This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

Automatic Permission Check Analysis for Linux Kernel

Jinmeng Zhou, Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed Azab, Ruowen Wang, Kui Ren, Peng Ning

Abstract—Permission checks play an essential role in operating system security by providing access control to privileged functionalities. However, it is challenging for kernel developers to scalably verify the soundness of existing checks due to the large codebase 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. Over the interprocedural control flow graph built by KIRIN, PeX automatically identifies 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. 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 45 new permission check errors, 17 of which have been confirmed by the kernel developers.

Index Terms—Linux kernel, static analysis, permission check, bug detection.

1 Introduction

Access control [1] 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 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 [2]. 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 [3] and AppArmor [4].

Unfortunately, for a new feature or a newly identified vulnerability, 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, Capabilities 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 have validated the required permission checks before reaching the function. Given the distributed nature of kernel development and the significant amount of daily updates, chances are that some 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

- *J. Zhou, W. Shen and K. Ren are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China.*
  Email: {11921110,shenwenbo,kuiren}@zju.edu.cn;
- *T. Zhang is with Samsung Electronics.*
  Email: ztong0001@gmail.com
- *D. Lee is with Department of Computer Science, Stony Brook University.*
  Email: dongyoon@cs.stonybrook.edu
- *C. Jung is with Department of Computer Science, Purdue University.*
  Email: chjung@purdue.edu
- *A. Azab is with Facebook.*
  Email: amazab80@gmail.com
- *R. Wang and P. Ning are with Google.*
  Email: {ruowenwang,pning}@google.com
- *W. Shen is the corresponding author.*

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

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 would like to address. First, Linux kernel uses indirect calls very frequently, yet its static call graph analysis is notoriously difficult. The Linux kernel v4.18.5 contains 15.8M LOC, 247K functions, and 115K indirect callsites, rendering existing precise solutions (e.g., SVF [6]) unscalable. The only workaround available to date is either to apply the solutions unsoundly (e.g., only on a small code partition as with K-Miner [7]) 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 false 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 [8], 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 automatically identifies permission checks and all their wrappers. Moreover, PeX leverages dominator analysis [9] to automatically identify the mappings between permission checks and their potentially privileged functions.

The contributions of this paper are summarized as follows:

- **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 45 new permission check bugs, 17 of which have been confirmed by kernel developers.
- **Community Contributions**: We 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 [10], [11]. 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.

TABLE 1: Commonly used permission checks in Linux kernel.

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_readlink, security_file_ioctl, etc..

In versions before v2.2, the Linux kernel uses a simple separation of normal users and the super user (i.e., root), where the root bypasses all the in-kernel permission checks. This motivates fine-grained access control scheme—such as Capabilities—to weaken the power of the root user.

2.2 Capabilities

Capabilities, since Linux kernel v2.2 (1999), enable a fine-grained access control by dividing the root privileges into small sets. As an example, for users with the `CAP_NET_ADMIN` capability, kernel allows them to use `ping`, without granting the full root privileges. 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 [2]. 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 in v4.18.5, 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 [12].

2.3 Linux Security Module (LSM)

LSM [13], 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. One common use of LSM is to implement Mandatory Access Control (MAC) [14] in Linux (e.g., SELinux [3], [15], AppArmor [4]). 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

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

```
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.

Fig. 1: Capabilities check errors discovered by PeX.

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 the 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 Capabilities Permission Check Errors

Figure 1 shows real code snippets of Capabilities 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  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.

Fig. 2: LSM 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 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.

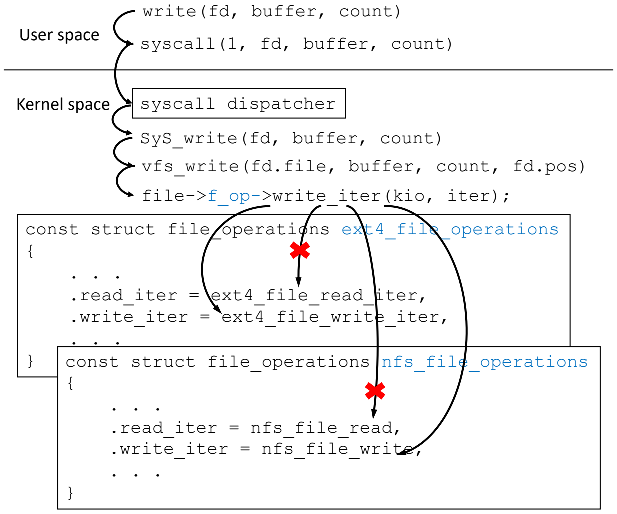
Figure 2b presents two sub-functions for the system call ioctl. Similar to the above case, ioctl calls ksys_ioctl, which includes its own LSM hook security_file_ioctl (Line 4) before do_vfs_ioctl (Line 6). This is a proper design, and there is no problem so far. However, it turns out that there is

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

```
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.



```
write(fd, buffer, count)
syscall(1, fd, buffer, count)
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.

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

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 the 'ioctl' allow rule may exploit the ioctl system call to trigger the vfs_readlink operation, while this operation might not be permitted by the 'read' allow rule.

The above two Capabilities 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 uses a diverse set of abstraction layers that specify the common *interfaces* to different concrete implementations. For example, Virtual File System (VFS) [16] 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 *kernel 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 [6], [17], [18], [19], [20] unscalable. In fact, the state-of-the-art technique—Static Value Flow (SVF) [6] uses flow- and context-sensitive value flow for high precision, but cannot scale to the huge Linux kernel code base. 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. Therefore, it is challenging to identify all permission checks and wrappers.

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 [8], which has been designed with the permission-based security model in mind from the beginning. Take the Android LocationManager class as an example;

[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.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

for the getLastKnownLocation method, the API document states explicitly that permission ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION is required [21].

Unfortunately, existing *static* permission error checking techniques are not readily applicable in order to address these problems. Automated LSM hook verification [22] 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 [23], [24], authorization constraints [25], annotation on sensitive objects [26]. These manual processes are particularly error-prone when applied to huge Linux code base. Alternatively, some works such as [27], [28] rely on *dynamic* analysis. However, such run-time approaches may significantly limit the code coverage being analyzed, thereby missing real bugs.

Moreover, none of the existing works can 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 automatically identifies basic permission checks and leverages them as a basis for finding other permission check wrappers (§6.3). In addition, PeX proposes a dominator analysis based solution to automatically infer the mappings between permission checks and privileged functions (§6.4).

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 [29] 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 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 @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.

```
%25 = load %struct.file_operations*,
↪ %struct.file_operations** %f_op, align 8
%write_iter.i.i = getelementptr inbounds
↪ %struct.file_operations,
↪ %struct.file_operations* %25, i64 0, i32 5
%26 = load i64 (%struct.kiocb*, %struct.iov_iter*)*,
↪ i64 (%struct.kiocb*, %struct.iov_iter*)**
↪ %write_iter.i.i, align 8
%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.

Fig. 4: Indirect callsite resolution for vfs_write.

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

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

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

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

```
#define container_of(ptr, type, member) ({           \
    type* __res;                                     \
    void* __mptr = ((void *)((void*)(ptr) -          \
    ↪ offsetof(type, member)));                      \
    memcpy(&__res, &__mptr, sizeof(void*));          \
    (__res);})
%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.

Fig. 5: Fixing container_of missing struct type problem.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

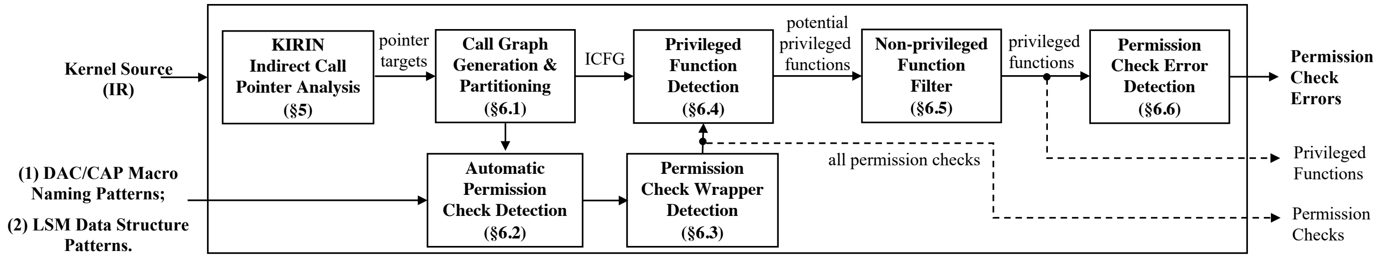

Kernel Source (IR)
(1) DAC/CAP Macro Naming Patterns;
(2) LSM Data Structure Patterns.
KIRIN Indirect Call Pointer Analysis (§5)
pointer targets
Call Graph Generation & Partitioning (§6.1)
ICFG
Privileged Function Detection (§6.4)
potential privileged functions
Non-privileged Function Filter (§6.5)
privileged functions
Permission Check Error Detection (§6.6)
Permission Check Errors
Automatic Permission Check Detection (§6.2)
Permission Check Wrapper Detection (§6.3)
all permission checks
Privileged Functions
Permission Checks

Fig. 6: PeX static analysis architecture. Kernel source code is the input of this architecture. PeX takes as input the naming patterns of DAC and Capabilities, and LSM certain data structure, to automatically detect the permission check. Finally PeX reports as output permission check errors. PeX also produces mappings between identified permission checks and privileged functions as output.

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`. Note that `container_of` is the only special case that needs to be handled. Overcoming this, KIRIN achieves a high indirect callsite resolution rate.

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., [7]).

6 Design of PeX

Figure 6 shows the architecture of PeX. It takes as input kernel source code (in the LLVM bitcode format), 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 [30]. Then, taking the DAC/CAP macro naming patterns and the LSM data structure patterns as input, PeX detects permission checks (§6.2) and their wrappers automatically (§6.3). After that, for a permission check, PeX identifies its potentially privileged functions on top of the ICFG (§6.4), followed by a heuristic-based filter to prune obviously non-privileged functions (§6.5). Finally, for each privileged function, PeX examines all user space reachable paths to it to detect any permission checks error on the paths (§6.6). 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) [9] used in the inference of the mapping between permission checks and privileged functions (§6.4).

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

TABLE 2: Automatically Detected Permission Check Functions

	defconfig			allyesconfig		
	DAC	CAP	LSM	DAC	CAP	LSM
# of basic checks	13	16	184	37	21	218
# of detected wrappers	10	18	104	97	68	159
# of all checks	16	20	288	106	72	377

```
// DAC: 8 macros in /include/linux/fs.h
#define MAY_EXEC            0x00000001
#define MAY_WRITE           0x00000002
#define MAY_READ            0x00000004
...
#define MAY_NOT_BLOCK       0x00000080

// Capabilities: 38 macros in
  ↪  /include/uapi/linux/capability.h
#define CAP_CHOWN                 0
#define CAP_DAC_OVERRIDE          1
#define CAP_DAC_READ_SEARCH       2
...
#define CAP_AUDIT_READ           37
```

(a) The definitions of permission macros.

```
int generic_permission(struct inode *inode, int
  ↪  mask)
{
    ...
    ret = acl_permission_check(inode, mask);
    ...
    mask &= MAY_READ | MAY_WRITE | MAY_EXEC;
    if (mask == MAY_READ)
        if (capable_wrt_inode_uidgid(inode,
          ↪  CAP_DAC_READ_SEARCH))
            return 0;
    ...
}
```

(b) The usages of permission macros in a permission check function.

Fig. 7: The permissions of DAC and Capabilities are implemented as macro-based definitions (v4.18.5).

6.2 Automatic Permission Check Detection

On the indirect call resolved call graph, PeX requires all permission checks at first, as shown in Figure 6. In the preliminary version of this paper [31], we manually select several basic permission checks as input for follow-up detection. Unfortunately, it has several drawbacks. First, the detection totally depends on the quality of the user input, and low quality leads to imprecision. If user input misses a basic check, we miss the wrapper permission checks derived from the basic one. Furthermore, PeX detects privileged functions and report bugs based on the identified checks, which results in missing bug reports. Second, the requirement of user input introduces a maintenance burden when using PeX because of the rapid evolution of Linux kernel. For different kernel versions, the users of PeX are required to recollect the set of permission checks, as Linux kernel developers might add or remove certain permission checks. To address these problems, we propose a new approach for detecting permission checks automatically for DAC, Capabilities, and LSM.

6.2.1 Permission Check Detection for DAC and Capabilities

Our insights. We observe that DAC and Capabilities use macros to represent the actual permissions, and these macros are used by the permission checks. Therefore, PeX can leverage usages of these macros to detect permission checks. Linux kernel defines 8 DAC macros and 38 Capabilities macros to represent permissions for DAC and Capabilities in v4.18.5, as shown in Figure 7a.

Algorithm 1 Basic Permission Check Detection for DAC and Capabilities

```
INPUT:
    $macros$ - DAC and Capabilities permission macros
OUTPUT:
    $macrousages$ - macro usage set
    $permchecks$ - basic permission checks
 1: procedure MACROUSAGEDETECTION($var$)
 2:     for $u \leftarrow Usage(var)$ do
 3:         if $u$ is an Assignment Instruction then
 4:             $v \leftarrow getSinkVar(u)$
 5:             $MacroUsageDetection(v)$
 6:         else
 7:             $macrousages.insert(u)$
 8:         end if
 9:     end for
10: end procedure
11: procedure PERMISSIONCHECKDETECTION($macros$)
12:     for $m \leftarrow macros$ do
13:         $MacroUsageDetection(m)$
14:     end for
15:     for $usage \leftarrow macrousages$ do
16:         if $usage$ is a Call Instruction then
17:             $func \leftarrow getCallee(usage)$
18:             $permchecks.insert(func)$
19:         else if $usage$ is a Comparison Instruction then
20:             $func \leftarrow getFunction(usage)$
21:             $permchecks.insert(func)$
22:         end if
23:     end for
24:     return $permchecks$
25: end procedure
```

These macros are defined following certain naming patterns, such as the ones for DAC are beginning with `MAY_` and the ones for Capabilities are beginning with `CAP_`. For example, `CAP_DAC_READ_SEARCH` represents the capability to perform file/directory read operations [32]. Therefore, PeX takes these macro naming patterns as the input to detect permission checks automatically.

Though conceptually simple, the detection faces two challenges. First, the defined permission macros might not be used directly for permission checks. A macro is possibly assigned to a variable and then propagates to more variables. These variables are then used for permission checks. To collect a comprehensive set of macro usages, PeX performs a field-sensitive inter-procedural analysis to collect both usages of macros (direct usages) and usages of those variables (indirect usages). Second, macro usages vary significantly in Linux kernel, such as being passed as arguments into a function or used in a function directly. To detect permission checks precisely, PeX needs to handle macro usages separately. Taking macro naming patterns as input, PeX conducts a two-step analysis, namely macro usage detection and permission check detection, to output basic permission checks, as shown in Algorithm 1.

Macro usage detection: PeX conducts a field-sensitive inter-procedural analysis to collect both direct and indirect macro usages, as shown in Algorithm 1. For each macro, the procedure `MacroUsageDetection` is called (Line 13) with the macro as an initial input. It traverses all usages of the input macro and collects them into a set $macrousages$. If the usage is an assignment instruction that assigns the macro to a sink variable v, `MacroUsageDetection` is called recursively with v as a new input (Line 4 and 5). Otherwise, the usages are inserted into the set $macrousages$ (Line 7).

The analysis is inter-procedural and field-sensitive. If the variable (field) is passed to a function call, the analysis digs into the callee to find its usages. Finally, the iterative algorithm collects

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

```
1  struct security_hook_heads {
2      ...
3      struct hlist_head key_permission;
4      ...
5  } __randomize_layout;
6
7  static struct security_hook_list selinux_hooks[]
    ↪  __lsm_ro_after_init = {
8      ...
9      LSM_HOOK_INIT(key_permission,
        ↪  selinux_key_permission),
10     ...
11 };
12
13 #define call_int_hook(FUNC, IRC, ...) ({ \
14     ...                                  \
15     hlist_for_each_entry(P,
    ↪   &security_hook_heads.FUNC, list) {   \
16         RC = P->hook.FUNC(__VA_ARGS__);  \
17         ...                              \
18 })
19
20 int security_key_permission(key_ref_t key_ref,
    ↪  const struct cred *cred, enum key_need_perm
    ↪  need_perm)
21 {
22     return call_int_hook(key_permission, 0,
        ↪  key_ref, cred, need_perm);
23 }
```

Fig. 8: LSM permission checks (hooks) uses `security_hook_heads` to organize registered callbacks.

all direct and indirect usages into the set *macrousages*.

Permission check detection: PeX leverages the collected macro usages to detect basic permission checks, as shown by the procedure `PermissionCheckDetection` in Algorithm 1. Here we regard the function as a basic permission check if the function either (1) directly/indirectly uses the macro as an argument or (2) contains a comparison instruction that directly/indirectly uses the macro as an operand.

Functions with permission macro arguments: PeX analyzes all call instructions in the macro usage set to detect these functions. If a call instruction directly/indirectly uses a macro as an argument, the analysis marks the callee as a permission check function. As shown in Figure 7b at Line 8, the macro `CAP_DAC_READ_SEARCH` is used as an argument in the call instructions. The analysis identifies the callee `capable_wrt_inode_uidgid` as a basic permission check.

Functions containing permission macro comparison instructions: PeX traverses all comparison instructions in the macro usage set. For each instruction, at least one operand is macro directly or the variable that is assigned with a macro. The functions that contain these comparison instructions are marked as permission check functions. As shown at Line 7 in Figure 7b, `MAY_READ` is used for the comparison, so the function `generic_permission` is marked as a permission check.

The automatically-detected basic permission checks are listed in Table 6b, which contain more functions than the ones in the user input. More specifically, the automatic detection approach detects 13 DAC and 16 Capabilities basic checks for `defconfig`, and detects 37 and 21 basic checks for `allyesconfig`.

6.2.2 Permission Check Detection for LSM

We observe that the organization of LSM data structure follows certain patterns, allowing PeX to detect all LSM permission checks automatically. Different from DAC and Capabilities, LSM does not rely on macros to check permissions. Currently, Linux kernel inserts LSM hooks on the critical paths. These LSM hooks can be instantiated with different implementations to achieve different access control mechanisms, such as SELinux or AppArmor. We regard these LSM hooks as LSM permission checks. Our insight is that LSM uses a critical data structure (termed as `H`) to gather all LSM hook implementations, and each member of `H` is linked to one LSM hook. PeX regards the LSM hooks that directly use the member of `H` as basic LSM permission checks. Therefore, by traversing all members of `H` and following the call chains of these members, PeX can detect all basic LSM permission checks automatically. Moreover, `H` is highly specialized for LSM use only. No other functionalities use it. Therefore, by using `H`, PeX is able to detect LSM permission checks with high precision.

Let us use the example in Figure 8 to illustrate the detection process. In Linux kernel, the data structure `H` is defined as `security_hook_heads` (Line 1). SELinux registers one hook implementation `selinux_key_permission` into the member `key_permission` of `security_hook_heads` at Line 9. This member is further called by the permission check `security_key_permission` (Line 20) via `call_int_hook` (Line 22 and 15). Therefore, by following the calling chain of `security_hook_heads.key_permission`, PeX is able to detect the permission check `security_key_permission` automatically. In our experiments, PeX detects 184 basic LSM permission checks for `defconfig` and 218 ones for `allyesconfig` automatically, as shown in Table 6b.

6.3 Precise Permission Check Wrapper Detection

After collecting basic permission checks, we need to detect the wrappers that are also used for permission checking. The identified basic checks are often wrappers of inner permission checks that perform low-level access control, and even worse, there could be outer wrappers of the wrappers. PeX requires wrapper detection to solve this. The wrapper detection in preliminary version [31] identifies wrappers by matching their parameter of wrappers with any parameter of user-provided basic permission checks, which introduces many false positives. In order to improve the precision of the permission check wrapper detection, we propose a new technique termed as *Macro-flow based Wrapper Detection* for DAC and Capabilities. PeX uses a backward analysis to find LSM wrappers. Finally, We leverage a heuristic-based method to filter false positives.

6.3.1 Macro-flow based Wrapper Detection

For DAC and Capabilities, a permission check might pass its permission macro to a callee function to do low-level permission check. For example in Figure 9, the identified basic check `capable_wrt_inode_uidgid` pass its last argument `cap` to the callee `ns_capable`, while `cap` accepts the permission `CAP_DAC_READ_SEARCH` at Line 8 in Figure 7b. As these wrappers are called inside basic permission checks, we refer to them as *inner-wrappers*. Similarly, there are cases in which basic permission checks get permission macros from their callers. We define these callers as *outer-wrappers*. Based on tracing macro-related parameters, PeX conducts a forward inter-procedural data-flow analysis on the ICFG to detect inner-wrappers and a backward analysis to detect outer-wrappers.

Inner-wrapper Detection: For each basic check, a forward analysis first recognizes the macro-related parameter. Then PeX analyzes inner call instructions for the one that passes the param-

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

```
1  bool capable_wrt_inode_uidgid(const struct inode
     ↪  *inode, int cap)
2  {
3    struct user_namespace *ns = current_user_ns();
4    return ns_capable(ns, cap) &&
      ↪  privileged_wrt_inode_uidgid(ns, inode);
5  }
6
7  int nfs_permission(struct inode *inode, int mask)
8  {
9    ...
10   if (res == 0)
11     res = generic_permission(inode, mask);
12   ...
13 }
```

Fig. 9: The example of inner/outer wrappers.

eter to its callee (inner-wrapper). Finally, the analysis recursively digs into the newly detected callee to find more inner wrappers.

For Capabilities: PeX directly tracks the macro-related parameter to detect the inner wrappers. Using the same example in Figure 7b, the function capable_wrt_inode_uidgid is marked as a basic permission check and its last argument cap accepts permission macro. PeX further digs into this function and starts to trace the last parameter cap. As shown in Figure 9, capable_wrt_inode_uidgid function calls three functions, and only the second one ns_capable accepts cap parameter. Therefore, ns_capable is identified as an inner-wrapper. Moreover, as the wrapper detection is a recursive process, PeX further analyzes ns_capable to detect more inner-wrappers.

For DAC: The detection for DAC is more complex than Capabilities because of the logic operation of macros. In Figure 7b, generic_permission is already considered as a basic permission check because it contains comparison instructions between permission macro MAY_READ and a masked variable mask. Note mask is masked with multiple DAC macros at Line 6, which is also a macro-related parameter. The analysis traces mask and finds out that mask is assigned to callee acl_permission_check at Line 4, which is marked as an inner-wrapper.

Outer-wrapper Detection: To detect outer-wrappers, PeX conducts a backward analysis on the call-chain, which takes all basic permission checks and inner-wrappers together as input. The analysis traverses all call sites of input functions and back traces macro-related arguments in the call sites. If the macro-related argument comes from the parameter of caller function, this caller is identified as an outer-wrapper. The function generic_permission is marked as a basic check before, and it is called with macro-related argument mask, as shown in Figure 9 at Line 11. Therefore, starting from this call site, PeX back traces the flow of mask and finds that it comes from the caller's parameter. As a result, the caller nfs_permission is marked as an outer-wrapper. The recursive analysis records the second parameter mask of nfs_permission as macro-related to find more outer-wrappers.

Compared with the approach in [31], our new approach traces the flow of macro-related arguments only rather than all arguments. Therefore, we reduce most of the false positives, as discussed in §7.3.2. Especially in defconfig, the false positive rate is close to zero.

6.3.2 LSM Wrapper Recognition

PeX conducts a backward traversal to detect LSM wrappers and leverages a heuristic-based filter to reduce false positives. PeX analyzes all callers of basic permission checks for the one that passes one of its parameters to a basic function. We assume these callers are LSM wrappers. In other words, we identify the callers that deliver their own formal parameters to the basic permission checks. Similar to DAC and Capabilities, the LSM wrapper detection is also recursive.

Algorithm 2 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 method of wrapper detection introduces many false positives, especially in allyesconfig kernel. PeX further uses a heuristic-based filter to prune out false permission check wrappers. In the current prototype, the filter consists of a set of kernel library functions and system calls. After the filtering, we further conduct a manual review to filter out more falsely detected wrappers. At last, we have 104 LSM wrappers for defconfig and 159 ones for allyesconfig, as shown in Table 6b.

6.4 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 [9] 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 (§6.3) 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), the callee is likely to be the one that should be protected by the check on all paths [2].

Algorithm 2 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.

[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.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

Algorithm 3 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 $pair \leftarrow pc - pv$ do
 3:         $pvfuncs \leftarrow pair.pv$                         ▷ privileged functions
 4:         $pcfunc \leftarrow pair.pc$                 ▷ permission check functions
 5:         for $f \leftarrow pvfuncs$ do
 6:             $allpath \leftarrow getAllPathUseFunc(f)$ ▷ get all user reachable paths that call the privileged function f
 7:             for $p \leftarrow allpath$ do
 8:                 $pvcall \leftarrow PrivilegeFunctionCallInPath(p)$
 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 $f \leftarrow kinitfuncs$ 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
```

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 7, LSM-17). Note that if some other unsound workaround such as [7] had been used, this bug would not have been detected.

6.5 Non-privileged Function Filter

Besides the wrapper filter, PeX also needs to filter out the non-privileged functions. The conservative approach in §6.4 may lead to false privileged functions. In this step, PeX applies heuristic-based filters to prune out false privileged functions. The filter contains a set of kernel library functions which are not privileged functions, e.g., `kmalloc`, `strcmp`, `kstrtoint`. If a detected privileged function falls into the filter set, PeX would remove it from the privileged functions.

PeX is currently designed to avoid false negatives on privileged functions, and thus chooses a conservative approach (i.e., a small set of library functions only) for filtering out privileged functions. On top of the existing filter, 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.

6.6 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 3 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 function (`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. With the `__init` attribute (§6.1), the functions in `kinitfuncs` can only be executed during booting and will be freed when the booting is done. Therefore, these functions do not need any permission checks. All detected permission checks are marked as redundant (Lines 22-24).

7 Implementation and Evaluation

The implementation of PeX has two versions. The first version was implemented on LLVM/Clang-6.0 [29] that contains about 7K lines of C/C++ code. It still requires the user-provided permission checks as the input [31]. The second version eliminates this requirement by automating the detection of permission checks. It was implemented on LLVM/Clang-11.0 and added about 2K lines of code. We evaluated PeX on kernel v4.18.5.

To support the automatic DAC and Capabilities permission detection, PeX redefines their permission macros to allow easy recognition. To differentiate from other constant numbers in kernel, PeX redefines the macros by adding a magic number. Moreover, to recognize multiple macros involving logical operations, the lower bits of the magic number must be zeroed out. Therefore, PeX chooses large prime numbers, left-shifts them by 12 bits and then uses them as the magic numbers. As such, PeX uses `0xF2827000` as the magic number for DAC and `0xF3FA3000` as the one for Capabilities. In total, PeX redefines 8 DAC macros and 38 Capabilities macros.

To support PeX, 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` [33]. This avoids building each kernel module separately or changing kernel build infrastructures, as observed in prior kernel static analysis works [7], [34]. In summary, KIRIN resolves 86%–92% of indirect callsites depending on its compilation configurations. PeX reported 45 permission check errors warnings to the Linux community, 17 of which have been confirmed as real bugs.

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`,

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

TABLE 3: Input Statistics for Kernel.

	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 4: 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

but also covers the majority of kernel code, allowing PeX to detect more bugs. In addition, we evaluated the two different versions of implementations. For the user input based check detection, 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. For the second version, we automatically detected the permission checks to find bugs. All experiments we carried out on a machine running Ubuntu 20.04.2 with two Intel Xeon Gold 5218R 2.10GHz and 512 DRAM. The other settings are the same with the first version.

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 [7] 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 [35] 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 4 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.

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)

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 [7]. 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 4. 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 4. 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

This section shows the results of PeX when using two different sets of permission checks, one set is user-provided and the other is automatically detected.

7.3.1 User-provided Permission Checks

Table 6a summarizes PeX's intermediate program analyses when using user-provided permission checks. 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

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

implement a multi-threaded version of PeX to further reduce the analysis time.

Given the small number of input DAC, Capabilities, and LSM permission checks (3, 3, and 190 each), PeX's permission check detection [31] 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 6a also shows the number of potentially privileged functions detected by PeX (§6.4) and the number of remaining privileged functions after kernel library filtering (§6.5). 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, Capabilities, and LSM, respectively. PeX reports 36 bugs using user-provided permission checks, 14 of which have been confirmed by Linux kernel developers.

Comparison. To highlight the effectiveness of KIRIN, we repeated the end-to-end PeX analysis of first version (user-provided permission checks) 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 7—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.

7.3.2 Automatically Detected Permission Checks

Table 6b summarizes the result of PeX with the automatic permission check detection. Similarly, we also focus on the result of `allyesconfig`, in which PeX finds 37, 21, and 218 basic checks for DAC, Capabilities, and LSM. Based on the basic ones, PeX detects 97, 68, and 159 permission check wrappers for DAC, Capabilities, LSM, respectively. The automatically-detected basic permission checks and wrappers are super-sets of the ones detected using user-provided checks. As a result, the automatic check detection reports all warnings found by the user input based check detection. Because more permission checks are detected, PeX is able to mark more privilege functions. For DAC, Capabilities, and LSM, 3129, 3020, and 3756 are detected after filtering. Finally, PeX reports 830, 491, and 558 warnings grouped by privilege functions for DAC, Capabilities, LSM. Note that the result of the user input based check detection contains many false permission checks, leading to more false warnings. PeX reports 9 more bugs using the automatic-detected checks than using the user-provided ones, 3 of which have been confirmed by Linux kernel developers. All 45 bugs are listed in Table 7. Kernel developers ignored some bugs and decided not to make changes because they believe that these bugs are not exploitable. We discuss them in detail in §7.5.

TABLE 6: The detection result using two different sets of permission checks.

(a) The result of user-provided permission checks (first version).

	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

(b) The result of automatic detection of permission checks (second version).

	defconfig			allyesconfig		
	DAC	**CAP**	**LSM**	**DAC**	**CAP**	**LSM**
# of basic checks	13	16	184	37	21	218
# of detected wrappers	10	18	104	97	68	159
# of priv func detected	619	964	2528	3301	3186	3890
# of priv func after filter	567	810	2315	3129	3020	3756
# of warnings grouped by priv func	129	137	337	830	491	558
total time (min)	1	2	2	235	207	130

After collecting the permission checks, we manually confirm true positives that are shown in Table 6b. For each detected permission check, we look up the source code, the calling context, and the annotation to confirm true positives.

Basic check detection comparison. For brevity, we term automatic-detected basic permission checks as P_{auto} and the user-provided ones as P_{user}. In `defconfig` kernel, P_{auto} contains all true positives of P_{user}. Note that P_{user} contains 190 LSM basic checks, which include 14 false positives and miss 8 true positives. In contrast, P_{auto} contains all 184 true positives of `defconfig` with no false positives. Furthermore, when enabling more configuration in `allyesconfig`, more permission checks are compiled into kernel IR, P_{user} misses more checks, as it uses the same input for both `defconfig` and `allyesconfig`. Therefore, the precision of P_{auto} is better than P_{user}.

Wrapper detection comparison. For brevity, we term wrappers identified based on automatic-detected checks as W_{auto} and the one identified based on user-provided checks as W_{user}. W_{auto} has fewer false positives than W_{user}. In `defconfig` kernel, the false positive rates of W_{user} are around 90%, 84%, and 90% for DAC, Capabilities and LSM, respectively. The rates are even higher in `allyesconfig`. On the contrary, the false positive rates of W_{auto} for `defconfig` are 0%, 0% and 3%, and for `allyesconfig` are 27%, 36%, and 14%, respectively.

Note that some wrappers in W_{user} are identified as basic checks in P_{auto}. For example, in `defconfig`, 4 of 11 DAC wrappers in Table 6a are automatically detected as basic checks in Table 6b, which explains why W_{auto} has one fewer DAC wrapper than W_{user}. In reference to the aggregate of both basic checks and wrappers, automatic-detected results include all true positives of user-provided ones and contain more checks.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

7.4 Manual Review of Warnings

The manual review process of reported warnings is to determine whether a privileged function identified by PeX (§6.4) 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 45 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 during program execution, resulting in false positives. One may devise a solution solving path constraints as in symbolic execution engines [36] to address this problem, PeX currently does not do so.

We conservatively reported 45 bug warnings to the community in total and 9 of them are newly detected by using automatic detection of permission checks. In total, 17 of them are confirmed by developers.

7.5 Discussion of Security Bug Findings

7.5.1 Missing Check

Figure 2b is one of the confirmed missing LSM checks (LSM-28). 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-28 missing check in `xfs_file_ioctl` led to another interesting discussion among kernel developers [37]. With this interface, a userspace program may perform low-level file system operations, but `security_inode_readlink` 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 [15]. 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 reported bugs 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-26 and LSM-27 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-19 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`.

`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 benefit of fine-grained capability-based access control.

The CAP-20 and CAP-21 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-20 and CAP-21 are worthwhile for further investigation.

The newly detected bug CAP-22 is in `rawsock_create` function. This function calls the privilege function `sk_alloc` by checking capability `CAP_SYS_ADMIN` in the init namespace by the permission check `capable`. However, it requires to check the capability in the user namespace using `ns_capable`. The kernel developers confirmed this bug and we already submitted a patch to fix it.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

TABLE 7: Bugs Reported By PeX. **C**onfirmed, **I**gnored or **P**ending.

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_SYS_RAWIO`	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	drivers/char/random.c	`_extract_crng()`	missing `CAP_SYS_ADMIN`	I
CAP-14	kernel/cgroup/cgroup-v1.c	`cgroup1_reconfigure()`	redundant `CAP_SYS_ADMIN`	C
CAP-15	drivers/scsi/sg.c	`sg_ioctl_common()`	missing `CAP_SYS_ADMIN` or `CAP_SYS_RAWIO`	I
CAP-16	kernel/audit.c	`audit_multicast_unbind()`	missing `CAP_AUDIT_READ`	I
CAP-17	net/unix/af_unix.c	`unix_create1()`	missing `CAP_NET_RAW`	I
CAP-18	drivers/isdn/mISDN/socket.c	`base_sock_create()`	missing `CAP_NET_RAW`	C
CAP-19	block/bsg.c	`bsg_ioctl`	inconsistent/missing `CAP_SYS_ADMIN`	C
CAP-20	kernel/sys.c	`prctl_set_mm_exe_file`	inconsistent capability check	I
CAP-21	kernel/sys.c	`prctl_set_mm_exe_file`	inconsistent capability and namespace check	I
CAP-22	net/nfc/rawsock.c	`rawsock_create()`	inconsistent namespace check	C
CAP-23	block/scsi_ioctl.c	`blk_verify_command`	redundant check `CAP_SYS_RAWIO`	I
LSM-24	fs/ecryptfs/inode.c	`ecryptfs_removexattr(), _setxattr()`	missing `security_inode_removexattr()`	C
LSM-25	mm/mmap.c	`remap_file_pages`	missing `security_mmap_file()`	I
LSM-26	fs/binfmt_elf.c	`load_elf_binary()`	missing `security_kernel_read_file`	I
LSM-27	fs/binfmt_elf.c	`load_elf_library()`	missing `security_kernel_read_file`	I
LSM-28	fs/xfs/xfs_ioctl.c	`xfs_file_ioctl()`	missing `security_inode_readlink()`	C
LSM-29	kernel/workqueue.c	`wq_nice_store()`	missing `security_task_setnice()`	C
LSM-30	fs/ecryptfs/inode.c	`ecryptfs_listxattr()`	missing `security_inode_listxattr`	C
LSM-31	include/linux/sched.h	`comm_write()`	missing `security_task_prctl()`	C
LSM-32	fs/binfmt_misc.c	`load_elf_binary()`	missing `security_bprm_set_creds()`	I
LSM-33	drivers/android/binder.c	`binder_set_nice`	missing `security_task_setnice()`	I
LSM-34	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	missing `security_socket_bind`	I
LSM-35	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	missing `security_socket_listen`	I
LSM-36	fs/dlm/lowcomms.c	`tcp_create_listen_sock`	missing `security_socket_bind`	I
LSM-37	fs/dlm/lowcomms.c	`tcp_create_listen_sock`	missing `security_socket_listen`	I
LSM-38	fs/dlm/lowcomms.c	`sctp_listen_for_all`	missing `security_socket_listen`	I
LSM-39	net/socket.c	`kernel_bind`	missing `security_socket_bind`	I
LSM-40	net/socket.c	`kernel_listen`	missing `security_socket_listen`	I
LSM-41	net/socket.c	`kernel_connect`	missing `security_socket_connect`	I
LSM-42	net/ipv4/route.c	`ipv4_sk_update_pmtu()`	missing `security_sk_classify_flow`	I
LSM-43	net/socket.c	`__sys_accept4_file`	missing `security_socket_create`	I
LSM-44	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	redundant `security_socket_create`	C
LSM-45	fs/ocfs2/cluster/tcp.c	`o2net_open_listening_sock()`	redundant `security_socket_create`	C

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-23 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-44 and LSM-45 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.

7.6 Limitations and Future Work

In this section, we discuss PeX limitations and the future work.

Incomplete validation of detection results. In this paper, we propose automatic methods to identify permission checks and the associated privileged functions. However, there are no official documents that specify the complete set of permission checks and privileged functions. As a result, we have no ground truth to evaluate our detection methods. We tried our best to build a ground truth and evaluate our approach upon it. However, the built ground truth might not be complete due to the limitation of our domain knowledge. Therefore, our future work is to build the complete ground truth and evaluate the detection methods thoroughly. Moreover, to build the complete ground truth, we plan to open our detection results to the public, so that kernel developers can contribute their domain knowledge to perfect the set of permission checks, privileged functions and their mappings.

Imperfect coverage of privileged functions. The privileged-function-oriented approach adopted by PeX might miss certain cases. The critical resources protected by permission checks might not be accessed through privileged functions. One such case is in `timerslack_ns_write`, after the capability checking, the critical field `p->timer_slack_ns` is updated directly, without using any privileged functions [38]. Therefore, to improve the analysis coverage, our future work is to identify the critical resources automatically to complement the privileged-function-oriented approach.

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 [23] verifies the complete mediation of LSM hooks relying manually specified security rules. While [24] automates LSM hook placements utilizing manually

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3165368, IEEE Transactions on Dependable and Secure Computing

written specification of security sensitive operations. However, required manual processes are error-prone when applied to huge Linux code base. Edwards et al. [27] proposed to use dynamic analysis to detect LSM hook inconsistencies. While PeX is using static analysis, can achieve better code coverage.

AutoISES [22] 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 [25], [26], [28] 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. On 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 [39] 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 [40].

Sparse [41] 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** [42] 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 [43], **Check-it-again** [34] focus on detecting time of check to time of use (TOCTTOU) bugs. **Dr. Checker** [44] is designed for analyzing Linux kernel drivers. It adopts the modular design, allowing new bug detectors to be plug-in easily. **KINT** [45] applies taint analysis to detect integer errors in Linux kernel while **UniSan** [46] leverages the same analysis to detect uninitialized kernel memory leakages to the userspace. **Chucky** [47] 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 [48] is an annotation based static analysis framework, and it can detect security rule violations in Linux. **APISan** [49] 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 [50]. **RoleCast** [51] leverages software engineering patterns to detect missing security checks in web applications. **TESLA** [52] 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** [53] 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 [54] is a tool designed for detecting semantic bugs in filesystem while **PScout** [55] is a static analysis tool for validating Android permission checking mechanisms. **Kratos** [56] 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 [8], 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 identify permission check functions and infer mappings between permission checks and privileged functions. Therefore PeX can 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 45 permission check bugs of DAC, Capabilities, and LSM, 17 of which have already been confirmed by the kernel developers. PeX source code is available at https://github.com/Jeimon/PermCheck, 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.

References

[1] R. S. Sandhu and P. Samarati, “Access control: principle and practice,” *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[2] “capabilities - overview of linux capabilities,” http://man7.org/linux/man-pages/man7/capabilities.7.html.

[3] S. Smalley, C. Vance, and W. Salamon, “Implementing selinux as a linux security module,” *NAI Labs Report*, vol. 1, no. 43, p. 139, 2001.

[4] “Apparmor,” https://gitlab.com/apparmor/apparmor/wikis/home/.

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

[6] Y. Sui and J. Xue, “Svf: interprocedural static value-flow analysis in llvm,” in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 265–266.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

[7] D. Gens, S. Schmitt, L. Davi, and A.-R. 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.

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

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

[10] L. Qiu, Y. Zhang, F. Wang, M. Kyung, and H. R. Mahajan, "Trusted computer system evaluation criteria," in *National Computer Security Center*. Citeseer, 1985.

[11] N. C. S. C. (US), *A guide to understanding discretionary access control in trusted systems*. National Computer Security Center, 1987, vol. 3.

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

[13] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security module framework," in *Ottawa Linux Symposium*, vol. 8032, 2002, pp. 6–16.

[14] "Mandatory access control," https://en.wikipedia.org/wiki/Mandatory_access_control.

[15] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android." in *NDSS*, vol. 310, 2013, pp. 20–38.

[16] "Virtual file system," https://en.wikipedia.org/wiki/Virtual_file_system.

[17] B. Hardekopf and C. 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*. IEEE Computer Society, 2011, pp. 289–298.

[18] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 2009, pp. 126–135.

[19] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 290–299.

[20] B. Hardekopf and C. Lin, "Exploiting pointer and location equivalence to optimize pointer analysis," pp. 265–280, 2007.

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

[22] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "Autoises: Automatically inferring security specification and detecting violations." in *USENIX Security Symposium*, 2008, pp. 379–394.

[23] X. Zhang, A. Edwards, and T. Jaeger, "Using cqual for static analysis of authorization hook placement." in *USENIX Security Symposium*, 2002, pp. 33–48.

[24] V. Ganapathy, T. Jaeger, and S. Jha, "Automatic placement of authorization hooks in the linux security modules framework," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 330–339.

[25] D. Muthukumaran, N. Talele, T. Jaeger, and G. Tan, "Producing hook placements to enforce expected access control policies," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2015, pp. 178–195.

[26] V. Ganapathy, T. Jaeger, and S. Jha, "Towards automated authorization policy enforcement," in *Proceedings of Second Annual Security Enhanced Linux Symposium*. Citeseer, 2006.

[27] A. Edwards, T. Jaeger, and X. 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*. ACM, 2002, pp. 225–234.

[28] D. Muthukumaran, T. Jaeger, and V. Ganapathy, "Leveraging choice to automate authorization hook placement," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 145–156.

[29] C. Lattner and V. 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*, ser. CGO '04, 2004, pp. 75–.

[30] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 37–48.

[31] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019.

[32] "capabilities(7) - linux manual page," https://man7.org/linux/man-pages/man7/capabilities.7.html, (Accessed on 02/09/2022).

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

[34] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of ACM conference on Computer and communications security*. ACM, 2018.

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

[36] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

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

[38] "timerslack_ns_write," https://elixir.bootlin.com/linux/v4.18.5/source/fs/proc/base.c#L2361.

[39] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Acm sigops operating systems review*, vol. 42, no. 4. ACM, 2008, pp. 247–260.

[40] P. Wang, J. Krinke, K. Lu, G. Li, and S. 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.

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

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

[43] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," 2018.

[44] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. checker: A soundy analysis for linux kernel drivers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*. USENIX Association, 2017, pp. 1007–1024.

[45] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with kint." in *OSDI*, vol. 12, 2012, pp. 163–177.

[46] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 920–932.

[47] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 499–510.

[48] J. Yang, T. Kremenek, Y. Xie, and D. 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*. ACM, 2003, pp. 321–334.

[49] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing api usages through semantic cross-checking." in *USENIX Security Symposium*, 2016, pp. 363–378.

[50] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 57–72.

[51] S. Son, K. S. McKinley, and V. Shmatikov, "Rolecast: finding missing security checks when you do not know what checks are," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 1069–1084.

[52] J. Anderson, R. N. Watson, D. Chisnall, K. Gudka, I. Marinos, and B. Davis, "Tesla: temporally enhanced system logic assertions," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 19.

[53] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "Jigsaw: Protecting resource access by inferring programmer expectations." in *USENIX Security Symposium*, 2014, pp. 973–988.

[54] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 361–377.

[55] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[56] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework." in *NDSS*, 2016.

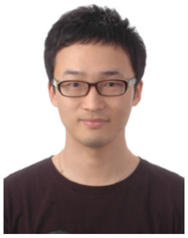
1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.

Jinmeng Zhou is currently working toward the Ph.D. degree in the Department of Computer Science, Zhejiang University, Zhejiang, China. She received the B.S. degree in information security from Wuhan University 2019. Her research interests include OS kernel security and static program analysis.

Tong Zhang is a compiler engineer at Samsung Electronics. He earned his PhD degree from Virginia Tech in 2019. He is interested in computer architecture, programming language and operating systems.

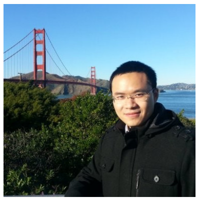
Wenbo Shen is currently a ZJU 100-Young Professor at Zhejiang University, China. He received the Ph.D. degree from the Computer Science Department of North Carolina State University in 2015. His research interests are system security and software security, including container security, OS kernel security, and program analysis using LLVM/clang.

Dongyoon Lee is an assistant professor in Department of Computer Science at Stony Brook University. He received the Ph.D. (2013) degree in Computer Science and Engineering at the University of Michigan, Ann Arbor under the guidance of Prof. Satish Narayanasamy. Before joining Stony Brook University, he worked as an assistant professor at Virginia Tech (2014-2019). His research interests are software/hardware systems, program analysis, software reliability and security.



Changhee Jung (Member, IEEE) received the PhD degree in computer science from Georgia Institute of Technology, Atlanta, Georgia, in 2013. He is currently an associate professor of computer science with Purdue University, West Lafayette. His research interests span the field of compilers and computer architecture, with an emphasis on performance, reliability, and security.

Ahmed Azab Ahmed Azab is a software engineer and researcher at Facebook. He received the Ph.D. degree in computer science from North Carolina State University, in 2011. In 2013, he joined Samsung Research America to lead a Research and Development group that developed multiple technologies to enhance the security of Samsung consumer devices. His research interests are in building systems and technologies to provide a trusted verifiable environment for secure code execution, as well as to prevent or mitigate dangerous forms of cyber-security attacks.



Ruowen Wang received his PhD at North Carolina State University. His research interests include system security, mobile security, access control, etc. He is currently working on malware analysis at Google. Before that, he was doing Android security research at Samsung Research America.

Kui Ren received degrees from three different majors, i.e., his Ph.D in Electrical and Computer Engineering from Worcester Polytechnic Institute, USA, in 2007, M.Eng in Materials Engineering in 2001, and B.Eng in Chemical Engineering in 1998, both from Zhejiang University, China.

He is currently a Professor and Associate Dean of College of Computer Science and Technology at Zhejiang University, where he also directs the Institute of Cyber Science and Technology. Kuis current research interests include Data Security, IoT Security, AI Security, and Privacy.

Peng Ning is currently in Google. He was a full professor of Computer Science in the College of Engineering at North Carolina State University. He received his PhD degree in Information Technology from George Mason University in 2001. Prior to his PhD study, he received an ME in Communication and Electronic Systems in 1997, and a BS degree in Information Science in 1994, both from University of Science and Technology of China. Peng Ning is a member of the ACM, the ACM SIGSAC, the IEEE, and the IEEE Computer Society.

1545-5971 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Zhejiang University. Downloaded on June 15,2022 at 08:11:56 UTC from IEEE Xplore. Restrictions apply.