ACHyb: A Hybrid Analysis Approach to Detect Kernel Access Control Vulnerabilities

Yang Hu
The University of Texas at Austin
Austin, Texas, USA
huyang@utexas.edu

Wenxi Wang
The University of Texas at Austin
Austin, Texas, USA
wenxiw@utexas.edu

Casen Hunger
The University of Texas at Austin
Austin, Texas, USA
casen.h@utexas.edu

Riley Wood
The University of Texas at Austin
Austin, Texas, USA
riley.wood@utexas.edu

Sarfraz Khurshid
The University of Texas at Austin
Austin, Texas, USA
khurshid@ece.utexas.edu

Mohit Tiwari
The University of Texas at Austin
Austin, Texas, USA
tiwari@austin.utexas.edu

ABSTRACT

Access control is essential for the Operating System (OS) security. Incorrect implementation of access control can introduce new attack surfaces to the OS, known as Kernel Access Control Vulnerabilities (KACVs). To understand KACVs, we conduct our study on the root causes and the security impacts of KACVs. Regarding the complexity of the recognized root causes, we particularly focus on two kinds of KACVs, namely KACV-M (due to missing permission checks) and KACV-I (due to misusing permission checks). We find that over 60% of these KACVs are of critical, high or medium security severity, resulting in a variety of security threats including bypass security checking, privileged escalation, etc. However, existing approaches can only detect KACV-M. The state-of-the-art KACV-M detector called PeX is a static analysis tool, which still suffers from extremely high false-positive rates.

In this paper, we present ACHyb, a precise and scalable approach to reveal both KACV-M and KACV-I. ACHyb is a hybrid approach, which first applies static analysis to identify the potentially vulnerable paths and then applies dynamic analysis to further reduce the false positives of the paths. For the static analysis, ACHyb improves PeX in both the precision and the soundness, using the interface analysis, callsite dependence analysis and constraint-based invariant analysis with a stronger access control invariant. For the dynamic analysis, ACHyb utilizes the greybox fuzzing to identify the potential KACVs. In order to improve the fuzzing efficiency, ACHyb adopts our novel clustering-based seed distillation approach to generate high-quality seed programs. Our experimental results show that ACHyb reveals 76 potential KACVs in less than 8 hours and 22 of them are KACVs (19 KACV-M and 3 KACV-I). In contrast, PeX reveals 2,088 potential KACVs in more than 11 hours, and only 14 of them are KACVs (all KACV-M). Furthermore, ACHyb successfully uncovers 7 new KACVs, and 2 of them (1 KACV-M and 1 KACV-I) have been confirmed by kernel developers.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468627

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Software security engineering**.

KEYWORDS

Program Analysis, Access Control, Operating System

ACM Reference Format:
Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. 2021. ACHyb: A Hybrid Analysis Approach to Detect Kernel Access Control Vulnerabilities. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468627

1 INTRODUCTION

Access control [48] is a fundamental and indispensable security mechanism for the OS kernel. It secures the resources in the OS by blocking the accesses which violate authorization rules. The Linux kernel provides several access control modules, including Access Control List (ACL) [15], Linux Capabilities [23], Linux Security Module (LSM) [51], etc. Thanks to these modules, the Linux kernel has been applied in many security sensitive environments [14].

Access control provides a security guarantee to the *privileged functions* (i.e., the kernel functions implementing the security critical kernel functionalities), which ensures that privileged functions can be called only when the caller/user has the permission. This is usually achieved through the *access control decision* returned by the *permission check* (i.e., the kernel function verifying if the permissions granted to the caller/user are consistent with the caller/user operations). The security guarantee is realized in two steps: 1) generating an access control decision via a permission check before calling a privileged function, and 2) enforcing the access control decision in the control flow so that the privileged function will not be called if the access control decision is denied. However, attackers could break the security guarantee, if the access control is incorrectly implemented in the above two steps. This security issue is known as the Kernel Access Control Vulnerability (KACV).

To understand KACVs, we conduct an empirical study in this paper to learn the root causes and the security impact of KACVs. We first collected 101 KACVs from the National Vulnerability Database [40]. After manual inspections, we classify KACVs into three categories based on our identified three root causes. Given the

complexity of one root cause, in this paper, we only focus on two categories, namely KACV-M which is due to missing permission checks, and KACV-I which is due to misusing permission checks. According to the Common Vulnerability Scoring System (CVSS), 24.4% of the KACV-M and KACV-I were scored as high or critical security severity, and 38.8% were scored as the medium severity. In addition, our study found that KACV-M and KACV-I cause a variety of security threats, including bypass security checking, privileged escalation, denial of services, etc.

To the best of our knowledge, existing approaches can only detect KACV-M, and no work has been proposed to detect KACV-I. Zhang et al. [62] propose a static analysis tool called PeX, which is the state-of-the-art KACV-M detector. PeX conducts its static analysis in three steps including 1) the permission check identification, 2) the privileged function identification, and 3) the invariant analysis over the permission checks and privileged functions to uncover KACVs. To identify permission checks, PeX requires users to provide an incomplete list of permission checks. It then performs program slicing [28] to identify the wrappers of the initially provided permission checks as the new permission checks. Next, PeX finds an over-approximation of the privileged functions with a control-flow analysis to collect the kernel functions which always execute after permission checks. Last, it performs a control-flow based invariant analysis to search for the potentially vulnerable paths where a privilege function is not preceded by any permission checks. However, PeX suffers from significant false-positive rates due to the limitation in each step. First, the permission check identification is unsound especially when the user-provided permission checks lack diversity. Second, the privileged function identification is imprecise due to the weak over-approximation. Third, the invariant analysis is also imprecise due to the weak invariant.

To mitigate the limitations of PeX, we present a precise, scalable KACV detector called ACHyb which is capable of detecting both KACV-M and KACV-I. ACHyb is a *hybrid* analysis approach, which first applies static analysis to identify the potentially vulnerable paths and then applies dynamic analysis to further reduce the false positives of the paths. For the static analysis, ACHyb follows the three steps of PeX, but with its own improvements for each step to enhance both the precision and the soundness. For permission check identification, ACHyb performs a semi-automated *interface analysis* which is a *soundy* (i.e., mostly sound [32]) approach. For privileged function identification, instead of using control-flow analysis as PeX, ACHyb performs our proposed *callsite dependency analysis* which is a data-flow analysis that could significantly improve the precision. For invariant analysis, ACHyb proposes a stronger invariant and performs a constraint-based analysis to check the invariant. To improve the efficiency, instead of conducting the standard inter-procedural analysis, ACHyb conducts the lightweight *intra-procedural analysis* by exploiting the features of access control .

Moreover, instead of requesting human effort to do the manual inspection as PeX, ACHyb applies dynamic analysis to reduce the false positives of the potentially vulnerable paths identified. The idea is to identify the feasible potentially vulnerable paths in which the access control decisions are either missing (potential KACV-M) or denied (potential KACV-I). To achieve this, ACHyb injects invariant checks on the potentially vulnerable paths and conducts greybox fuzzing to trigger these checks. To improve the fuzzing efficiency, ACHyb adopts our novel clustering-based seed distillation approach to generate high-quality seed programs.

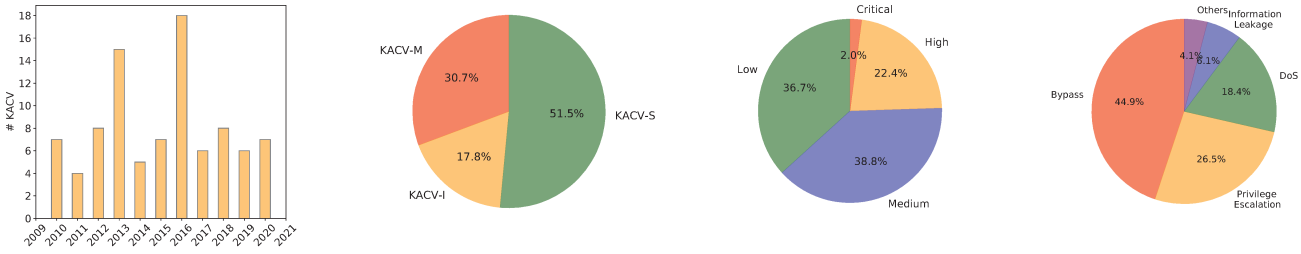
For static analysis, we implement ACHyb on top of the LLVM pass framework [45]. For dynamic analysis, we build ACHyb based on the greybox fuzzer called Syzkaller [57]. We perform an empirical evaluation of ACHyb on the Linux kernel v4.18.5. As a result, ACHyb reports 76 potential KACVs, 22 of which are KACVs including 19 KACV-M and 3 KACV-I. In contrast, PeX reports 2,088 potential KACVs, 14 of which are KACV-M. Besides, the KACVs detected by ACHyb contain all the KACVs detected by PeX. We report 7 new KACVs (5 KACV-M and 2 KACV-I) to kernel developers. By the time of the paper publication, 2 new KACVs (1 KACV-M and 1 KACV-I) have been confirmed. The results show that ACHyb is not only more precise than PeX, but also capable of detecting new KACVs. In addition, ACHyb takes less than 8 hours to detect the KACVs while PeX takes more than 11 hours, which shows that ACHyb is more efficient than PeX. The source code of ACHyb and the dataset of our study are publicly available at https://github.com/githubhuyang/achyb. The contributions of this paper are:

- **Study.** We did an empirical study on KACVs mainly in two aspects: the root causes and the security impacts of KACVs.
- **Approach.** We present ACHyb, which combines static and dynamic analysis to detect both KACV-M and KACV-I precisely and efficiently. To the best of our knowledge, ACHyb is the first tool that is capable of detecting KACV-I.
- **Implementation.** We implement ACHyb on top of the LLVM and Syzkaller with about 5,000 lines of code.
- **Empirical Evaluation.** We did an empirical evaluation of ACHyb on the Linux kernel v4.18.5 The experimental results show that ACHyb is more precise and scalable than the state-of-the-art tool PeX.
- **Practical Impacts** ACHyb is able to detect 7 new KACVs, 2 of which have been confirmed by the kernel developers.

2 A STUDY ON KACVS

We study the KACVs mainly in two aspects: the root causes and the security impact of KACVs. In order to get KACVs, we first collect all the CVE reports related to KACVs from the National Vulnerability Database [40]. We use the `cve-search` tool [11] to find the CVE reports that contain the keywords related to the access control, such as "ACL", "capability", "permission", etc. We filtered out the old CVE reports on the kernel version lower than v2.6, since we want to focus on the KACVs in the newer kernel versions. As a result, 101 CVE reports were collected. Fig. 1a shows the number of CVE reports related to the KACVs in the recent 10 years. We can observe that from 4 to 18 KACVs were reported each year since 2010. Next, we extract the key information from the collected CVE reports including the vulnerability descriptions, the CVSS ratings (version 3.0), and the patches that fix the KACVs.

We manually inspect the vulnerability descriptions and patches in our collected CVE reports to identify the root causes of the KACVs. As a result, we classify the KACVs into three categories based on the identified root causes, as shown in Fig. 1b. The first category is called KACV-M, which refers to the KACVs due to *missing permission checks*. The second category is called KACV-I, which refers to the KACVs due to *misusing permission checks*. The third



(a) KACV counting across years (b) Root causes (c) Severity in CVSS (d) Security threats

Figure 1: Statistical results of our KACV study.

```
1  /* fs/namespace.c in the Linux kernel v2.6.21 */
2  static int do_change_type(...) {
3    ...
4    // Missing the access control decision.
5    /* Patch:
6       if (!capable(CAP_SYS_ADMIN))
7       return -EPERM;*/
8    ...
9    // Privileged function
10   change_mnt_propagation(m, type);
11   ...
12 }
```

Figure 2: An example of KACV-M.

```
1  /* fs/read_write.c in the Linux kernel v4.9 */
2  int vfs_dedupe_file_range(...) {
3    // Direct callsite to a permission check
4    bool is_admin = capable(CAP_SYS_ADMIN);
5    // Direct callsite to a non-privileged funtion
6    ret = clone_verify_area(...);
7    ...
8    for (...) {// condition C1
9      ...
10     if (...) {// condition C2
11       ...
12     } else if (!(is_admin||...)) {//incorrect
       condition C3
13       ...
14     } else {
15       // Indiret callsite to a privileged function
16       deduped = dst_file->f_op->dedupe_file_range(...)
       ;
17       ...
18     }
19   }
20 }
```

Figure 3: An example of KACV-I.

category is called KACV-S, which refers to the KACVs due to *incorrect internal access control states*. The percentage of KACV-M, KACV-I, and KACV-S is 30.7%, 17.8%, and 51.5%, respectively. Due to the complexity of the internal access control states and the state transitions, we leave KACV-S detection as our further research work. In this paper, we only focus on studying and detecting KACV-M and KACV-I. Note that, to the best of our knowledge, no existing work has been proposed to detect KACV-I.

We show two examples of KACV-M and KACV-I, respectively. Fig. 2 shows a KACV-M example. Obviously, the function do_change_type misses calling the function capable to check if the caller has the permission (i.e., CAP_SYS_ADMIN capability) to call the privileged function change_mnt_propagation (line 10). Fig. 3 shows a KACV-I example. The function vfs_dedupe_file_range calls the permission check function capable to query if the caller has the permission (i.e., CAP_SYS_ADMIN capability) to call the privileged function stored in the function pointer dedupe_file_range (line 16). However, due to the incorrect usage of the access control decision (is_admin, line 12), there exists a feasible path where the privileged function is called when the access control decision is false (i.e., denied).

Fig. 1c shows the vulnerability severity of KCAV-M and KACV-I which was measured in CVSS v3.0 ratings. 24.4% of the KCAV-M and KACV-I were scored as high or critical severity, and 38.8% were scored as the medium severity. Fig. 1d shows the statistics of the security threats caused by KCAV-M and KACV-I, based on the CVE reports. The most frequent security threat is bypassing security checking (44.9%); the second frequent threat is the privilege escalation (26.5%); followed by the denial of service (18.4%); the next is the information leakage (6.1%). The data shows that KACV-M *and* KACV-I *have profound security impacts on the Linux kernel.* The goal of this paper is to propose a precise and scalable approach to detect KACV-M and KACV-I.

3 METHODOLOGY

3.1 Overview of ACHyb

The overview of ACHyb is shown in Fig. 4. The input of ACHyb is the Linux kernel under examination encoded in LLVM Intermediate Representation (IR); the output is the detected KACVs (KACV-M or KACV-I). ACHyb is composed of two phases: the static analysis phase (left part of Fig. 4) and the dynamic analysis phase (right part of Fig. 4). The static phase firstly takes in the kernel IR and outputs the potentially vulnerable paths, while the dynamic phase reduces false positives of those paths and outputs the KACVs.

Specifically, in the static analysis phase, ACHyb performs a semi-automated interface analysis to identify the permission checks (Section 3.2.1). It then conducts a *callsite dependency* analysis to identify the privileged functions (Section 3.2.2). Finally, ACHyb proposes a stronger access control invariant over the detected permission checks and privileged functions, and performs a constraint-based invariant analysis to get the potentially vulnerable paths (Section 3.2.3). In the dynamic analysis phase, ACHyb firstly injects run-time invariant checks on these potentially vulnerable paths

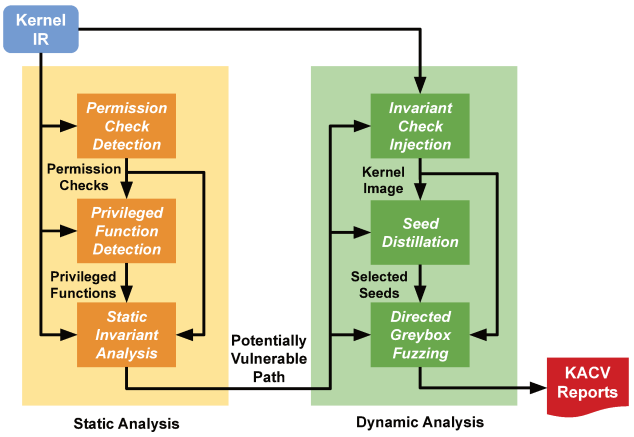


Figure 4: Overview of ACHyb

(Section 3.3.1). Next, ACHyb conducts seed distillation to generate high-quality seed programs, and finally applies the greybox fuzzing using those seed programs to trigger the injected invariant checks and get the KACVs (Section 3.3.2).

3.2 Static Analysis

3.2.1 Permission Check Identification. The detection of KACV-M and KACV-I requires the full list of permission checks. Since most permission checks are not well documented, researchers are seeking to employ program analysis to automatically collect the permission checks. PeX [62] identifies the permission checks by utilizing call graph slicing to search the wrappers of those permission checks, given an incomplete list of the user-provided permission checks. This approach is unsound, especially when the user-provided permission checks lack diversity.

To mitigate the problem, our insight is that we can firstly get an over-approximation of the permission checks and further remove the false positives with manual efforts. However, a weak over-approximation might significantly increase the manual efforts. To solve this problem, ACHyb performs a *soundy* (i.e., mostly sound [32]) interface analysis to give a good approximation of the permission checks. Our approximation is based on the two key observations. One is that a permission check is usually implemented as an *access control interface*, which refers to a kernel function in the access control module that can be called by the functions outside the module. The other one is that a permission check usually returns an access control decision through a boolean/integer variable. Based on these two observations, ACHyb collects the access control interfaces with boolean or integer return types to serve as a soundy approximation of the permission checks.

After obtaining the approximation set of the permission checks, instead of asking users to manually inspect every access control interface, ACHyb only requests users to manually inspect a few selected ones. ACHyb achieves this in two folds. First, it performs a dependency analysis on the return variables of the access control interfaces to divide the interfaces into several equivalence classes. The equivalence class of access control interfaces is defined in Definition 3.1. Here, we say that a variable x depends on a variable y iff the

```
1  /* kernel/capability.c in the Linux kernel v4.9 */
2  ...
3  bool capable(int cap) {
4    return ns_capable(&init_user_ns, cap);
5  }
6  EXPORT_SYMBOL(capable);
7
8  bool ns_capable(struct user_namespace *ns, int cap) {
9    return ns_capable_common(ns, cap, true);
10 }
11 EXPORT_SYMBOL(ns_capable);
12
13 bool ns_capable_noaudit(struct user_namespace *ns, int
       cap) {
14   return ns_capable_common(ns, cap, false);
15 }
16 EXPORT_SYMBOL(ns_capable_noaudit);
17 ...
```

Figure 5: An example of permission check detection.

value of y determines the value of x. Second, for each equivalence class, we ask users to manually inspect only one representative of the class. If the representative is manually identified as a permission check, then all the interfaces in the same equivalence class are automatically taken as the permission checks; otherwise, all the interfaces in the equivalence class are not taken as the permission checks.

Definition 3.1 (The Equivalence Class of Access Control Interfaces)**.** The interface f_x and the interface f_y are in the same equivalence class iff there exists a kernel function f_z such that the return variables of both f_x and f_y depend on the return variable of f_z. □

To illustrate, we use an example in Fig. 5. Consider the three access control interfaces, which are `capable`, `ns_capable`, and `ns_capable_noaudit`. Based on the above definition of the equivalence class, we know that the interface `capable` is equivalent to the interface `ns_capable`, as the return variable of the interface `capable` depends on that of the interface `ns_capable` (f_z can be f_x or f_y in Definition 3.1). Besides, the interface `ns_capable` is equivalent to the interface `ns_capable_noaudit`, as both of their return variables depend on the return variable of the interface `ns_capable_common`. Therefore, the three interfaces are classified into one equivalence class. If users manually identify any one of the three interfaces (e.g., the interface `ns_capable`) as a permission check, then the others (e.g., the interface `capable` and the interface `ns_capable_noaudit`) will be automatically classified as permission checks. In the end, users only manually inspects one interface, but identifies three permission checks.

3.2.2 Privileged Function Identification. Besides the permission checks, the detection of KACV-M and KACV-I also requires a list of privileged functions. Unlike the permission checks which can be approximated by syntactic features, identifying privileged functions needs to exploit the semantic features. To address this problem, PeX [62] over-approximates the privileged functions by finding the kernel functions whose executions are *dominated* [37] by the permission checks. To be specific, if a callsite of the kernel function f_x always executes after a callsite of a permission check, then PeX identifies the function f_x as the privileged function. However, this over-approximation is weak and might cause high false-positive

rates. To illustrate, we use the KACV-I example shown in Fig. 3. Since the callsite of a permission check (line 4) dominates the callsite of the function clone_verify_area (line 6), PeX classifies the function clone_verify_area as a privileged function. However, the function clone_verify_area is actually a non-privileged function.

To lower the false-positive rates, instead of using the control-flow analysis as PeX, ACHyb applies a more fine-grained data-flow analysis to over-approximate the privileged functions. The data-flow analysis is actually a *callsite dependency* analysis which searches for the kernel functions whose executions depend on the access control decisions. The definition of the callsite dependency is given in Definition 3.2. For the KACV-I example in the Fig. 3, the callsite of a privileged function (line 16) depends on the callsite of a permission check capable (line 4). Because the access control decision is_admin returned by the permission check is used in a condition (line 12) which determines the execution of the privileged function.

Definition 3.2 (Callsite Dependency). A callsite c_x depends on a callsite c_y iff the return value of c_y controls the execution of the callsite c_x. □

Note that identifying such dependency usually needs an *inter-procedural* data-flow analysis, which is notorious for the scalability limitation [21]. To mitigate the issue, ACHyb performs an *intra-procedural* data-flow analysis to identify the dependencies. Compared to the inter-procedural analysis which extracts every data-flows in the kernel, the intra-procedural analysis only focuses on identifying the data-flows inside the body of each individual kernel function, thereby achieving much better efficiency and scalability. Moreover, the intra-procedural analysis would not cause much accuracy loss in approximating the privileged functions, as we observe that for most kernel functions excluding the permission checks, the access control decision is only used *inside* the function where it is defined and rarely used as an argument or a return variable. In other words, the access control decision is rarely propagated across the kernel functions.

Algorithm 1 demonstrates our method to detect the privileged functions. ACHyb firstly gets the direct/indirect callsites in each of the kernel functions excluding the permission checks (line 2-5). ACHyb then conducts a callsite dependency analysis to get all the callsites that depend on the collected callsites of permission checks, and store them in the set S' (line 6-8). Next, for each callsite in the set S' excluding the ones of permission checks, ACHyb identifies its callees as the privileged functions (line 10-12). To illustrate, Fig. 6 presents the intra-procedural data-flow analysis for the KACV-I example shown in Fig. 3. ACHyb firstly detects all the callsites inside the function vfs_dedupe_file_range, including the callsite of the function capable (line 4), and the callsite of the functions btrfs_dedupe_file_range and xfs_file_dedupe_range (line 16). Given a permission check list detected in the last section, ACHyb identifies that only the function capable is a permission check. Next, ACHyb conducts a callsite dependency analysis based on the access control decision is_admin returned by the permission check capable. Since is_admin is used in an if condition (line 12) and controls the execution of the functions btrfs_dedupe_file_range and xfs_file_dedupe_range, ACHyb classifies these two functions as the privileged functions.

Algorithm 1 Privileged function detection algorithm.

Input: the kernel intermediate representation R
the set of permission checks F_{perm}
Output: the set of privileged functions F_{priv}

1: **function** $GET\_PRIV\_FUNCTIONS(R, F_{perm})$
2: $F_{non\_perm} \leftarrow R.get\_functions() - F_{perm}$
3: $F_{priv} \leftarrow \emptyset$
4: **for** $f \in F_{non\_perm}$ **do**
5: $S \leftarrow f.get\_callsites()$
6: **for** $s \in S$ **do**
7: **if** $s.is\_perm\_callsite()$ **then**
8: $S' \leftarrow callsite\_dependence\_analysis(f, s)$
9: **for** $s' \in S'$ **do**
10: **if** $\neg s'.is\_perm\_callsite()$ **then**
11: $F_{callee} \leftarrow R.get\_callees(s')$
12: $F_{priv} \leftarrow F_{priv} \cup F_{callee}$
13: **return** F_{priv}

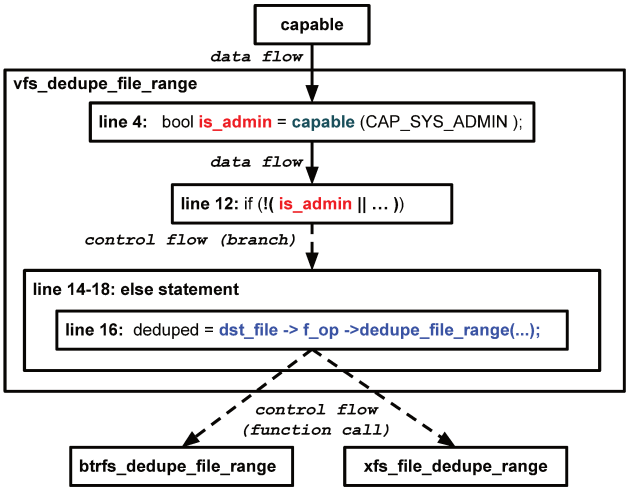


Figure 6: An example to illustrate our privileged function detection algorithm. btrfs_dedupe_file_range and xfs_file_dedupe_range are identified as privileged functions, as their callsite in line 16 depends on the callsite of the capable function (a permission check) in line 4.

3.2.3 Static Invariant Analysis. The goal of the invariant analysis is to identify the potentially vulnerable paths where the callsites of privileged functions are not protected by the permision checks. The major limitation of the existing invariant analysis for KACV detection is that the defined invariant is not strong enough which may cause false negatives. PeX [62] proposes an invariant: *for each path in the control-flow graph, every callsite of a privileged function must be preceded by at least one callsite of a permission check.* This invariant analysis cannot reveal the KACV-I in Fig. 3, as the indirect callsite of the privileged function(s) (line 16) always executes after the callsite of the permission check capable (line 4).

To mitigate the limitations, ACHyb firstly proposes a stronger access control invariant: *a privileged function should not be executed*

when its corresponding access control decision is denied or not generated at all. One straightforward way to check this invariant is to apply Symbolic Execution [5] on every path from each entry of the kernel (i.e., the system calls) to the callsites of the privileged function, and solve the constraints collected along the paths. However, this method suffers from the path explosion and the high complexity of the path constraints. ACHyb adopts two strategies to make the constraint-based invariant analysis scalable.

One strategy is to perform an intra-procedural analysis instead of the inter-procedural analysis, based on the observation mentioned in Section 3.2.2 that the access control decision is rarely propagated across the kernel functions. For every path which reaches a privileged function, ACHyb only collects the constraints inside the *caller* of the privileged function. By replacing the inter-procedural analysis with the intra-procedural analysis, both the path space and the complexity of the constraints can be largely reduced. However, since only partial path constraints are collected, this analysis can only find *potentially* vulnerable paths. ACHyb further reduces false positives among these paths using dynamic analysis which is introduced in Section 3.3.

The other strategy is to perform the above analysis only on the non-privileged functions which call at least one privileged functions. This is based on our observation that the non-privileged functions usually need to request access control decisions before they call the privileged functions, while the privileged functions rarely request access control decisions with the assumption that their callers request and check the access control decisions beforehand. Besides, there is no need to conduct the analysis on permission checks, as they never call privileged functions.

Algorithm 2 shows our invariant analysis algorithm. ACHyb first collects all the kernel functions excluding the permission checks and the privileged functions. Inside each of these functions, ACHyb collects the callsites of the privileged functions (line 2-5). It then gets all the intra-procedural paths (with finite loop unrolling) that reach these callsites (line 6-7). For each path, ACHyb obtains the callsites of the permission checks in the path (line 8-9). If no such callsite is found, the path is taken as a potentially vulnerable path (line 10-11). Otherwise, ACHyb fetches the access control decision and the path constraints, and checks the satisfiability of the constraint which is the conjunction of the path constraints and an additional constraint stating that all the access control decisions are `denied` (line 13-15). If the constraint is satisfiable, which means that the path to the privileged function is feasible but the access control decisions in the path are denied, the path is then identified as a potentially vulnerable path (line 16-17).

We demonstrate our static invariant analysis using the relevant path conditions of the `KACV-I` example shown in Fig. 3. Since there is a callsite of a privileged function in line 16, ACHyb analyzes the path to the callsite inside the function `vfs_dedupe_file_range`. For simplicity, we consider the loop is unrolled only once. The path to the callsite of the privileged function includes lines 2-10, 12, 14-16. First, ACHyb collects 3 path constraints inside the function `vfs_dedupe_file_range`: C_1 ,$\neg C_2$, and $\neg C_3$. Next, ACHyb generates an additional constraint $C_4 := (is\_admin = false)$, which indicates that the access control decision of the function `capable` is `denied` (line 4). Then, ACHyb checks if the following constraint

Algorithm 2 Static invariant analysis algorithm.

Input: the kernel intermediate representation R
the set of permission checks F_{perm}
the set of privileged functions F_{priv}
Output: the set of potentially vulnerable paths P_{pot}

1: **function** $STATIC\_INV\_ANALYSIS(R, F_{perm}, F_{priv})$
2: $F_{other} \leftarrow R.get\_functions() - F_{perm} - F_{priv}$
3: $P_{pot} \leftarrow \emptyset$
4: **for** $f \in F_{other}$ **do**
5: $S_{priv} \leftarrow f.get\_priv\_callsites(F_{priv})$
6: **for** $s \in S_{priv}$ **do**
7: $P_{local} \leftarrow f.get\_paths(s)$ ▷ get paths inside f which reach the callsite s.
8: **for** $p \in P_{local}$ **do**
9: $S_{perm} \leftarrow p.get\_perm\_callsites(F_{perm})$
10: **if** $S_{perm} = \emptyset$ **then**
11: $P_{pot} \leftarrow P_{pot} \cup \{p\}$
12: **else**
13: $D \leftarrow p.get\_dec\_var(S_{perm})$
14: $C \leftarrow p.get\_path\_constraints()$
15: $C \leftarrow C \cup \{d = \texttt{denied} | d \in D\}$
16: **if** $is\_satisfiable(\bigwedge_{c \in C} c)$ **then**
17: $P_{pot} \leftarrow P_{pot} \cup \{p\}$
18: **return** P_{pot}

is satisfiable:

$$C_1 \wedge \neg C_2 \wedge \neg C_3 \wedge C_4.$$

Since the constraint is satisfiable, the path is identified as a potentially vulnerable path.

3.3 Dynamic Analysis

The dynamic analysis focuses on reducing false positives among the potentially vulnerable paths detected by the static analysis. ACHyb achieves this in two steps. First, ACHyb injects the run-time invariant checks to the kernel image, as introduced in Section 3.3.1. Then, ACHyb conducts greybox fuzzing to cover the potentially vulnerable paths so that the invariant checks can be triggered and the KACVs can be revealed, as presented in Section 3.3.2.

3.3.1 Invariant Check Injection. As mentioned in Section 3.2.3, the intra-procedural invariant analysis would cause false positives. To remedy this problem, ACHyb rigorously checks the access control invariant in the run time. For each potentially vulnerable path, ACHyb instruments the kernel with a run-time invariant check which is added exactly before the callsite of each privileged function. As introduced in Section 3.2.3, the static invariant analysis can detect two kinds of potentially vulnerable paths reachable to the privileged functions: the paths with missing permission checks and the paths with denied access control decisions. If a test execution triggers the run-time checks and covers the potentially vulnerable path with missing permission checks, the potentially vulnerable path is feasible and thus it is taken as the path that could reveal `KACV-M`. Similarly, if the potentially vulnerable path with denied access control decision is feasible, the path is considered to reveal the `KACV-I`. For the `KACV-I` example in Fig. 3, ACHyb injects a run-time

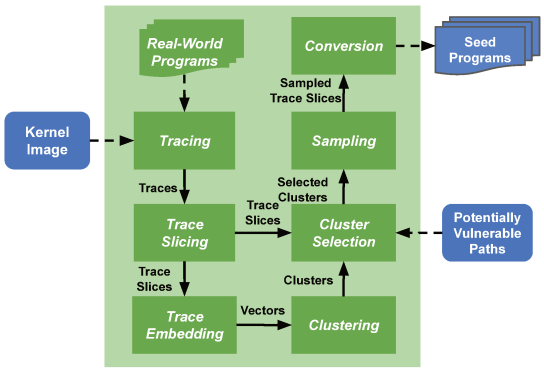


Figure 7: An overview of our seed distillation approach.

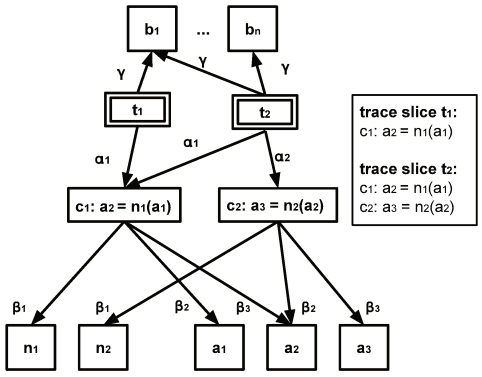


Figure 8: An example of trace embedding.

invariant check `assert(is_admin)` before calling the privileged function (line 16), to check if the potentially vulnerable path (line 4-10, 12, 14-16) is feasible when the access control decision `is_admin` is `false` (i.e., denied).

3.3.2 Greybox Fuzzing with Seed Distillation. Greybox fuzzing has been widely used in patch testing, bug detection, and bug reproduction [7, 9, 41]. ACHyb regards the invariant checks as the targets, and utilizes the greybox fuzzing to trigger them. To further improve the efficiency of fuzzing, we propose a novel clustering-based seed distillation approach to generate high-quality seed programs (i.e., sequences of system calls) which could guide the fuzzer to rapidly approach the invariant checks with less chance of being trapped. Fig. 7 briefly shows our seed distillation approach. ACHyb first collects execution traces from the real-world programs in the Linux Test Project (LTP) [49]. We choose LTP because it is well maintained (by many companies such as IBM, Cisco, Red Hat, etc) aiming at generating good test programs for the Linux development. It has also been applied by the state-of-the-art seed distillation tool called `Moonshine` [42]. After collecting the traces, ACHyb then performs program slicing [20, 34] to split the traces into trace slices. Next, ACHyb does the trace embedding to get the vector representation of each trace slice, and use the `scikit-learn` machine learning framework [8] to perform the K-means clustering [30] on the trace slices. ACHyb selects the clusters which are "closer" to the potentially vulnerable paths. Details about the trace embedding and the cluster selection are introduced in the following sections. For each selected cluster, ACHyb randomly samples a few trace slices and converts them into the seed programs. These seed programs are finally fed to the greybox fuzzer.

Trace Embedding. Inspired by the existing program embedding approaches [1, 2, 6, 47, 58], we propose our trace embedding approach to convert the trace slices into vector representations. First, we convert all the trace slices into our defined *multi-relation graph* which encodes the syntactic and semantic information of the trace slices. Second, we use the `PyTorch-BigGraph` system [29] to generate the vector representation for each node in the graph based on the information from their neighbors. Last, we extract the embedding of each node representing the trace slice as our trace embedding.

Formally, we define the multi-relation graph as a directed graph represented by a tuple (V, R, E), where V is a set of nodes, R is a set of relations, and E is a set of labeled directed edges $x \overset{r}{\mapsto} y$ where $x, y \in V$ and $r \in R$. The multi-relation graph depicts the following syntactic and semantic information in trace slices: each trace slice consists of at least one system call; each system call has its system call name, its arguments (if any) and its return value (if any); each trace slice covers a set of code branches. Accordingly, we define five node types to represent a trace slice (dented as t), a system call (denoted as c), the name of a system call (denoted as n), the argument/return value of a system call (denoted as a), and the branch covered by a trace slice (denoted as b), respectively. Besides, we define five relation types as follows:

- Trace-to-Call Relations: The relation α_i between the trace slice t and its i-th system call c_i is denoted as $t \overset{\alpha_i}{\mapsto} c_i$.
- Call-to-Name/Arguments/Return Relations: The relation β_1 between the system call c and its callname n is denoted as $c \overset{\beta_1}{\mapsto} n$; the relation β_2 between the system call c and its argument a is denoted as $c \overset{\beta_2}{\mapsto} a$; the relation β_3 between the system call c and its return value v is denoted as $c \overset{\beta_3}{\mapsto} v$.
- Trace-to-Coverage Relation: The relation γ between a trace slice t and the covered code branch b is denoted as $t \overset{\gamma}{\mapsto} b$.

Fig. 8 shows the multi-relation graph of a brief example. There are two trace slices t_1 and t_2 in the example. The trace slice t_1 contains one system call c_1, and the trace slice t_2 contains a sequence of two system calls c_1 and c_2. We get the vector representation of each node in the graph with the node embedding, and extract the vector representations of all the trace slice nodes (t_1 and t_2) as our trace embedding.

Cluster Selection. Intuitively, if the paths covered by a cluster of trace slices are similar to the potentially vulnerable paths, the seed programs converted from the trace slices in the cluster are inclined to cover the potentially vulnerable paths with a few mutations. Our cluster selection is based on this intuition. For each potentially vulnerable path, ACHyb selects the cluster which has the maximal path similarity to the path. The path similarity is defined based on the Jaccard distance between the potentially vulnerable path and the paths covered by system calls in the trace slices. Formally, let p be a potentially vulnerable path, and Q be a set of paths which are covered by the clusters; the path similarity between p and Q is defined as $S(p, Q) = mean_{q \in Q}(J(B(p), B(q)))$, where $J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$ and $B(p)$ is the set of branches covered by p. The trace slices which are sampled from selected clusters are finally

Table 1: The KACV detection precision of ACHyb and Pex. # perm refers to the number of detected permission checks. # priv refers to the number of detected privileged functions. # pvp refers to the number of detected potentially vulnerable paths. # wrn refers to the number of warnings. # kcav refers to the number of detected KACVs.

AC Module	ACHyb						PeX					
	# perm	# priv	# pvp	# wrn	# kacv	precision	# perm	# priv	# pvp	# wrn	# kacv	precision
CAP	28	560	108	38	9	23.7%	19	3,245	850	850	4	0.5%
LSM	243	2,254	90	31	9	29.0%	243	10,260	1,017	1,017	7	0.7%
DAC	27	609	29	7	4	57.1%	22	537	221	221	3	1.4%
Total	298	3,423	227	76	22	28.9%	284	14,042	2,088	2,088	14	0.7%

converted to seed programs. Suppose that the trace slice t_2 in Fig. 8 has been sampled from a selected cluster. The seed program converted from t_2 consists of two system calls from t_2, which is $n_1(a_1); n_2(a_2)$.

3.4 Implementation

We implement the static analysis of ACHyb based on the LLVM pass framework [45] with about 3,200 lines of C++ code. For the dynamic analysis, we build the greybox fuzzing and the seed distillation on top of the kernel greybox fuzzer called Syzkaller [56] with about 1,100 lines of GO code and 600 lines of Python code.

4 EVALUATION

4.1 Experimental Setup

Baseline. To evaluate the KACV detection performance of ACHyb, we choose PeX [62], the state-of-the-art tool for KACV detection which has its publicly available implementation[1], as our baseline. Besides, to evaluate our seed distillation approach, we choose Moonshine [42] which is the state-of-the-art seed distillation tool, as our baseline.

Kernel Version and Compilation. We evaluate ACHyb on the Linux kernel v4.18.5, the version PeX uses in its evaluation. We compile the kernel source with the `allyesconfig` configuration using the `clang-9` toolchain [55] and the `wllvm` tool [46].

Subjects. We take all the three subject modules in the PeX evaluation as our subjects. They are commonly used kernel access control modules, including Linux Capabilities (CAP), Linux Security Modules (LSM) and Discretionary Access Control (DAC).

Environment. All the experiments are conducted on a machine with two Intel(R) Xeon(R) E5-2620 v4 processors (32 logical cores in total) and 256-GB RAM. The operating system is Ubuntu 20.10.

4.2 Research Questions

We try to answer the three following research questions in our experiments:

Question 1. How precisely can ACHyb detect KACVs?
Question 2. How efficiently can ACHyb detect KACVs?
Question 3. Can ACHyb detect new KACVs?

4.3 RQ1: Detection Precision

To evaluate the detection precision, we need to identify the KACVs from the warnings (i.e., potential KACVs) reported by ACHyb and PeX. To do so, we conduct a three-step manual inspection. First, we manually inspect all the warnings reported by both tools, among which we obtain 15 KACVs that have been confirmed by the kernel developers or reported by the authors of PeX. Then, we manually inspect the rest warnings reported by ACHyb and identify 7 warnings as new KACVs. Next, we report these 7 new KACVs to the kernel developers. By the time of the paper publication, they have confirmed 2 new KACVs.

4.3.1 The Overall Detection Precision. Table 1 shows the detection precision of ACHyb and PeX. For CAP module, the detection precision of ACHyb is 23.7%, while the precision of PeX is only 0.5%. For LSM module, the detection precision of ACHyb is 29.0%, while the precision of PeX is only 0.7%. For DAC module, the detection precision of ACHyb is 57.1%, while the precision of PeX is only 1.4%. We can also observe that, among the three modules, both tools perform with the highest precision on module DAC; perform with the second highest precision on module LSM; perform with the lowest precision on module CAP. In total, for all the three modules, 28.9% of the warnings reported by ACHyb are KACVs, while only 0.7% of the warnings reported by PeX are KACVs. Furthermore, the KACVs detected by ACHyb contain all the ones detected by PeX. *Overall, we can say that* ACHyb *is much more precise than* PeX *in KACV detection.*

4.3.2 Static and Dynamic Analysis. Table 1 also shows the intermediate analysis results, which can help us understand how each analysis in ACHyb contributes to its precise detection.

Permission Check Identification. As for the permission check identification, ACHyb reports 14 more permission checks than PeX. We manually inspect these permission checks and confirm that all of them are the real permission checks. *The results show that the soundy interface analysis of* ACHyb *can help identify more permission checks than* PeX.

Privileged Function Identification. From Table 1, we can observe that ACHyb reports much less privileged functions than PeX. To study the quality of the reported privileged functions, we randomly sample 400 functions reported by ACHyb and PeX, respectively. After manually inspecting the sampled privileged functions, we found that 83% of the sampled privileged functions identified by ACHyb are the real privileged functions, while only 8% of the sampled privileged functions identified by PeX are the real privileged functions. In addition, the real privileged functions detected by ACHyb contain all the real ones detected by PeX. *In general, we can conclude that our proposed callsite dependency analysis improves the precision of the privileged function identification.*

[1] https://github.com/lzto/pex

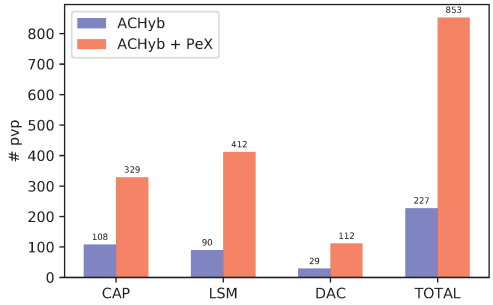


Figure 9: The number of potentially vulnerable paths detected by the invariant analysis. "ACHyb+PeX" refers to the results of PeX invariant analysis with the permission checks and privileged functions detected by ACHyb. # pvp refers to the number of detected potentially vulnerable paths.

Table 2: The time cost (in minutes) of static analysis. t_{perm} refers to the time cost of permission check identification. t_{priv} refers to the time cost of privileged function identification. t_{inv} refers to the time cost of invariant analysis. $t_{overall}$ refers to the total time cost of the entire static analysis.

AC Module	t_{perm}		t_{priv}		t_{inv}		$t_{overall}$	
	ACHyb	PeX	ACHyb	PeX	ACHyb	PeX	ACHyb	PeX
CAP	0.1	0.3	8.3	8.6	37.3	272.2	45.7	281.1
LSM	0.1	0.3	7.9	8.8	34.2	254.9	42.2	264.0
DAC	0.1	0.2	8.0	8.5	10.5	119.3	18.6	128.0
Total	0.3	0.8	24.2	25.9	82.0	646.4	106.5	673.1

Invariant Analysis. The invariant analysis is based on the detected permission checks and privileged functions. To conduct an apple-to-apple comparison on the invariant analysis, we run PeX invariant analysis on the permission checks and privileged functions detected by ACHyb. Fig. 9 shows the number of potentially vulnerable paths detected by the invariant analysis of both tools. PeX reports 853 potentially vulnerable paths, while ACHyb reports only 227 potentially vulnerable paths. After manually checking all the 853 paths reported by PeX under the new configuration, we found that the number of the real vulnerable paths reported by PeX is 19, while the number of real paths reported by ACHyb is 22 (as shown in Table 1). Therefore, the precision of ACHyb invariant analysis is 9.7%, while the precision of PeX invariant analysis is 2.2%. Besides, the real vulnerable paths reported by ACHyb contain all the ones reported by PeX. *The results show that the invariant analysis of* ACHyb *is more precise and sound than the invariant analysis of* PeX.
Dynamic Analysis. Finally, we notice that 33.4% of the potentially vulnerable paths are reported as warnings by our dynamic analysis. We manually check the 151 unreported paths and found that all of them are false positives. In contrast, PeX directly reports all the 2,088 potentially vulnerable paths as warnings which causes the extremely high false-positive rates. *The results show that our dynamic analysis helps to reduce the false positives in KACV detection.*

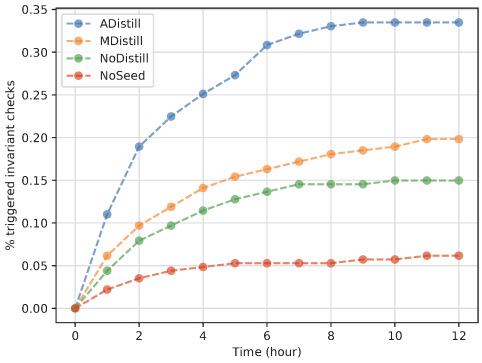


Figure 10: Time efficiency in triggering the invariant checks using four seed distillation approaches.

4.4 RQ2: Detection Efficiency

4.4.1 Static Analysis. Table 2 shows the time cost by the static analysis of ACHyb and PeX. The results show that, in total, ACHyb is 6.3x faster than PeX. The time differences are subtle in both the permission check detection and the privileged function detection. However, the big time differences lie in the invariant analysis where ACHyb is 7.9x faster than PeX. In addition to the time costs of the automated analysis, PeX requires users to provide an initial list of permission checks, while ACHyb requires manual effort to inspect the representative permission check candidates. For all the three subject modules, the user-provided list in PeX evaluation contains 196 permission checks, while the candidate list produced by ACHyb only contains 17 permission checks. To measure the required manual effort in the candidate inspection, each student author in this paper inspects the 17 candidates independently. As a result, each person spends 6 minutes on average (4 minutes in minimum and 8 minutes in maximum) on this inspection. *Overall, the results indicate that the static analysis of* ACHyb *is more efficient than* PeX.

4.4.2 Dynamic Analysis. To evaluate the efficiency of the dynamic analysis, we focus on evaluating our proposed seed distillation approach. We choose the state-of-the-art seed distillation tool called Moonshine [42] as our baseline, and construct four groups of the seed programs: 1) no seed programs (denoted as NoSeed); 2) the seed programs which are converted from the execution traces without any distillation (denoted as NoDistill); 3) the seed programs which are distilled from the execution traces using Moonshine (denoted as MDistill); 4) the seed programs which are distilled from the execution traces using ACHyb (denoted as ADistill). We evaluate the time efficiency and the number of triggered invariant checks of ACHyb using each seed group. To get the reliable experimental results, we follow the fashion of the recent research work on fuzzing evaluation [18, 27]. For each group of the seed programs, we run ACHyb with 16 logical cores in 12 hours and repeat the experiment 11 times. We perform the Mann-Whitney U test [35] to analyze the significance of the performance differences.

Fig. 10 shows the median percentage of the triggered invariant checks. We can observe that only about 5% of the invariant checks can be triggered when no seed program is applied. While the seed programs without distillation help to trigger more invariant checks,

Table 3: New KACVs Detected by ACHyb.

ID	File Path	Function	Type	Description	Status
CAP-1	net/core/rtnetlink.c	do_setlink	KACV-I	Misusing the CAP_NET_ADMIN check.	Confirmed
CAP-2	drivers/char/random.c	_extract_crng	KACV-M	Missing the CAP_SYS_ADMIN check.	Confirmed
CAP-3	net/ipv6/addrconf.c	addrconf_join_anycast	KACV-M	Missing the CAP_NET_ADMIN check.	Ignored
CAP-4	drivers/tty/sysrq.c	sysrq_do_reset	KACV-M	Missing the CAP_SYS_BOOT check.	Ignored
LSM-1	kernel/signal.c	send_sig_info,force_sigsegv	KACV-M	Missing the security_task_kill check.	Ignored
LSM-2	ipc/sem.c	newary	KACV-I	Misusing the security_sem_alloc check.	Ignored
DAC-1	fs/coredump.c	cn_print_exe_file	KACV-M	Missing the inode_permission check.	Ignored

only 15% of the invariant checks can be triggered. Furthermore, there is less than 5% improvement when using the seed programs distilled by Moonshine, compared to using the seed programs without any distillation. When using the seed programs distilled by ACHyb, more than 30% of the invariant checks can be triggered. Furthermore, the results of the Mann-Whitney U test indicate that the triggered checks using ACHyb seed distillation approach are significantly more than the triggered checks using the other three approaches (all three p-values of Mann-Whitney U test are smaller than 0.001). *Based on the above results, we can conclude that* ACHyb *seed distillation approach can significantly improve the efficiency of triggering the invariant checks.*

4.4.3 The Overall Detection Efficiency. As introduced, PeX requires human effort to remove the false positives of the results produced by its static analysis, while ACHyb applies dynamic analysis to remove the false positives. After manually inspecting the potentially vulnerable paths, we find that ACHyb has successfully triggered all the invariant checks associated with 22 KACVs in the first 6 hours. In addition to the 112.5 minute time cost in the static analysis (106.5 minutes for automatic analysis and 6 minutes for manual inspection of permission check candidates), ACHyb successfully detects all 22 KACVs in less than 8 hours. On the contrary, PeX spends more than 11 hours on only the static analysis phase without taking into account the time taken by the manual false positive removal. *The results show that* ACHyb *is more efficient than* PeX.

4.5 RQ3: New KACVs

Table 3 shows the 7 new KACVs (5 KACV-M and 2 KACV-I) that we report to the kernel developers. As a result, 2 new KACVs (1 KACV-M and 1 KACV-I) are confirmed by the kernel developers. We are still waiting for the feedback for the rest of 5 new KACVs. In detail, 4 KACVs (CAP-1, CAP-2, CAP-3, and CAP-4) are due to missing or misusing CAP checks in drivers or the net subsystem; 2 KACVs (LSM-1 and LSM-2) are due to missing or misusing LSM checks in the signal mechanism or the semaphore mechanism; 1 KACV (DAC-1) is due to missing a DAC check in the file system. *Overall, we can say that* ACHyb *is able to detect new KACVs.*

5 DISCUSSION

In this section, we want to discuss the limitations of ACHyb and our future work. We recognized four limitations of ACHyb. First, ACHyb cannot detect *non-function* permission checks. Second, ACHyb cannot detect privileged functions which are never protected by any identified permission checks. However, this is a rare case, as it is quite unlikely that kernel developers failed to add any permission checks to protect a privileged function especially in recent kernel versions. Third, ACHyb may get false positives in terms of privileged function detection, as there may exist non-privileged functions which are also protected by permission checks. Fourth, ACHyb cannot guarantee that all potentially vulnerable paths can be covered in a given time budget, which may cause false negatives. Nevertheless, the coverage can be improved by adding more diverse seed programs. In future work, we will try to enhance ACHyb to overcome the above limitations. We also plan to upgrade ACHyb to support more access control modules in the Linux kernel. In addition, we plan to propose effective approaches to detect KACV-S. One possible direction would be to specify the correctness of the internal access control states and validate these states using the specifications dynamically.

6 RELATED WORK

Missing Check Detection. Detecting missing checks is pioneered by Engler's work [12], which attempts to automatically extract the programmers' beliefs from the source code to detect missing checks. Following this direction, several static analysis tools have been proposed to detect missing checks. AutoISES [54] automatically infers the security specification given a set of user-provided security checks and detects the security violations in the Linux kernel. ROLECAST [52] leverages the standard software engineering patterns/conventions to detect the missing security checks in the Web applications. CRIX [31] proposes a novel peer slicing approach to detect missing checks for the critical variables in the Linux kernel. LRSan [60] proposes its specialized data-flow and control-flow analysis to detect missing rechecks for critical variables. PeX [62] is the state-of-the-art tool to detect the missing access control permission checks (KACV-M), which is the most related work to ACHyb. ACHyb differs from PeX in two main aspects. First, ACHyb focuses on detecting both KACV-M and KACV-I. Second, ACHyb performs a novel hybrid analysis to make the KACV detection both scalable and precise, while PeX is a purely static analysis tool suffering from high false-positive rates.

Greybox Fuzzing of OS Kernel. Greybox fuzzing achieves big success in revealing real-world vulnerabilities in recent decades [33, 59]. Several greybox fuzzers for OS kernel including Syzkaller [57], TriforceAFL [19] and Trinity [22] have been released. Besides the tool developments, researchers make great effort in enhancing both the effectiveness and the efficiency of the kernel fuzzing. Breakthroughs have been made in the complex path condition solving [4, 24, 26], seed generation [17], seed distillation [42], file system testing [25, 61], driver/firmware testing [10, 36, 44, 53, 63],

error handling testing [43], etc. Different from the existing greybox fuzzers, ACHyb proposes a novel clustering-based seed distillation approach to facilitate the greybox fuzzing in KACV detection.

Kernel Verification and Validation. Several approaches [3, 13, 16, 38, 39, 50] have been proposed to verify or validate the correctness of the kernel source code. For example, Serval [38] is a framework for verifying system software. Given an interpreter provided by users, it performs symbolic execution on the system code to do the verification. Besides, TESLA [3] provides users a language to specify the dynamic safety properties of the Linux kernel. The specified properties are then converted into run-time checks to validate the kernel. Different from the above approaches which require the users to provide the specification of the Linux kernel, ACHyb is able to detect KACVs based on the invariants of the access control.

7 CONCLUSION

In this paper, we first conduct an empirical study on KACVs using National Vulnerability Database. Motivated by our study, we focus on detecting two kinds of KACVs: KACV-M and KACV-I. We present a precise and scalable hybrid analysis approach called ACHyb to detect both KACV-M and KACV-I. ACHyb first performs a more precise and more sound static analysis to identify the potentially vulnerable paths, and then applies an efficient dynamic analysis to reduce the false positives of these paths. Our experimental results show that ACHyb outperforms PeX, the state-of-the-art KACV detector, in terms of both the detection precision and the efficiency. Furthermore, ACHyb detects 7 new KACVs, 2 of which have been confirmed by the kernel developers.

ACKNOWLEDGMENTS

We would like to thank the anonymous FSE'21 reviewers, S&P'21 reviewers and kernel developers for their valuable feedback. This work was supported by Intel Strategic Research Alliance (ISRA) grant, SRC grant TS-2965, NSF grants 26101114, 26101313, and CCF-1718903, and a grant from the Army Research Office accomplished under Cooperative Agreement Number W911NF-19-2-0333. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[3] Jonathan Anderson, Robert NM Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. 2014. TESLA: temporally enhanced system logic assertions. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *The Network and Distributed System Security Symposium (NDSS)*, Vol. 19. 1–15.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[6] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. 2933–2942.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[8] Matthieu Brucher. 2020. scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/.

[9] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[11] Dulaunoy, Moreels Alexandre, Vinot Pieter-Jan, and Raphael. 2020. CVE-SEARCH PROJECT. https://www.cve-search.org/.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.

[13] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 287–305.

[14] Google. 2020. System and kernel security. https://source.android.com/security/overview/kernel-security.

[15] Andreas Grünbacher. 2003. POSIX Access Control Lists on Linux.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 259272.

[16] Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. 2019. Integrating Formal Schedulability Analysis into a Verified OS Kernel. In *International Conference on Computer Aided Verification*. Springer, 496–514.

[17] HyungSeok Han and Sang Kil Cha. 2017. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2345–2358.

[18] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[19] Jesse Hertz. 2016. A linux system call fuzzer using TriforceAFL. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[20] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. 1989. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. 28–40.

[21] Joxan Jaffar, Vijayaraghavan Murali, Jorge A Navas, and Andrew E Santosa. 2012. Path-sensitive backward slicing. In *International Static Analysis Symposium*. Springer, 231–247.

[22] Dave Jones. 2011. Trinity: Linux system call fuzze. https://github.com/kernelslacker/trinity.

[23] Michael Kerrisk. 2019. overview of linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html.

[24] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *The Network and Distributed System Security Symposium (NDSS)*.

[25] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 147–161.

[26] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. Cab-fuzz: Practical concolic testing techniques for {COTS} operating systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 689–701.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[28] Bogdan Korel and Juergen Rilling. 1998. Program slicing in understanding of large programs. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE, 145–152.

[29] Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-biggraph: A large-scale graph embedding system. *arXiv preprint arXiv:1903.12287* (2019).

[30] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. 2003. The global k-means clustering algorithm. *Pattern recognition* 36, 2 (2003), 451–461.

[31] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1769–1786.

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

[33] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science,

and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).
[34] Dror E Maydan, John L Hennessy, and Monica S Lam. 1991. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1–14.
[35] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.
[36] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. 2021. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[37] Steven Muchnick et al. 1997. *Advanced compiler design implementation*. Morgan kaufmann.
[38] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 225–242.
[39] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 252–269.
[40] NIST. 2020. National Vulnerability Database. https://nvd.nist.gov/.
[41] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2289–2306.
[42] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing {OS} Fuzzer Seed Selection with Trace Distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 729–743.
[43] Aditya Pakki and Kangjie Lu. 2020. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.
[44] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. (2020).
[45] LLVM Project. 2020. Writing an LLVM Pass. https://llvm.org/docs/WritingAnLLVMPass.html.
[46] Tristan Ravitch. 2020. Whole Program LLVM. https://github.com/travitch/whole-program-llvm.
[47] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. *ACM SIGPLAN Notices* 51, 1 (2016), 761–774.
[48] Ravi S Sandhu and Pierangela Samarati. 1994. Access control: principle and practice. *IEEE communications magazine* 32, 9 (1994), 40–48.
[49] SGI, OSDL, and Bull. 2012. Linux Test Project. https://linux-test-project.github.io.
[50] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A framework for design and verification of information flow control systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 287–305.
[51] Stephen Smalley, Timothy Fraser, and Chris Vance. 2020. Linux Security Modules: General Security Hooks for Linux. https://www.kernel.org/doc/html/latest/security/lsm.html.
[52] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1069–1084.
[53] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary.. In *The Network and Distributed System Security Symposium (NDSS)*.
[54] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.
[55] The Clang Team. 2019. Clang 9 documentation. https://releases.llvm.org/9.0.0/tools/clang/docs/ReleaseNotes.html.
[56] Dmitry Vyukov. 2015. syzbot. https://syzkaller.appspot.com/upstream.
[57] Dmitry Vyukov. 2015. Syzkaller. https://github.com/google/syzkaller.
[58] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 481–495.
[59] Pengfei Wang and Xu Zhou. 2020. SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. *arXiv preprint arXiv:2005.11907* (2020).
[60] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.
[61] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.
[62] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1205–1220.
[63] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1099–1114.