permission checks and privileged functions. For each privileged function, PeX examines all possible paths to the function to check if necessary permission checks are correctly enforced before it is called.

We evaluated PeX on the latest stable Linux kernel v4.18.5 for three types of permission checks: Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). PeX reported 36 new permission check errors, 14 of which have been confirmed by the kernel developers.

1 Introduction

Access control [38] is an essential security enforcement scheme in operating systems. They assign users (or processes) different access rights, called permissions, and enforce that only those who have appropriate permissions can access critical resources (e.g., files, sockets). In the kernel, access control is often implemented in the form of *permission checks* before the use of *privileged functions* accessing the critical resources.

Over the course of its evolution, Linux kernel has employed three different access control models: Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). *DAC* distinguishes privileged users (a.k.a., root) from unprivileged ones. The unprivileged users are subject to various permission checks, while the root bypasses them all [4]. Linux kernel v2.2 divided the root privilege into small units and introduced *Capabilities* to allow more fine-grained access control. From kernel v2.6, Linux adopted *LSM* in which various security hooks are defined and placed on critical paths of privileged operations. These security hooks can be instantiated with custom checks, facilitating different security model implementations as in SELinux [41] and AppArmor [3].

Unfortunately, for a new feature or vulnerability found, these access controls have been applied to the Linux kernel code in an ad-hoc manner, leading to *missing*, *inconsistent*, or *redundant* permission checks. Given the ever-growing complexity of the kernel code, it is becoming harder to manually reason about the mapping between permission checks and privileged functions. In reality, kernel developers rely on their own judgment to decide which checks to use, often leading to over-approximation issues. For instance, *Capabilities* were originally introduced to solve the "super" root problem, but it turns out that more than 38% of *Capabilities* indeed check `CAP_SYS_ADMIN`, rendering it yet another root [5].

Even worse, *there is no systematic, sound, and scalable way to examine whether all privileged functions (via all possible paths) are indeed protected by correct permission checks*. The lack of tools for checking the soundness of existing or new permission checks can jeopardize the kernel security putting the privileged functions at risk. For example, DAC, CAP and LSM introduce hundreds of security checks scattered over millions of lines of the kernel code, and it is an open problem to verify if all code paths to a privileged function encounter its corresponding permission check before reaching the function. Given the distributed nature of kernel development and the significant amount of daily updates, chances are that some

[*]This work was started when Tong Zhang interned at Samsung Research America, mentored by Wenbo Shen and Ahmed M. Azab.
[†]Corresponding author.
[‡]Now at Google.

parts of the code may miss checks on some paths or introduce the inconsistency between checks, weakening the operating system security.

This paper presents PeX, a static permission check analysis framework for Linux kernel. PeX makes it possible to soundly and scalably detect any missing, inconsistent and redundant permission checks in the kernel code. At a high level, PeX statically explores all possible program paths from user-entry points (e.g., system calls) to privileged functions and detects permission check errors therein. Suppose PeX finds a path in which a privileged function, say `PF`, is protected (preceded) by a check, say `Chk` in one code. If it is found that any other paths to `PF` bypass `Chk`, then it is a strong indication of a missing check. Similarly, PeX can detect inconsistent and redundant permission checks. While conceptually simple, it is very challenging to realize a sound and precise permission check error detection at the scale of Linux kernel.

In particular, there are two daunting challenges that PeX should address. First, Linux kernel uses indirect calls very frequently, yet its static call graph analysis is notoriously difficult. The latest Linux kernel (v4.18.5) contains 15.8M LOC, 247K functions, and 115K indirect callsites, rendering existing precise solutions (e.g., SVF [43]) unscalable. Only workaround available to date is either to apply the solutions unsoundly (e.g., only on a small code partition as with K-Miner [22]) or to rely on naive imprecise solutions (e.g., type-based analysis). Either way leads to undesirable results, i.e., false negatives (K-Miner) or positives (type-based one).

For a precise and scalable indirect call analysis, we introduce a novel solution called *KIRIN* (Kernel InteRface based Indirect call aNalysis), which leverages kernel abstraction interfaces to enable precise yet scalable indirect call analysis. Our experiment with Linux v4.18.5 shows that KIRIN allows PeX to detect many previously unknown permission check bugs, while other existing solutions either miss many of them or introduce too many false warnings.

Second, unlike Android which has been designed with the permission-based security model in mind [2], Linux kernel does not document the mapping between a permission check and a privileged function. More importantly, the huge Linux kernel code base makes it practically impossible to review them all manually for the permission check verification.

To tackle this problem, PeX presents a new technique which takes as input a small set of known permission checks and automatically identifies all other permission checks including their wrappers. Moreover, PeX's dominator analysis [31] automates the process of identifying mappings between permission checks and their potentially privileged functions as well. Our experiment with Linux kernel v4.18.5 shows that starting from a small set of well-known 3 DAC, 3 Capacities, and 190 LSM checks, PeX automatically (1) identifies 19, 16, and 53 additional checks, respectively, and (2) derives 9243 pairs of permission checks and privileged functions.

The contributions of this paper are summarized as follows:

Table 1: Commonly used permission checks in Linux.

Type	*Total #*	*Permission Checks*
DAC	3	generic_permission, sb_permission, inode_permission
Capabilities	3	capable, ns_capable, avc_has_perm_noaudit
LSM	190	security_inode_readlinkat, security_file_ioctl, etc..

- **New Techniques**: We proposed and implemented PeX, a static permission check analysis framework for Linux kernel. We also developed new techniques that can perform scalable indirect call analysis and automate the process of identifying permission checks and privileged functions.
- **Practical Impacts**: We analyzed DAC, Capabilities, and LSM permission checks in the latest Linux kernel v4.18.5 using PeX, and discovered 36 new permission check bugs, 14 of which have been confirmed by kernel developers.
- **Community Contributions**: We will release PeX as an open source project, along with the identified mapping between permission checks and privileged functions. This will allow kernel developers to validate their codes with PeX, and to contribute to PeX by refining the mappings with their own domain knowledge.

2 Background: Permission Checks in Linux

This section introduces DAC, Capabilities, and LSM in Linux kernel. Table 1 lists practically-known permission checks in Linux. Unfortunately, the full set is not well-documented.

2.1 Discretionary Access Control (DAC)

DAC restricts the accesses to critical resources based on the identity of subjects or the group to which they belong [36,46]. In Linux, each user is assigned a user identifier (uid) and a group identifier (gid). Correspondingly, each file has properties including the owner, the group, the `rwx` (read, write, and execute) permission bits for the owner, the group, and all other users. When a process wants to access a file, DAC grants the access permissions based on the process's uid, gid as well as the file's permission bits. For example in Linux, `inode_permission` (as listed in Table 1) is often used to check the permissions of the current process on a given inode. More precisely speaking, however, it is a wrapper of `posix_acl_permission`, which performs the actual check.

In a sense, DAC is a coarse-grained access control model. Under the Linux DAC design, the "root" bypasses all permission checks. This motivates fine-grained access control scheme—such as Capabilities—to reduce the attack surface.

2.2 Capabilities

Capabilities, since Linux kernel v2.2 (1999), enable a fine-grained access control by dividing the root privilege into small sets. As an example, for users with the `CAP_NET_ADMIN` capability, kernel allows them to use `ping`, without the need to grant the full root privilege. Currently, Linux kernel v4.18.5 supports 38 Capabilities including `CAP_NET_ADMIN`, CAP_SYS_ADMIN, and so on. Functions capable and ns_capable are the most commonly used permission checks for Capabilities (as listed in Table 1). Both determine whether a process has a particular capability or not, while ns_capable performs an additional check against a given user namespace. They internally use security_capable as the basic permission check.

Capabilities are supposed to be fine-grained and distinct [4]. However, due to the lack of clear scope definitions, the choice of specific Capability for protecting a privileged function has been made based on kernel developers' own understanding in practice. Unfortunately, this leads to frequent use of CAP_SYS_ADMIN (451 out of 1167, more than 38%), and it is just treated as yet another root [5]; grsecurity points out that 19 Capabilities are indeed equivalent to the full root [1].

2.3 Linux Security Module (LSM)

LSM [51], introduced in kernel v2.6 (2003), provides a set of fine-grained pluggable hooks that are placed at various security-critical points across the kernel. System administrators can register customized permission checking callbacks to the LSM hooks so as to enforce diverse security policies. The latest Linux kernel v4.18.5 defines 190 LSM hooks. One common use of LSM is to implement Mandatory Access Control (MAC) [8] in Linux (e.g., SELinux [40, 41], AppArmor [3]). MAC enforces more strict and non-overridable access control policies, controlled by system administrators. For example, when a process tries to read the file path of a symbolic link, security_inode_readlink is invoked to check whether the process has read permission to the symlink file. The SELinux callback of this hook checks if a policy rule can grant this permission (e.g., allow domain_a type_b:lnk_file read). It is worth noting that the effectiveness of LSM and its MAC mechanisms highly depend on whether the hooks are placed *correctly* and *soundly* at all security-critical points. If a hook is missing at any critical point, there is no way for MAC to enforce a permission check.

3 Examples of Permission Check Errors

This section illustrates different kinds of permission check errors, found by PeX and confirmed by the Linux kernel developers. We refer to those functions, that validate whether a process (a user or a group) has proper permission to do certain operations, as *permission checks*. Similarly, we define *privileged functions* to be those functions which only a privileged process can access and thus require permission checks.

3.1 Capability Permission Check Errors

Figure 1 shows real code snippets of Capability permission check errors in Linux kernel v4.18.5. Figure 1a shows the kernel function scsi_ioctl, in which sg_scsi_ioctl (Line 7) is safeguarded by two Capability checks, CAP_SYS_ADMIN and CAP_SYS_RAWIO (Line 5). To the contrary, scsi_cmd_ioctl in Figure 1b calls the same function sg_scsi_ioctl (Line 8) without any Capability check. These two functions share three similarities. First, both of them are reachable from the userspace by ioctl system call. Second, both call sg_scsi_ioctl with a userspace parameter, void __user *arg. Last, there is no preceding Capability check on all possible paths to them (though scsi_ioctl performs two checks).

```
1  int scsi_ioctl(struct scsi_device *sdev, int cmd,
   ↪  void __user *arg)
2  {
3    ...
4    case SCSI_IOCTL_SEND_COMMAND:
5      if (!capable(CAP_SYS_ADMIN) ||
       ↪  !capable(CAP_SYS_RAWIO))
6        return -EACCES;
7      return sg_scsi_ioctl(sdev->request_queue, NULL,
       ↪  0, arg);
8    ...
9  }
```

(a) sg_scsi_ioctl (Line 7) is called **with** CAP_SYS_ADMIN and CAP_SYS_RAWIO capability checks (Line 5). arg is user space controllable.

```
1  int scsi_cmd_ioctl(struct request_queue *q, ...,
   ↪  void __user *arg)
2  {
3    ...
4    case SCSI_IOCTL_SEND_COMMAND:
5      ...
6      if (!arg)
7        break;
8      err = sg_scsi_ioctl(q, bd_disk, mode, arg);
9      break;
10   ...
11   return err;
12 }
```

(b) sg_scsi_ioctl (Line 8) is called **without** capability checks. arg is user space controllable.

```
1  int sg_scsi_ioctl(struct request_queue *q, struct
   ↪  gendisk *disk, fmode_t mode, struct
   ↪  scsi_ioctl_command __user *sic)
2  {
3    ...
4    err = blk_verify_command(req->cmd, mode);
5    ...
6    return err;
7  }
8
9  int blk_verify_command(unsigned char *cmd, fmode_t
   ↪  mode)
10 {
11   ...
12   if (capable(CAP_SYS_RAWIO))
13     return 0;
14   ...
15   return -EPERM;
16 }
```

(c) sg_scsi_ioctl calls blk_verify_command, which checks CAP_SYS_RAWIO capability.

Figure 1: Capability check errors discovered by PeX.

The kernel is supposed to sanitize userspace inputs and check permissions to ensure that only users with appropriate permissions can conduct certain privileged operations. As SCSI (Small Computer System Interface) functions manipulate the hardware, they should be protected by Capabilities. At first glance, scsi_ioctl seems to be correctly protected (while scsi_cmd_ioctl misses two Capability checks).

However, delving into sg_scsi_ioctl ends up with a different conclusion. As shown in Figure 1c, sg_scsi_ioctl calls blk_verify_command, which in turn checks CAP_SYS_RAWIO. Considering all together, scsi_ioctl checks CAP_SYS_ADMIN once but CAP_SYS_RAWIO "twice", leading to a *redundant* permission check. On the other hand, scsi_cmd_ioctl checks

```
1  static int do_readlinkat(int dfd, const char __user
   ↪  *pathname, char __user *buf, int bufsiz)
2  {
3    ...
4    error = security_inode_readlink(path.dentry);
5    if (!error) {
6      touch_atime(&path);
7      error = vfs_readlink(path.dentry, buf, bufsiz);
8    }
9    ...
10 }
```

(a) Kernel LSM usage in system call `readlinkat`. `vfs_readlink` (Line 7) is protected by `security_inode_readlink` (Line 4). Both `pathname` and `buf` (Line 1 and Line 7) are user controllable.

```
1  int ksys_ioctl(unsigned int fd, unsigned int cmd,
   ↪  unsigned long arg)
2  {
3    ...
4    error = security_file_ioctl(f.file, cmd, arg);
5    if (!error)
6      error = do_vfs_ioctl(f.file, fd, cmd, arg);
7    ...
8  }
9
10 int xfs_readlink_by_handle(struct file *parfilp,
   ↪  xfs_fsop_handlereq_t *hreq)
11 {
12   ...
13   error = vfs_readlink(dentry, hreq->ohandle, olen);
14   ...
15 }
```

(b) Kernel LSM usage in system call `ioctl`. It calls `security_file_ioctl` (Line 4) to protect `do_vfs_ioctl` (Line 6). `hreq->ohandle` and `olen` are also user controllable.

Figure 2: LSM check errors discovered by PeX.

only `CAP_SYS_RAWIO`, resulting in a *missing* permission check for `CAP_SYS_ADMIN`. In particular, PeX detects this bug as an *inconsistent* permission check because the two paths disagree with each other, and further investigation shows that one is redundant and the other is missing.

3.2 LSM Permission Check Errors

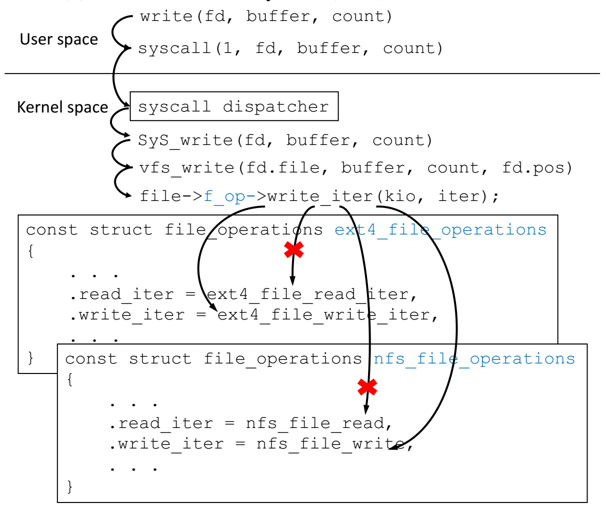
The example of LSM permission check errors is related to how LSM hooks are instrumented for two different system calls `readlinkat` and `ioctl`.

Figure 2a shows the LSM usage in the `readlinkat` system call. On its call path, `vfs_readlink` (Line 7) is protected by the LSM hook `security_inode_readlink` (Line 4) so that a LSM-based MAC mechanism, such as SELinux or AppArmor, can be realized to allow or deny the `vfs_readlink` operation.

Figure 2b presents two sub-functions for the system call `ioctl`. Similar to the above case, `ioctl` calls `ksys_ioctl`, which includes its own LSM hook `security_file_ioctl` (Line 4) before `do_vfs_ioctl` (Line 6). This is proper design, and there is no problem so far. However, it turns out that there is a path from `do_vfs_ioctl` to `xfs_readlink_by_handle` (Line 10), which eventually calls the same privileged function `vfs_readlink` (see Line 7 in Figure 2a and Line 13 in Figure 2b). While this function is protected by the `security_inode_readlink` LSM hook in `readlinkat`, that is not the case for the path to the function going through `xfs_readlink_by_handle`. The problem is that SELinux maintains separate 'allow' rules for `read` and `ioctl`. With the *missing* LSM `security_inode_readlink` check, a user only with

```
struct file_operations {
  ...
    ssize_t (*read_iter) (struct kiocb *, struct
    ↪  iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct
    ↪  iov_iter *);
    ...
}
```

(a) The Virtual File System (VFS) kernel interface.


User space
write(fd, buffer, count)
syscall(1, fd, buffer, count)
Kernel space
syscall dispatcher
SyS_write(fd, buffer, count)
vfs_write(fd.file, buffer, count, fd.pos)
file->f_op->write_iter(kio, iter);
const struct file_operations ext4_file_operations
{
. . .
.read_iter = ext4_file_read_iter,
.write_iter = ext4_file_write_iter,
. . .
}
const struct file_operations nfs_file_operations
{
. . .
.read_iter = nfs_file_read,
.write_iter = nfs_file_write,
. . .
}

(b) VFS indirect calls in Linux kernel.

Figure 3: Indirect call examples via the VFS kernel interface.

the 'ioctl allow rule' may exploit the `ioctl` system call to trigger the `vfs_readlink` operation, which should only be permitted by the different 'read allow rule'.

The above two Capability and LSM examples show how challenging it is to ensure correct permission checks. There are no tools available for kernel developers to rely on to figure out whether a particular function should be protected by a permission check; and, (if so) which permission checks should be used.

4 Challenges

This section discusses two critical challenges in designing static analysis for detecting permission errors in Linux kernel.

4.1 Indirect Call Analysis in Kernel

The first challenge lies in the frequent use of indirect calls in Linux kernel and the difficulties in statically analyzing them in a scalable and precise manner. To achieve a modular design, the kernel proposes a diverse set of abstraction layers that specify the common *interfaces* to different concrete implementations. For example, Virtual File System (VFS) [12] abstracts a file system, thereby providing a unified and transparent way to access local (e.g., `ext4`) and network (e.g., `nfs`) storage devices. Under this kernel programming paradigm, an abstraction layer defines an interface as a set of indirect function pointers while a concrete module initializes these pointers with its own implementations. For example, as shown in Figure 3a, VFS abstracts all file system operations in a *ker-*

nel interface struct file_operations that contains a set of function pointers for different file operations. When a file system is initialized, it initializes the VFS interface with the concrete function addresses of its own. For instance, Figure 3b shows that ext4 file system sets the write_iter function pointer to ext4_file_write_iter, while nfs sets the pointer to nfs_file_write.

However, kernel's large code base challenges the resolution of these numerous function pointers within kernel interfaces. For example, the kernel used in our evaluation (v4.18.5) includes 15.8M LOC, 247K functions, and 115K indirect callsites. This huge code base makes existing precise pointer analysis techniques [23–25, 35, 43] unscalable. In fact, Static Value Flow (SVF) [43], i.e., the state-of-the-art analysis that uses flow- and context-sensitive value flow for high precision, failed to scale to the huge Linux kernel. That is because SVF is essentially a whole program analysis, and its indirect call resolution thus requires tracking all objects such as functions, variables, and so on, making the value flow analysis unscalable to the large-size Linux kernel. In our experiment of running SVF for the kernel on a machine with 256GB memory, SVF was crashed due to an out of memory error[1].

Alternatively, one may opt for a simple "type-based" function pointer analysis, which would scale to Linux kernel. However, the type-based indirect call analysis would suffer from serious imprecision with too many *false* targets, because function pointers in the kernel often share the same type. For example, in Figure 3a, two function pointers read_iter and write_iter share the same function type. Type based pointer analysis will even link write_iter to ext4_file_read_iter falsely, which may lead to false permission check warnings.

PeX addresses this problem with a new kernel-interface aware indirect call analysis technique, detailed in §5.

4.2 The Lack of Full Permission Checks, Privileged Functions, and Their Mappings

The second challenge lies in soundly enumerating a set of permission checks and inferring correct mappings between permission checks and privileged functions in Linux kernel.

Though some commonly used permission checks for DAC, Capabilities, and LSM are known (Table 1), kernel developers often devise custom permission checks (wrappers) that internally use basic permission checks. Unfortunately, the complete list of such permission checks has never been documented. For example, ns_capable is a commonly used permission check for Capabilities, but it calls ns_capable_common and security_capable in sequence. It is the last security_capable that performs the actual capability check. In other words, all the others are "wrappers" of the "basic" permission check security_capable. This example shows how hard it is for a permission check analysis tool to keep up with all permission checks.

To make matters worse, Linux kernel has no explicit documentation that specifies which privileged function should be protected by which permission checks. This is different from Android [2], which has been designed with the permission-based security model in mind from the beginning. Take the Android LocationManager class as an example; for the getLastKnownLocation method, the API document states explicitly that permission ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION is required [7].

Unfortunately, existing *static* permission error checking techniques are not readily applicable in order to address these problems. Automated LSM hook verification [44] works only with clearly defined LSM hooks, which would miss many wrappers in the kernel setting. Many other tools require heavy manual efforts such as user-provided security rules [20, 56], authorization constraints [33], annotation on sensitive objects [21]. These manual processes are particularly error-prone when applied to huge Linux code base. Alternatively, some works such as [18, 32] rely on *dynamic* analysis. However, such run-time approaches may significantly limit the code coverage being analyzed, thereby missing real bugs.

Moreover, all of above existing works cannot detect permission checks soundly. Their inability to recognize permission checks or wrappers leads to missing privileged functions or false warnings for those that are indeed protected by wrappers. Since the huge Linux kernel code base makes it practically impossible to review them all manually, reasoning about the mapping is considered to be a daunting challenge.

In light of this, PeX presents a novel static analysis technique that takes as input a small set of known permission checks to identify their basic permission checks and leverages them as a basis for finding other permission check wrappers (§6.2). In addition, PeX proposes a dominator analysis based solution to automatically infer the mappings between permission checks and privileged functions (§6.3).

5 KIRIN Indirect Call Analysis

PeX proposes a precise and scalable indirect call analysis technique, called KIRIN (Kernel InteRface based Indirect call aNalysis), on top of the LLVM [27] framework. KIRIN is inspired by two key observations: (1) almost all (95%) indirect calls in the Linux kernel are originated from kernel interfaces (§4.1) and (2) the type of a kernel interface is preserved both at its initialization site (where a function pointer is defined) and at the indirect callsite (where a function pointer is used) in LLVM IR. For example in Figure 3b, the kernel interface object ext4_file_operations of the type struct file_operations is statically initialized where ext4_file_write_iter is assigned to the field of write_iter. For the indirect call site file→f_op→write_iter, one can identify that f_op is of the type struct file_operations and

[1]SVF internally uses LLVM SparseVectors to save memory overhead by only storing the set bits. However, it still blows up both the memory and the computation time due to the expensive insert, expand and merge operations.

```
1  @ext4_file_operations = dso_local local_unnamed_addr
   ↪  constant %struct.file_operations {
2  %struct.module* null,
3  i64 (%struct.file*, i64, i32)* @ext4_llseek,
4  i64 (%struct.file*, i8*, i64, i64*)* null,
5  i64 (%struct.file*, i8*, i64, i64*)* null,
6  i64 (%struct.kiocb*, %struct.iov_iter*)*
   ↪  @ext4_file_read_iter,
7  i64 (%struct.kiocb*, %struct.iov_iter*)*
   ↪  @ext4_file_write_iter,
```

(a) LLVM IR of `ext4_file_operations` initialization.

```
1  %25 = load %struct.file_operations*,
   ↪  %struct.file_operations** %f_op, align 8
2  %write_iter.i.i = getelementptr inbounds
   ↪  %struct.file_operations,
   ↪  %struct.file_operations* %25, i64 0, i32 5
3  %26 = load i64 (%struct.kiocb*, %struct.iov_iter*)*,
   ↪  i64 (%struct.kiocb*, %struct.iov_iter*)**
   ↪  %write_iter.i.i, align 8
4  %call.i.i = call i64 %26(%struct.kiocb* nonnull
   ↪  %kiocb.i, %struct.iov_iter* nonnull %iter.i) #10
```

(b) LLVM IR of callsite `file→f_op→write_iter` in `vfs_write`.

Figure 4: Indirect callsite resolution for `vfs_write`.

infer that `ext4_file_write_iter` is one of potential call targets. Based on this observation, PeX first collects indirect call targets at kernel interface initialization sites (§5.1) and then resolves them at indirect callsites (§5.2).

5.1 Indirect Call Target Collection

In Linux kernel, a kernel interface is often defined in a `C struct` comprised of function pointers (§4.1): e.g., `struct file_operations` in Figure 3a. Many kernel interfaces (`C structs`) are *statically* allocated and initialized as with `ext4_file_operations` and `nfs_file_operations` in Figure 3b. Some interfaces may be *dynamically* allocated and initialized at run time for reconfiguration.

For the former, KIRIN scans all Linux kernel code linearly to find all statically allocated and initialized `struct` objects with function pointer fields. Then, for each `struct` object, KIRIN keep tracks of which function address is assigned to which function pointers field using an offset as a key for the field. For instance, Figure 4a shows the LLVM IR of statically initialized `ext4_file_operations`. KIRIN finds that the kernel interface type is `struct file_operations` (Line 1), and `ext4_file_write_iter` is assigned to the 5th field `write_iter` (Line 7). Therefore, KIRIN figures out that `write_iter` may point to `ext4_file_write_iter`, not `ext4_file_read_iter` (even though they have the same function type).

For the rest dynamically initialized kernel interfaces, KIRIN performs a data flow analysis to collect any assignment of a function address to the function pointer inside a kernel interface. KIRIN's field-sensitive analysis allows the collected targets to be associated with the individual field of a kernel interface.

```
1  struct usb_driver* driver =
   ↪  container_of(intf->dev.driver, struct
   ↪  usb_driver, drvwrap.driver);
2  retval = driver->unlocked_ioctl(intf,
   ↪  ctl->ioctl_code, buf);
```

(a) C code of a `container_of` usage, followed by an indirect call.

```
1  #define container_of(ptr, type, member) ({                \
2      void *__mptr = (void *)(ptr);                         \
3      ((type *)(__mptr - offsetof(type, member))); })
4  %unlocked_ioctl = getelementptr inbounds i8*, i8**
   ↪  %add.ptr76, i64 3
```

(b) Original `container_of` and the LLVM IR for the callsite.

```
1  #define container_of(ptr, type, member) ({                \
2      type* __res;                                          \
3      void* __mptr = ((void *)((void*)(ptr) -               \
       ↪  offsetof(type, member)));                          \
4      memcpy(&__res, &__mptr, sizeof(void*));               \
5      (__res);})
6  %unlocked_ioctl = getelementptr inbounds
   ↪  %struct.usb_driver, %struct.usb_driver* %20, i64
   ↪  0, i32 3
```

(c) Modified `container_of` and the LLVM IR for the callsite.

Figure 5: Fixing `container_of` missing struct type problem.

5.2 Indirect Callsite Resolution

KIRIN stores the result of the above first pass in a key-value map data structure in which the key is a pair of kernel interface type and an offset (a field), and the value is a set of call targets. At each indirect callsite, KIRIN retrieves the type of a kernel interface and the offset from LLVM IR, looks up the map using them as a key, and figures out the matched call targets. For example, Figure 4b shows the LLVM IR snippet in which an indirect call `file→f_op→write_iter` is made inside of `vfs_write`. When an indirect call is made (Line 4), KIRIN finds that the kernel interface type is `struct file_operations` (Line 1) and the offset is 5 (Line 2). In this way, KIRIN reports that `ext4_file_write_iter` (assigned at Line 7 in Figure 4a) is one of potential call targets that are indirectly called by dereferencing `write_iter`.

When applying KIRIN to Linux kernel, we found in certain callsites, the kernel interface type is not presented in the LLVM IR, making their resolution impossible. For example, the macro `container_of` is commonly used in order to get the starting address of a `struct` object by using a pointer to its own member field. Figure 5a shows an example of using `container_of` (Line 1). It calculates the starting address of `usb_driver` through its own member `drvwrap.driver`. Based on the address, the code at Line 2 makes an indirect call by using a function pointer `unlocked_ioctl` that is another member of the `struct usb_driver` object.

Figure 5b shows the original macro `container_of` (Lines 1-3) and resulting LLVM IR (Line 4). The problem of this macro is that it involves a pointer manipulation, the LLVM IR of which voids the `struct` type information, i.e., the second argument of the macro. To solve this problem, KIRIN redefines `container_of` in a way that the `struct` type is preserved in the LLVM IR (on which KIRIN works), as in Figure 5c (Lines 1-5). This adds back the kernel interface type struct.usb_driver in the LLVM IR (Line 6), thereby enabling KIRIN to infer the correct type of driver and resolve the targets for unlocked_ioctl.

Our experiment (§7.2) shows that KIRIN resolves 92% of total indirect callsites for allyesconfig. PeX constructs a more sound (less missing edges) and precise (less false edges) call graph than other existing workarounds (e.g., [22]).

6 Design of PeX

Figure 6 shows the architecture of PeX. It takes as input kernel source code (in the LLVM bitcode format) and common permission checks (Table 1), analyzes and reports all detected permission check errors, including missing, inconsistent, and redundant permission checks. In addition, PeX produces the mapping of permission checks and privileged functions, which has not been formally documented.

At a high-level, PeX first resolves indirect calls with our new technique called KIRIN (§5). Next, PeX builds an augmented call graph—in which indirect callsites are connected to possible targets—and cuts out only the portion reachable from user space (§6.1). Based on the partitioned call graph, PeX then generates the interprocedural control flow graph (ICFG) where each callsite is connected to the entry and the exit of the callee [17]. Then, starting from a small set of (user-provided) permission checks, PeX automatically detects their wrappers (§6.2). After that, for a given permission check, PeX identifies its potentially privileged functions on top of the ICFG (§6.3), followed by a heuristic-based filter to prune obviously non-privileged functions (§6.4). Finally, for each privileged function, PeX examines all user space reachable paths to it to detect any permission checks error on the paths (§6.5). The following section describes these steps in detail.

6.1 Call Graph Generation and Partition

PeX generates the call graph leveraging the result of KIRIN (§5), and then partitions it into two groups.

User Space Reachable Functions: Starting from functions with the common prefix SyS_ (indicating system call entry points), PeX traverses the call graph, marks all visited functions, and treats them as user space reachable functions. The user reachable functions in this partition are investigated for possible permission check errors.

Kernel Initialization Functions: Functions that are used only during booting are collected to detect redundant checks. The Linux kernel boots from the start_kernel function, and calls a list of functions with the common prefix __init. PeX performs multiple call graph traversals starting from start_kernel and each of the __init functions to collect them.

Other functions such as IRQ handlers and kernel thread functions are not used in later analysis since they cannot be directly called from user space. The partitioned call graph serves as a basis for building an interprocedural control flow graph (ICFG) [31] used in the inference of the mapping between permission checks and privileged functions (§6.3).

6.2 Permission Check Wrapper Detection

Sound and precise detection of permission check errors requires a complete list of permission checks, but they are not readily available (§4.2). One may name some commonly used permission checks, as in Table 1. However, they are often the wrapper of basic permission checks, which actually perform the low-level access control, and even worse there could be other wrappers of the wrapper.

PeX solves this by automating the process of identifying all permission checks including wrappers. PeX takes an incomplete list of user-provided permission checks as input. Starting from them, PeX detects basic permission checks, by performing the *forward call graph slicing* [26, 37, 45] over the augmented call graph. For a given permission check function, PeX searches all call instructions inside the function for the one that passes an argument of the function to the callee. In other words, PeX identifies the callees of the permission check function which take its *actual* parameter as their own *formal* parameter. Similarly, PeX then conducts *backward call graph slicing* [26, 37, 45] from these basic permission checks to detect the list of their wrappers. PeX refers to only those callers that pass permission parameters as wrappers, excluding other callers just using the permission checks.

Figure 7 shows an example of the permission check wrapper detection. Given a known permission check ns_capable (Lines 10-13), PeX first finds security_capable (Line 4) as a basic permission check, and then based on it, PeX detects another permission check wrapper has_ns_capability (Lines 14-20). Note that the parameter cap is passed from both the parents ns_capable_common and has_ns_capability to the child security_capable; and the result of security_capable is returned to them. Our evaluation (§7.3) shows that based on 196 permission checks in Table 1, PeX detects 88 wrappers.

6.3 Privileged Function Detection

It is important to understand the mappings between permission checks and privileged functions for effective detection of any permission check errors therein. However, the lack of clear mapping in Linux kernel complicates the detection of permission check errors (§4.2).

To address this problem, PeX leverages an interprocedural *dominator* analysis [31] that can automatically identify the privileged functions protected by a given permission check. PeX conservatively treats all targets (callees) of those call instructions, that are dominated by each permission check (§6.2) on top of the ICFG (§6.1), as its *potential* privileged functions. The rationale behind the dominator analysis is based on the following observation: since there is no single path that allows the dominated call instruction to be reached without visiting the dominator (i.e., the permission check),

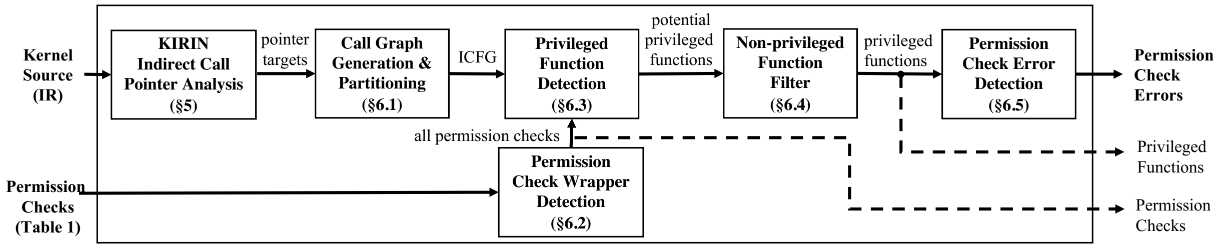

Kernel Source (IR)
KIRIN Indirect Call Pointer Analysis (§5)
pointer targets
Call Graph Generation & Partitioning (§6.1)
ICFG
Privileged Function Detection (§6.3)
potential privileged functions
Non-privileged Function Filter (§6.4)
privileged functions
Permission Check Error Detection (§6.5)
Permission Check Errors
all permission checks
Permission Checks (Table 1)
Permission Check Wrapper Detection (§6.2)
Privileged Functions
Permission Checks

Figure 6: PeX static analysis architecture. PeX takes as input kernel source code and permission checks, and reports as output permission check errors. PeX also produces mappings between identified permission checks and privileged functions as output.

```
static bool ns_capable_common(struct user_namespace
↪  *ns, int cap, bool audit)
{
    ....
    capable = audit ?
        security_capable(current_cred(), ns, cap) :
        ↪
        security_capable_noaudit(current_cred(), ns,
        ↪  cap);
    if (capable == 0)
        return true;
    return false;
}
bool ns_capable(struct user_namespace *ns, int cap)
{
    return ns_capable_common(ns, cap, true);
}
bool has_ns_capability(struct task_struct *t,
               struct user_namespace *ns, int cap)
{
    ...
    ret = security_capable(__task_cred(t), ns, cap);
    ...
}
```

Figure 7: Permission check wrapper examples.

Algorithm 1 Privileged Function Detection

```
INPUT:
    $pcfuncs$ - all permission checking functions
OUTPUT:
    $pvfuncs$ - privileged functions
 1: procedure PRIVILEGED FUNCTION DETECTION
 2:     for $f \leftarrow pcfuncs$ do
 3:         for $u \leftarrow User(f)$ do
 4:             $CallInst \leftarrow CallInstDominatedBy(u)$    ▷ Inter-procedural analysis, for full program path
 5:             $callee \leftarrow getCallee(CallInst)$
 6:             $pvfuncs.insert(callee)$
 7:         end for
 8:     end for
 9:     return $pvfuncs$
10: end procedure
```

the callee is likely to be the one that should be protected by the check on all paths [2].

Algorithm 1 shows how PeX uses the dominator analysis to find potential privileged functions `pvfuncs` for a given list of permission check functions `pcfuncs`. For each permission check function `f` (Line 2), PeX finds all users of `f`, i.e., the callsite invoking `f` (Line 3). For each user (callsite) `u`, PeX performs interprocedural dominator analysis over the ICFG to find all dominated call instructions (Line 4). All their callees are then added to `pvfuncs` (Lines 5-6).

Note that the call graph generated by KIRIN (§5) has resolved most of the indirect calls, which allows PeX to perform—on top of the resulting ICFG—more sound privileged function detection. For example, our experiment (§7.3) shows that KIRIN can identify `ecryptfs_setxattr` (reachable via indirect calls over the ICFG) as a privileged function and detect its missing permission check bug (Table 6, LSM-17). Note that if some other unsound workaround such as [22] had been used, this bug could not have been detected.

[2]This does not necessarily mean that the permission check dominates all call instructions of ICFG which invoke the resulting privileged function. As long as some call instructions are dominated by the check, their callees are treated as privileged functions.

6.4 Non-privileged Function Filter

The conservative approach in §6.3 may lead to too many potential privileged functions. In this step, PeX applies heuristic-based filters to prune out false privileged functions. In the current prototype, the filter contains a set of kernel library functions which are not privileged functions, e.g., `kmalloc`, `strcmp`, `kstrtoint`. Though PeX is currently designed to avoid false negatives (and thus leverages a small set of library filters only), one can add more aggressive filters to purge more false privileged functions. With releasing PeX, we expect a good opportunity for the kernel development community to contribute to the design of non-privileged function filters where domain knowledge is helpful.

6.5 Permission Check Error Detection

This last step is validating the use of privileged functions to detect any potential permission check errors. For a given mapping between a permission check and a privileged function, PeX performs a backward traversal of the ICFG, starting from the privileged functions with the corresponding permission check in mind. Note that PeX validates every possible path to each privileged function of interest.

Algorithm 2 shows PeX's permission check error detection algorithm. Recall that PeX treats user reachable kernel functions and kernel initialization functions separately and detects different forms of errors (§6.1). Lines 2-12 shows how PeX detects missing, redundant, and inconsistent checks in user reachable kernel functions. For each privileged function `f` (Line 5) in a mapping, PeX finds all possible paths `allpath` from user entry points to that privileged function `f` over the ICFG (Line 6). Line 7-18 checks each path `p` for the preceding permission check function, the lack of which should be reported as a bug. If the call to the privileged function (`pvcall`) is not preceded by the corresponding permission check func-

Algorithm 2 Permission Check Error Detection

```
INPUT:
    pc − pv - permission check function to privileged function mapping
    pcfuncs - all permission check functions
    kinitfuncs - kernel init functions
1:  procedure PERMISSION CHECK ERROR DETECTION
2:      for pair ← pc − pv do
3:          pvfuncs ← pair.pv                          ▷ privileged functions
4:          pcfunc ← pair.pc                           ▷ permission check functions
5:          for f ← pvfuncs do
6:              allpath ← getAllPathUseFunc(f)   ▷ get all user reachable paths that
    call the privileged function f
7:              for p ← allpath do
8:                  pvcall ← PrivilegeFunctionCallInPath(p)
9:                  if pvcall not Preceded by pcfunc then
10:                     if pvcall not Preceded by any pcfuncs then
11:                         report(p)                  ▷ Report missing checks
12:                     else
13:                         report(p)                  ▷ Report inconsistent check
14:                     end if
15:                 else if pvcall Preceded by multiple same pcfunc then
16:                     report(p)                      ▷ Report redundant checks
17:                 end if
18:             end for
19:         end for
20:     end for
21:     for f ← kinitfuncs do
22:         if f uses any pcfuncs then
23:             report(f)          ▷ Report unnecessary checks during kernel boot
24:         end if
25:     end for
26: end procedure
```

tion (`pcfunc`) and any other check functions (those in `pcfuncs`) over a given path `p`, then PeX reports a missing check (Lines 6-7). And if `pvcall` is preceded not by the corresponding check (`pcfunc`) but other check in `pcfuncs`, PeX reports an inconsistent check. Finally, if PeX discovers that `pvcall` is indeed preceded by `pcfunc` checks but multiple times, then it reports a redundant check (Lines 15-17). Besides, Lines 21-25 shows how PeX detects redundant checks in kernel initialization functions. As `kinitfuncs` includes a conservative list of functions that can only be executed during booting (thus obviating the need of any checks), all detected permission checks are marked as redundant (Lines 22-24).

7 Implementation and Evaluation

PeX was implemented using LLVM [27]/Clang-6.0. It contains about 7K lines of C/C++ code. Clang was modified to preserve the kernel interface type at allocation/initialization sites by using an *identified struct* type instead of using unnamed *literal struct* type. We also automated the generation of the single-file whole vmlinux LLVM bitcode `vmlinux.bc` using `wllvm` [13]. This avoids building each kernel module separately or changing kernel build infrastructures, as observed in prior kernel static analysis works [22, 49]. We evaluated PeX on the latest stable Linux kernel v4.18.5. In summary, KIRIN resolves 86%–92% of indirect callsites depending on its compilation configurations. PeX reported 36 permission check errors warnings to the Linux community, 14 of which have been confirmed as real bugs.

Table 2: Input Statistics for Kernel v4.18.5.

	defconfig	allyesconfig
# of yes(=y) config	1284	9939
# of compiled LOC	2,414,772	15,881,692
vmlinux size	481 MB	3.8 GB
vmlinux.bc size	387 MB	3.3 GB
# of total functions	42,264	247,465
# of syscall entries	857	1,027
# of init functions	1,570	9,301
# of indirect callsites (ICS)	20,338	115,537

Table 3: Indirect Call Pointer Analysis.

	defconfig			allyesconfig		
	KIRIN	TYPE	KM	KIRIN	TYPE	KM
% of ICS resolved	86	100	1	92	100	na
# of avg target	3.6	10K	3.6	6.2	81K	na
analysis time (min)	1	1	9,869	6.6	1	na

7.1 Evaluation Methodology

We evaluated PeX with two different kernel configurations: (1) `defconfig`, the (commonly-used) default configuration, and (2) `allyesconfig` with all non-conflict configuration options enabled. The use of `allyesconfig` not only stress-tests PeX (including KIRIN) with a larger code base than `defconfig`, but also covers the majority of kernel code, allowing PeX to detect more bugs. In addition, we used 3 DAC, 3 Capabilities, and 190 LSM permission checks(Table 1) as input permission checks, from which PeX finds other wrappers. For the non-privileged function filter, we collected 1827 library functions from `lib` directory in the kernel source code. All experiments were carried out on a machine running Ubuntu 16.04 with two Intel Xeon E5-2650 2.20GHz CPU and 256GB DRAM.

7.2 Evaluation of KIRIN

We compared the effectiveness and efficiency of KIRIN with type-based approach and SVF-based K-Miner approach.

K-Miner [22] works around the scalability problem in SVF by analyzing the kernel on a per system call basis, rather than taking the entire kernel code for analysis. K-Miner generates a (small-size) partition of kernel code which can be reached from a given system call, and (unsoundly) applies SVF for that partition. For comparison, we took K-Miner's implementation from the github [6] and added the logic to count the number of resolved indirect callsites and the average number of targets per callsite. As K-Miner was originally built on LLVM/Clang-3.8.1, we recompiled the same kernel v4.18.5 using `wllvm` with the same kernel configurations.

Table 3 summaries evaluation results of KIRIN, comparing it to the type-based approach and K-Miner approach in terms of the percentage of indirect callsite (ICS) resolved, the average number of targets per ICS, and the total analysis time.

7.2.1 Resolution Rate

For K-Miner, we observe somewhat surprising results: it resolves only 1% of all indirect callsites. After further investigation, we noticed that SVF runs on each partition whose code base is smaller than the whole kernel, its analysis scope is significantly limited and unable to resolve function pointers in other partitions, leading to the poor resolution rate.

Besides, we found out that K-Miner does not work for `allyesconfig` which contains a much larger code base than `defconfig`. Note that K-Miner evaluated its approach only for `defconfig` in the original paper [22]. The K-Miner approach turns out to be not scalable to handle `allyesconfig` which ends up encountering out of memory error even for analyzing a single system call.

7.2.2 Resolved Average Targets

For KIRIN, the number of average indirect call targets per resolved indirect callsite is much smaller than that of the type-based approach as shown in the second row of Table 3. The reason is that the type-based approach classifies all functions (not only address-taken functions) into different sets based on the function type. This implies that all functions in the set are regarded as possible call targets of that function pointer. For example, as shown in Figure 3a, two functions `ext4_file_read_iter` and `ext4_file_write_iter` share the same type. As a result, the type-based approach incorrectly identifies both functions as possible call targets of the function pointer `f_ops→write_iter`.

7.2.3 Analysis Time

The total analysis times of each ICS resolution approach are shown in the last row of Table 3. As expected, the type-based approach is the fastest, finishing analysis in 1 minute for both configurations. KIRIN runs slower than the type-based approach. However, the analysis time of KIRIN (≈1 minute) is comparable to that of the type-based approach for `defconfig`, while KIRIN takes 6.6 minutes for `allyesconfig`.

For a fair comparison with K-Miner, care must be taken when we collect its indirect call analysis time. For a given system call, we measured K-Miner's running time from the beginning until it produces the SVF point-to result, which does not include the later bug detection time. To obtain the total analysis time of K-Miner, we summed up the running times of all system calls. The result shows that SVF based K-Miner takes about 9,869 minutes to finish analyzing all system calls of `defconfig`, which is much slower than KIRIN's.

7.3 PeX Result

Table 4 summarizes PeX's intermediate program analyses. As `allyesconfig` subsumes `defconfig` in static analysis, we focus on discussing `allyesconfig` results here. Overall, PeX finishes all analyses within a few hours and reports about two thousand groups of warnings, which are classified by privileged functions. One may implement a multi-threaded version of PeX to further reduce the analysis time.

Given the small number of input DAC, CAP, and LSM permission checks (3, 3, and 190 each), PeX's permission check

Table 4: PeX Results.

	defconfig			allyesconfig		
	DAC	**CAP**	**LSM**	**DAC**	**CAP**	**LSM**
# of input checks	3	3	190	3	3	190
# of detected wrappers	11	13	34	19	16	53
# of priv func detected	174	869	2030	631	3770	10915
# of priv func after filter	116	582	1635	537	3245	10260
# of warnings grouped by priv func	72	210	853	221	850	1017
total time (min)	6	8	11	83	247	169

Table 5: Comparison of PeX warnings when used with different indirect call analyses.

	defconfig				allyesconfig			
	DAC	**CAP**	**LSM**	Bugs	**DAC**	**CAP**	**LSM**	Bugs
KIRIN	72	210	853	21	221	850	1017	36
TYPE	218	348	1319	21	164	964	4364	19 (PeX Timeout)
KM	54	196	241	6	na	na	na	na (SVF Timeout)

detection (§6.2) was able to identify 19, 16 and 53 permission check wrappers. For example, PeX detects wrappers such as `nfs_permission` and `may_open` for DAC; `sk_net_capable` and `netlink_capable` for Capabilities; and `key_task_permission` and `__ptrace_may_access` for LSM.

Table 4 also shows the number of potentially privileged functions detected by PeX (§6.3) and the number of remaining privileged functions after kernel library filtering (§6.4). We found that there are typically 1-to-1 or 2-to-1 mapping between permission checks and privileged functions. Overall, PeX reports 221, 850, and 1017 warnings (grouped by privileged functions) for DAC, CAP, and LSM, respectively.

Table 6 shows the list of 36 bugs we reported, 14 of which have been confirmed by Linux kernel developers. Kernel developers ignored some bugs and decided not to make changes because they believe that the bugs are not exploitable. We discuss them in detail in §7.5.

Comparison. To highlight the effectiveness of KIRIN, we repeated the end-to-end PeX analysis using type-based (PeX+TYPE) and K-Miner-style (PeX+KM) indirect call analyses. Table 5 shows the resulting number of warnings and detected bugs when the 36 bugs— shown in Table 6—are used as an oracle for false negative comparison.

For `allyesconfig`, PeX+TYPE and PeX+KM could not complete the analysis within the 12-hour experiment limit. PeX+TYPE generated too many (false) edges in ICFG and suffered from path explosion during the last phase of PeX analysis; only 19 bugs were reported before the timeout. In the mean time, PeX+KM timed out on an earlier pointer analysis phase, thereby failing to report any bug.

When `defconfig` is used for comparison, PeX+TYPE and PeX+KM were able to complete the analysis. In this setting, PeX+KIRIN (original) and PeX+TYPE both report 21 bugs (a subset of 36 bugs detected with `allyesconfig`). Though PeX+TYPE can capture them all (as type-based analysis is sound yet imprecise), it generates up to 3x more warnings, placing a high burden on the users side for their manual review. On the other hand, as an unsound solution, PeX+KM produces a limited number of warnings, resulting in the detection of only 6 bugs missing the rest.

7.4 Manual Review of Warnings

The manual review process of reported warnings is to determine whether a privileged function identified by PeX (§6.3) is a *true* privileged function or not. As long as one can confirm that a function is indeed privileged, reported warnings regarding its missing, inconsistent, and redundant permission checks should be *true positives* from PeX's point of view.

Though kernel developers with domain knowledge may be able to discern them with no complication, we (as a third-party) try to understand whether a given function can be used to access critical resources (e.g., device, file system, etc.). As a result, we conservatively reported 36 bug warnings to the community; we suspect that there could be more true warnings missed due to our lack of domain knowledge. We plan to release PeX and the list of potential privileged functions, hoping kernel developers will contribute to identify privileged functions and fix more true permission errors.

Certain static paths reported by PeX may not be feasible dynamically during program execution, resulting in false positives. One may devise a solution solving path constraints as in symbolic execution engines [16] to address this problem, PeX currently does not do so.

7.5 Discussion of Security Bug Findings

7.5.1 Missing Check

Figure 2b is one of the confirmed missing LSM checks (LSM-21). We discuss two more confirmed cases.

The CAP-4 missing check in kernel `random` device driver is particularly critical and triggered active discussion in the kernel developer community (including Torvalds). Random number generator serves as the foundation of many cryptography libraries including OpenSSL, and thus the quality of the random number is very critical. This security bug allows attackers to manipulate entropy pool, which can potentially corrupt many applications using cryptography libraries. Specifically, a problematic path starts from `evdev_write` and reaches the privileged function `credit_entropy_bits`, which can control the entropy in the entropy pool, while bypassing the required `CAP_SYS_ADMIN` permission check.

The LSM-21 missing check in `xfs_file_ioctl` led to another interesting discussion among kernel developers [9]. With this interface, a userspace program may perform low-level file system operations, but `security_inode_read_link` LSM hook was missing. An adversary could exploit this interface and gain access to the whole file system that is not allowed by LSM policy. Interestingly, however, the privileged function performed `CAP_SYS_ADMIN` Capability permission check. This created disagreement between kernel developers: one group argues that the LSM hook is necessary, while another thinks that `CAP_SYS_ADMIN` is sufficient. We agree with the former because LSM is designed to limit the damage of a compromised process to the system, even the ones of root user [40]. We believe that LSM permission checks should still be enforced as always for better security even when the current user is root.

Kernel developers decided not to fix 9 of our reports because they believe these bugs are not exploitable. As discussed earlier, PeX in the current form neither validates if a suspicious static path is dynamically reachable, nor generates a concrete exploit to demonstrate the security issue; we believe both are good future works. Nonetheless, we have one complaint to share.

For the LSM-19 and LSM-20 cases, PeX found that the LSM hooks `security_kernel_read_file` and `security_kernel_post_read_file` were used to protect the privileged functions `kernel_read_file` and `kernel_post_read_file` in some program paths. We reported missing LSM hooks in `load_elf_binary` and `load_elf_library` for these privileged functions. However, the kernel developers responded that those hooks are used to monitor loading firmware/kernel modules only (not other files), and thus no patch is required. Here, the implication we found is three-fold. First, the permission check names are ambiguous and misleading. Second, we were not able to find any documentation of such LSM specification regarding the protection of firmware/kernel modules. Last, PeX actually found a counter-example in `IMA` where the same checks are indeed used for loading other files (neither firmware nor kernel modules). Consequently, we suggest changing the function name and documenting the clear intention to avoid any confusion and to prevent system administrators from creating an LSM policy that does not work.

7.5.2 Inconsistent Check

The CAP-13 inconsistent check has been discussed in Figure 1. One program path in Figures 1a and 1c has two `CAP_SYS_RAWIO` checks and one `CAP_SYS_ADMIN` check, while another path in Figures 1b and 1c has only one `CAP_SYS_ADMIN` check. PeX detects this bug as an inconsistent check, but from the viewpoint of correction, which requires adding `CAP_SYS_RAWIO`, this may also be viewed as a missing check. There is a separate redundant check error in `CAP_SYS_RAWIO`.

Upon further investigation, we were interested in learning the practices in using multiple capabilities together. `scsi_ioctl` in Figure 1a checks both `CAP_SYS_ADMIN` **and** `CAP_SYS_RAWIO`. However, in a different network subsystem (not shown), we found that `too_many_unix_fds` performs a *weaker* permission check with the `CAP_SYS_ADMIN` **or** `CAP_SYS_RAWIO` condition. We believe this OR-based weaker check is not a good practice because this in effect makes `CAP_SYS_ADMIN` too powerful (like root), diminishing the ben-

Table 6: Bugs Reported By PeX. **C**onfirmed or **I**gnored.

Type-#	File	Function	Description	Status
DAC-1	fs/btrfs/send.c	btrfs_send	missing DAC check when traversing a snapshot	C
DAC-2	fs/ecryptfs/inode.c	ecryptfs_removexattr(),_setxattr()	missing xattr_permission()	C
DAC-3	fs/ecryptfs/inode.c	ecryptfs_listxattr()	missing xattr_permission()	C
CAP-4	drivers/char/random.c	write_pool(), credit_entropy_bits()	missing CAP_SYS_ADMIN	C
CAP-5	drivers/scsi/sg.c	sg_scsi_ioctl()	missing CAP_SYS_ADMIN or CAP_RAW_IO	I
CAP-6	drivers/block/pktcdvd.c	add_store(), remove_store()	missing CAP_SYS_ADMIN	I
CAP-7	drivers/char/nvram.c	nvram_write()	missing CAP_SYS_ADMIN	I
CAP-8	drivers/firmware/efi/efivars.c	efivar_entry_set()	missing CAP_SYS_ADMIN	C
CAP-9	net/rfkill/core.c	rfkill_set_block(), rfkill_fop_write()	missing CAP_NET_ADMIN	C
CAP-10	block/scsi_ioctl.c	mmc_rpmb_ioctl()	missing verify_command or CAP_SYS_ADMIN	I
CAP-11	drivers/platform/x86/thinkpad_acpi.c	acpi_evalf()	missing CAP_SYS_ADMIN	I
CAP-12	drivers/md/dm.c	dm_blk_ioctl()	missing CAP_RAW_IO	I
CAP-13	block/bsg.c	bsg_ioctl	inconsistent/missing CAP_SYS_ADMIN	C
CAP-14	kernel/sys.c	prctl_set_mm_exe_file	inconsistent capability check	I
CAP-15	kernel/sys.c	prctl_set_mm_exe_file	inconsistent capability and namespace check	I
CAP-16	block/scsi_ioctl.c	blk_verify_command	redundant check CAP_SYS_RAWIO	I
LSM-17	fs/ecryptfs/inode.c	ecryptfs_removexattr(), _setxattr()	missing security_inode_removexattr()	C
LSM-18	mm/mmap.c	remap_file_pages	missing security_mmap_file()	I
LSM-19	fs/binfmt_elf.c	load_elf_binary()	missing security_kernel_read_file	I
LSM-20	fs/binfmt_elf.c	load_elf_library()	missing security_kernel_read_file	I
LSM-21	fs/xfs/xfs_ioctl.c	xfs_file_ioctl()	missing security_inode_readlink()	C
LSM-22	kernel/workqueue.c	wq_nice_store()	missing security_task_setnice()	C
LSM-23	fs/ecryptfs/inode.c	ecryptfs_listxattr()	missing security_inode_listxattr	C
LSM-24	include/linux/sched.h	comm_write()	missing security_task_prctl()	C
LSM-25	fs/binfmt_misc.c	load_elf_binary()	missing security_bprm_set_creds()	I
LSM-26	drivers/android/binder.c	binder_set_nice	missing security_task_setnice()	I
LSM-27	fs/ocfs2/cluster/tcp.c	o2net_start_listening()	missing security_socket_bind	I
LSM-28	fs/ocfs2/cluster/tcp.c	o2net_start_listening()	missing security_socket_listen	I
LSM-29	fs/dlm/lowcomms.c	tcp_create_listen_sock	missing security_socket_bind	I
LSM-30	fs/dlm/lowcomms.c	tcp_create_listen_sock	missing security_socket_listen	I
LSM-31	fs/dlm/lowcomms.c	sctp_listen_for_all	missing security_socket_listen	I
LSM-32	net/socket.c	kernel_bind	missing security_socket_bind	I
LSM-33	net/socket.c	kernel_listen	missing security_socket_listen	I
LSM-34	net/socket.c	kernel_connect	missing security_socket_connect	I
LSM-35	fs/ocfs2/cluster/tcp.c	o2net_start_listening()	redundant security_socket_create	C
LSM-36	fs/ocfs2/cluster/tcp.c	o2net_open_listening_sock()	redundant security_socket_create	C

efit of fine-grained capability-based access control.

The CAP-14 and CAP-15 inconsistent error reports were acknowledged but ignored by the kernel developers for the following reason. For the same privileged function prctl_set_mm_exe_file, which is used to set an executable file, PeX discovered one case requiring CAP_SYS_RESOURCE in user namespace, and another case checking CAP_SYS_ADMIN in init namespace. Kernel developers responded that each case is fine by design for that specific context. PeX does not consider the precise context in which prctl_set_mm_exe_file is used (similar to aforementioned security_kernel_read_file used for loading kernel modules), leading to an imprecise report, but we believe that both CAP-14 and CAP-15 are worthwhile for further investigation.

7.5.3 Redundant Check

A redundant check occurs in two forms. **First**, for user-reachable functions, it happens when a privileged function is covered by the same permission checks multiple times. We reported three cases. The CAP-16 case was discussed in Figures 1a and 1c with two CAP_SYS_RAWIO checks, which was ignored by kernel developers. On the other hand, for the LSM-35 and LSM-36 cases found in the ocfs2 file system, the other kernel developer group confirmed and promised to fix the bugs. **Second**, any permission check in kernel-initialization functions is marked as redundant because the boot thread is executed under root. PeX detected tens of such cases, but we did not report them as they are less critical.

8 Related Work

8.1 Hook Verification and Placement

There is a series of studies on checking kernel LSM hooks. Automated LSM hook verification work [56] verifies the complete mediation of LSM hooks relying manually specified security rules. While [20] automates LSM hook placements utilizing manually written specification of security sensitive operations. However, required manual processes are error-prone when applied to huge Linux code base. Edwards et al. [18] proposed to use dynamic analysis to detect LSM hook inconsistencies. While PeX is using static analysis, can achieve better code coverage.

AutoISES [44] is the most closely related work to PeX. AutoISES regards data structures, such as the structure fields and global variables, as privileged, applies static analysis to extract security check usage patterns, and validates the protections to these data structures. The difference between AutoISES and PeX is three-fold. First, PeX is privileged function oriented while AutoISES is more like data structure oriented.

Second, AutoISES is designed for LSM only, whose permission checks (hooks) are clearly defined, and therefore it is not applicable to DAC and Capabilities due to their various permission check wrappers. In contrast, PeX works for all three types of permission checks. Third, AutoISES uses type-based pointer analysis to resolve indirect calls, while PeX uses KIRIN to resolve indirect calls in a more precise manner.

There are also works [21, 32, 33] that extend authorization hook analysis to user space programs, including X server and postgresql. However, their approaches canot be applied to the huge kernel scale. Moreover, all of above works either do not analyze indirect calls at all, or apply over approximate indirect call analysis techniques, such as type-based approach or field insensitive approach. To the contrary, PeX uses KIRIN, a precise and scalable indirect call analysis technique, which can also benefit these works by finding more accurate indirect call targets.

8.2 Kernel Static Analysis Tools

Coccinelle [34] is a tool that detects a bug of pre-defined pattern based on text pattern matching on the symbolic representation of bug cases. This is basically intra-procedural analysis. Building upon Coccinelle, Wang et al. proposed another pattern matching based static tool which detects potential double-fetch vulnerabilities in the Linux kernel [48].

Sparse [11] is designed to detect the problematic use of pointers belonging to different address space (kernel space or userspace). Later, Sparse was used to build a static analysis framework called **Smatch** [10] for detecting different sorts of kernel bugs. However, Smatch is also based on intra-procedural analysis, thus it can only find shallow bugs.

Double-Fetch [52], **Check-it-again** [49] focus on detecting time of check to time of use (TOCTTOU) bugs. **Dr. Checker** [29] is designed for analyzing Linux kernel drivers. It adopts the modular design, allowing new bug detectors to be plug-in easily. **KINT** [50] applies taint analysis to detect integer errors in Linux kernel while **UniSan** [28] leverages the same analysis to detect uninitialized kernel memory leakages to the userspace. **Chucky** [53] also uses a taint analysis to analyze missing checks in different sources in userspace programs and Linux kernel. However, Chucky can handle only kernel file system code due to the lack of pointer analysis. Note that to resolve indirect call targets, all these works leverage a type-based approach, which is not as accurate as KIRIN, thus suffering from false positives.

MECA [54] is an annotation based static analysis framework, and it can detect security rule violations in Linux. **APISan** [55] aims at finding API misuse. It figures out the right API usage through the analysis of existing code base and performs intra-procedural analysis to find bugs. To achieve the former, APISan relies on relaxed symbolic execution which is complementary to the techniques used in PeX.

8.3 Permission Check Analysis Tools

Engler et al. propose to use programmer beliefs to automatically extract checking information from the source code. They apply the checking information to detect missing checks [19]. **RoleCast** [42] leverages software engineering patterns to detect missing security checks in web applications. **TESLA** [14] implements temporal assertions based on LLVM instrument, in which the FreeBSD hooks are checked by inserted assertions dynamically. Different from TESLA, PeX uses KIRIN to analyze jump targets of all kernel function pointers statically, achieving better resolution rate and code coverage. **JIGSAW** [47] is a system that can automatically derive programmer expectations on resources access and enforce it on the deployment. It is designed for analyzing userspace programs, cannot be applied to kernel directly.

JUXTA [30] is a tool designed for detecting semantic bugs in filesystem while **PScout** [15] is a static analysis tool for validating Android permission checking mechanisms. **Kratos** [39] is a static security check framework designed for the Android framework. It builds a call graph using LLVM and tries to discover inconsistent check paths in the framework. However, Android has well-documented permission check specifications [2], i.e., privileged functions and the permission required for them are both clearly defined. In contrast, the Linux kernel has no such documentation, which makes it impossible to apply PScout and Kratos to Linux kernel permission checks.

9 Conclusion

This paper presents PeX, a static permission check analysis framework for Linux kernel, which can automatically infer mappings between permission checks and privileged functions as well as detect missing, inconsistent, and redundant permission checks for any privileged functions. PeX relies on KIRIN, our novel call graph analysis based on kernel interfaces, to resolve indirect calls precisely and efficiently.

We evaluated both KIRIN and PeX for the latest stable Linux kernel v4.18.5. The experiments show that KIRIN can resolve 86%-92% of all indirect callsites in the kernel within 7 minutes. In particular, PeX reported 36 permission check bugs of DAC, Capabilities, and LSM, 14 of which have already been confirmed by the kernel developers. PeX source code is available at https://github.com/lzto/pex, along with the identified mapping between permission checks and privileged functions. We believe that such a mapping allows kernel developers to validate their code with PeX and encourages them to contribute to PeX by refining the mapping with their domain knowledge.

Acknowledgments

The authors would like to thank anonymous reviewers for their insightful comments. This research is partially supported by the NSF under Grant No. CSR-1814430 and CSR-1750503.

References

[1] alse boundaries and arbitrary code execution. https://forums.grsecurity.net/viewtopic.php?f=7&t=2522.

[2] Android Permission Overview. https://developer.android.com/guide/topics/permissions/overview.

[3] Apparmor. https://gitlab.com/apparmor/apparmor/wikis/home/.

[4] capabilities - overview of linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html.

[5] CAP_SYS_ADMIN: the new root. https://lwn.net/Articles/486306/.

[6] K-miner: Data-flow analysis for the linux kernel. https://github.com/ssl-tud/k-miner.

[7] Locationmanager. https://developer.android.com/reference/android/location/LocationManager#getLastKnownLocation(java.lang.String).

[8] Mandatory access control. https://en.wikipedia.org/wiki/Mandatory_access_control.

[9] Re: Leaking path in xfs's ioctl interface(missing lsm check) by stephen smalley. https://lkml.org/lkml/2018/9/26/668.

[10] Smatch: pluggable static analysis for c. https://lwn.net/Articles/691882/.

[11] Sparse. https://www.kernel.org/doc/html/v4.14/dev-tools/sparse.html.

[12] Virtual file system. https://en.wikipedia.org/wiki/Virtual_file_system.

[13] Whole Program LLVM: a wrapper script to build whole-program llvm bitcode files. https://github.com/travitch/whole-program-llvm.

[14] Jonathan Anderson, Robert NM Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. Tesla: temporally enhanced system logic assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, page 19. ACM, 2014.

[15] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[16] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[17] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–48. ACM, 1995.

[18] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 225–234. ACM, 2002.

[19] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.

[20] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 330–339. ACM, 2005.

[21] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Towards automated authorization policy enforcement. In *Proceedings of Second Annual Security Enhanced Linux Symposium*. Citeseer, 2006.

[22] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2018.

[23] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices*, volume 42, pages 290–299. ACM, 2007.

[24] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. pages 265–280, 2007.

[25] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 289–298. IEEE Computer Society, 2011.

[26] Bogdan Korel and Juergen Rilling. Program slicing in understanding of large programs. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 145–152. IEEE, 1998.

[27] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004.

[28] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932. ACM, 2016.

[29] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. checker: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1007–1024. USENIX Association, 2017.

[30] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377. ACM, 2015.

[31] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[32] Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. Leveraging choice to automate authorization hook placement. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 145–156. ACM, 2012.

[33] Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. Producing hook placements to enforce expected access control policies. In *International Symposium on Engineering Secure Software and Systems*, pages 178–195. Springer, 2015.

[34] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Acm sigops operating systems review*, volume 42, pages 247–260. ACM, 2008.

[35] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 126–135. IEEE, 2009.

[36] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.

[37] Sanjay Rawat, Laurent Mounier, and Marie-Laure Potet. Listt: An investigation into unsound-incomplete yet practical result yielding static taintflow analysis. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 498–505. IEEE, 2014.

[38] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[39] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[40] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[41] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[42] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[43] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[44] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[45] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994.

[46] National Computer Security Center (US). *A guide to understanding discretionary access control in trusted systems*, volume 3. National Computer Security Center, 1987.

[47] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting resource access by inferring programmer expectations. In *USENIX Security Symposium*, pages 973–988, 2014.

[48] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *USENIX Security Symposium*, 2017.

[49] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of ACM conference on Computer and communications security*. ACM, 2018.

[50] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *OSDI*, volume 12, pages 163–177, 2012.

[51] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, pages 6–16, 2002.

[52] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. 2018.

[53] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510. ACM, 2013.

[54] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 321–334. ACM, 2003.

[55] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *USENIX Security Symposium*, pages 363–378, 2016.

[56] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *USENIX Security Symposium*, pages 33–48, 2002.