LiCA: A Fine-grained and Path-sensitive Linux Capability Analysis Framework

Menghan Sun
The Chinese University of Hong Kong
Hong Kong, China
sm017@ie.cuhk.edu.hk

Zirui Song
The Chinese University of Hong Kong
Hong Kong, China
sz019@ie.cuhk.edu.hk

Xiaoxi Ren
Hunan University
Hunan, China
happyrxx@hnu.edu.cn

Daoyuan Wu
The Chinese University of Hong Kong
Hong Kong, China
dywu@ie.cuhk.edu.hk

Kehuan Zhang
The Chinese University of Hong Kong
Hong Kong, China
khzhang@ie.cuhk.edu.hk

ABSTRACT

The capability mechanism in Linux-based systems is designed for dispersing the root privileges into a set of more refined capabilities, making programs gain no-more-necessary privileges. However, it is challenging to check the necessity and sufficiency of capabilities assigned to programs due to the highly complicated call chains invoked in practice. Inappropriate capability assignment brings threats to the systems. For example, over-privileged programs could allow an attacker to misuse root privileges, while under-privileged programs may incur runtime errors.

In this paper, we propose a new Linux capability analysis framework called LiCA to find necessary and sufficient capabilities for programs effectively. LiCA presents fine-grained and path-sensitive code flow analysis based on LLVM to construct accurate mappings between system calls and their capabilities. In particular, we solve the constraint equations along each path from a given system call to individual capabilities and strategically overcome the path explosion problem. Our experiments show that LiCA can correctly find necessary capabilities for the Linux utility programs (e.g., ping and tcpdump) and the public programs from GitHub. By comparing the capabilities claimed by program developers and the results from LiCA, we identify a batch of programs requiring more capabilities than necessary, even root privileges. Therefore, LiCA could help those third-party developers validate their programs' capability setting to achieve the least privilege principle.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Linux capability, security analysis, mapping

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545966

ACM Reference Format:
Menghan Sun, Zirui Song, Xiaoxi Ren, Daoyuan Wu, and Kehuan Zhang. 2022. LiCA: A Fine-grained and Path-sensitive Linux Capability Analysis Framework. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3545948.3545966

1 INTRODUCTION

Privilege management is crucial for the security of Linux systems. It controls the access to sensitive resources of the system. Since the birth of Linux, many access control models have been proposed, e.g., SELinux [36] and AppArmor [33]. The capability mechanism is the fundamental one [19] that has been widely used in various Linux-based systems. It disperses the root privilege into a set of more delicate capabilities. Correctly assigning capabilities can avoid letting unnecessary programs obtain the root privilege and then manipulate the system arbitrarily.

If some capabilities have been assigned while the program does not need them, it may offset the benefits of the capability mechanism over the root mechanism. On the other hand, the insufficient privileges will result in exceptions and breaks of the normal process, which are unexpected for the security of the Linux kernel. In practice, the over-privileged cases are far more common than insufficient privilege cases because the over-assigned privileges will not harm the program's smooth execution and don't need the efforts of developers to check carefully. Users gradually acquiesced to these over-privileged programs due to the absence of vision about the long-term harms to the system. However, over-privileged cases go against the original intention of the capability mechanism design since adversaries may exploit over-assigned privileges to compromise the system.

Currently, Linux kernel uses a static and coarse-grained mechanism for capability checking, which leaves security loopholes, e.g., injected malicious code in a victim program may acquire all capabilities this program has and accomplish privilege escalation ([2, 7, 14, 15]). Existing works ([24, 29]) related to analyzing capabilities and system calls suffer from either high false positives or high false negatives. TCLP [29] uses dynamic analysis to monitor which capabilities are checked when system calls are invoked, but the mapping between system calls and capabilities provided is coarse-grained. If a program calls a system call that can trigger multiple capabilities, TCLP will report all the capabilities regardless

of whether these capabilities are needed. Moreover, the dynamic analysis may not cover all execution paths, which may result in false negatives (i.e., insufficient privilege). AutoPriv [24] uses whole program analysis during link-time optimization to determine where programs use system calls and remove those related capabilities when system calls return. However, in practical scenarios, one system call may be related to multiple capabilities, but only part of them will be triggered in one execution. This coarse assignment results in false positives (over-privileged). These problems motivate us to invent new techniques that could generate a fine-grained and path-sensitive mapping between system calls and Linux capabilities.

To correctly assign capabilities to programs, we propose LiCA, a capability analysis framework for Linux kernel to generate path-sensitive mappings between system calls and Linux capabilities, and effectively analyze the program. LiCA makes it possible to find the capabilities triggered by a given program precisely. At a high level, LiCA statically analyzes the Linux kernel and constructs a fine-grained and path-sensitive mapping between the system call and Linux capability offline. The mappings are generated by analyzing the invoked call chain and combining their branch conditions into mathematical constraint equations. When analyzing the program, LiCA leverages the mapping to predict the necessary capabilities. We implemented a full-featured prototype of LiCA and evaluated it on 114 real-world programs. Among these programs, we discovered three cases requiring more capabilities than they need, three cases asking users to assign root privilege, one inducing the user to assign root privilege without providing a capability list, and one program having different capability settings in different applications.

Contributions. Our paper makes three key contributions.

- *New framework.* We have proposed a new framework, to generate a fine-grained and path-sensitive mapping between system calls and Linux capabilities. Not only it can be used to discover the wrongly assigned capabilities in existing programs, but it can also help developers to assign the right capabilities to their work under the least privilege principle.
- *Prototype Implementation.* We have implemented a full-featured prototype LiCA. To handle the complex situations inside Linux kernel source code, we leveraged and customized some techniques, including backward program slicing, branch condition pruning, code modeling, and constraint equation converting.
- *Findings and evaluations on real-world programs.* We have evaluated LiCA prototype 114 real-world programs. Discrepancies are observed, and manual analysis shows that the differences are caused by various capability setting problems, including requiring root privilege (sudo), setting more capabilities than needed, no capability list provided but "no permission error" when running, and different capabilities being assigned to the same program in different applications.

Roadmap. The rest of this paper is organized as follows: Section 2 explains the background of our work; Section 3 describes the fine-grained and path-sensitive analysis with illustrating some motivation examples; Section 4 introduces our detailed design of LiCA; Section 5 evaluates LiCA; Section 6 discusses the limitations of LiCA; Section 7 reviews the state-of-the-art works related to our work; Section 8 concludes the paper.

2 BACKGROUND

This section presents the necessary background of LiCA. It begins with the Linux capability, which is throughout our task, and then briefly describes the program analysis techniques applied in LiCA.

2.1 Privilege Management Primitives

Linux Capabilities. Starting with Linux kernel 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled [19]. Capabilities provide fine-grained control to superuser permissions and are designed for avoiding misuse of root privileges. For example, `CAP_NET_RAW` enables privileges to capture the network flow but doesn't enable other network relative privileges like binding port. Likewise, the `CAP_SETUID` allows a process to change the user id of one file but doesn't allow a process to change the group id. But in practice, the choice of capabilities required by a program has been made based on the developers^{TM} understanding, which often causes some problems. The so-called "new root" problem is a prominent example, mis-assigning `CAP_SYS_ADMIN` to an untrusted user will generate a powerful adversary with nearly half of the privileges of a root user. In the recent version of Linux, `CAP_SYS_ADMIN` even accounts for more than 45% of all the capabilities [26]. Although Linux has many security mechanisms, the interdependence and mutual influence relationship among all kernel security mechanisms may make them fall into the "shortboard effect" and impair their protection capability [30]. Hence, detecting and eliminating vulnerabilities in the Linux kernel is a key for securing operating systems.

Linux Privilege Escalation. An over-privileged program can obtain privileges more than needed, leaving the space for the adversary to launch a privilege escalation attack by hijacking this program. For example, Tar is a normal user runnable program. Users can use it on previously created archives to extract files, store additional files, update and list files. In some Linux variations, Tar has `CAP_DAC_READ_SEARCH` capability [2] and can access anything. Adversaries could get password hashes by using Tar to read /etc/shadow, the root only readable file. Specifically, since tar has that capability, the adversary can archive /etc/shadow and extract it from the archive then read it. Another example is `CAP_DAC_OVERRIDE`, which allows full read/write/execute access. When obtaining this capability, adversaries can add a root user to /etc/passwd or /etc/shadow, modify cron jobs running by root, add a public ssh key to /root/authorized_keys, or simply open a root shell. Moreover, if `CAP_SETUID` capability is assigned to python, adversary can use "import os; os.setuid(0); os.system('/bin/sh')" to get the root shell.

2.2 Program Analysis Techniques

Symbolic Execution. Symbolic execution [27] is a widely used technology that analyze the program to get the set of inputs that make a specific code area execute smoothly. When applying symbolic execution to analyze a program, inputs of the program will be replaced with the symbolic formula rather than specific values. In LiCA, we apply the symbolic execution to organize the extracted branch conditions and generate the constraint equations for a path.

Z3 Solver. Z3 [40] is a state-of-the-art SMT (Satisfiability modulo theories) solver from Microsoft Research. It is designed for solving the logical formulas during program analysis likes checking the satisfiability of logical formulas over one or more theories [21]. In LiCA, we use Z3 to find solutions to satisfy all the constraints in paths from the system call to the capabilities. According to these solutions, we can find valid system calls' arguments that can reach specific capabilities we want.

3 EXAMPLES OF PATH-SENSITIVE CAPABILITY TRIGGERING

Previous works only perform a coarse-grained analysis to generate mappings between system calls and capabilities. For example, suppose capability C is found in a function that is invoked by system call S, previous works will regard capability C as required by system call S, regardless of the entrance arguments of system call S. However, cases in the Linux kernels are more complex. We have obtained the following observations besides the above direct mapping:

Observation 1: A function can trigger multiple capabilities under different conditions, depending on the arguments of the function; sometimes no capability is required.

Observation 2: Sometimes a function can work without a capability it checks - the capability is not compulsory.

Observation 3: Mapping between the system calls and capabilities is many-to-many mapping.

We will present some real-world examples from the Linux kernel to support the above observations and justify our motivation for a path-sensitive mapping between system calls and Linux capabilities.

3.1 Different capabilities can be triggered by a system call under different conditions

Fig. 1 shows system call `setsockopt` can trigger `CAP_NET_ADMIN`, `CAP_NET_RAW` or no capability with different settings of argument `optname`. (**Observation 1**). The path from system call `setsockopt` to `CAP_NET_ADMIN` and `CAP_NET_RAW` is shown below:

```
SYSCALL_DEFINE5(setsockopt) -> __sys_setsockopt ->
sock_setsockopt -> ns_capable(CAP_NET_ADMIN/RAW)
```

As shown in Fig. 1a, function `setsockopt` is defined as a system call (line 1) in Linux kernel and it calls `__sys_setsockopt` directly (line 3). In function `__sys_setsockopt` (Fig. 1b), function `sock_setsockopt` is called (line 9) only if (1) `optlen >= 0` (line 4); (2) the return sock of function `sockfd_lookup_light` is not NULL (line 6); (3) `level` equals to `SOL_SOCKET`. If all these three conditions are fulfilled, this function will finally set the option for the socket. Although `CAP_NET_ADMIN` appeared in this function many times (Fig. 1c shows part of the appearances of `CAP_NET_ADMIN`), we can notice that `CAP_NET_ADMIN` is required only when `optname` equals to `SO_SNDBUFFORCE` (line 10) or `SO_PRIORITY` (line 15) in this code snippet. Meanwhile, when `optname` equals to `SO_PRIORITY` and `0 <= val <= 6`, `CAP_NET_ADMIN` will not be used and there will be no capability required with no error as well. However, in traditional capability evaluation methods, `CAP_NET_ADMIN` will be regarded as compulsory for the program. When `optname` equals to `SO_BINDTOIFINDEX`, function `sock_setbindtodevice_locked` will be called (line 21). As shown in Fig. 1d, the function will check

```
1  SYSCALL_DEFINE5(setsockopt, int fd, int level, int
       optname, char __user * optval, int optlen)
2  {
3    return __sys_setsockopt(fd, level, optname, optval,
       optlen);
4  }
```

(a) System call `setsockopt` first calls `__sys_setsockopt` (line 3).

```
1  static int __sys_setsockopt(int fd, int level, int
       optname, char __user *optval, int optlen)
2  {
3    ...
4    if (optlen < 0)   return -EINVAL;
5    sock = sockfd_lookup_light(fd, &err,&fput_needed);
6    if (sock != NULL) {
7      ...
8      if (level == SOL_SOCKET)
9        err = sock_setsockopt(sock, level, optname, optval,
         optlen);
10     ...
11   }
```

(b) `__sys_setsockopt` calls `sock_setsockopt` (line 9) if (1) `optlen >= 0` (line 4); (2) the return sock of function `sockfd_lookup_light` is not NULL (line 6); (3) `level` equals to `SOL_SOCKET`.

```
1  int sock_setsockopt(struct socket *sock, int level, int
       optname, char __user *optval, unsigned int optlen)
2  {
3      ...
4    switch (optname) {
5    ...
6    case SO_BROADCAST: /* Trigger no capability*/
7      sock_valbool_flag(sk, SOCK_BROADCAST, valbool);
8      break;
9    case SO_SNDBUFFORCE: /* Trigger CAP_NET_ADMIN */
10     if (!capable(CAP_NET_ADMIN)) {
11       ret = -EPERM;
12       break;
13     }
14   case SO_PRIORITY: /* Trigger CAP_NET_ADMIN or no
       capability*/
15     if ((val >= 0 && val <= 6) || ns_capable(sock_net(sk)
       ->user_ns, CAP_NET_ADMIN))
16       sk->sk_priority = val;
17     else
18       ret = -EPERM;
19     break;
20   case SO_BINDTOIFINDEX: /* Trigger CAP_NET_RAW */
21     ret = sock_setbindtodevice_locked(sk, val);
22     break;
23   ...
24 }
```

(c) With different settings of argument `optname`, `CAP_SYS_ADMIN` or no capability will be triggered.

```
1  static int sock_setbindtodevice_locked(struct sock *sk,
       int ifindex)
2  {
3    ...
4    ret = -EPERM;
5    if (!ns_capable(net->user_ns, CAP_NET_RAW))
6      goto out;
7      ...
8  out:
9      return ret;
10 }
```

(d) `sock_setbindtodevice_locked` will be called when `optname` equals to `SO_BINDTOIFINDEX`, then it checks if there is `CAP_NET_RAW`.

Figure 1: An example of different capabilities can be triggered by a system call under different conditions

the capability. if there is no CAP_NET_RAW capability, this function will return -EPERM (Operation not permitted) error.

3.2 Function still works without the Linux capability it checks

Here is another example from system call open to CAP_SYS_ADMIN:

```
SYSCALL_DEFINE3(open) -> do_sys_open ->
do_filp_open -> path_openat ->
alloc_empty_file -> capable(CAP_SYS_ADMIN)
```

As shown in Fig. 2, function alloc_empty_file checks if the program has capability CAP_SYS_ADMIN (line 4). However, this check only happens when the result of function get_nr_files (this function will return the total number of open files in the system) is large than the maximum value of files set in the system. It means that privileged users can go above max_files. Without CAP_SYS_ADMIN, system call open still works when the number of open files is less than the maximum (**Observation 2**).

3.3 Mapping between the system calls and Linux capabilities

The above examples show the traditional way cannot accurately analyze which capabilities a program requires. There is no tool for developers and users to figure out which capabilities should be assigned to a program precisely. It's essential to get a fine-grained system call and capability mapping to avoid being over-privileged. An example of the mapping is shown in Fig. 3, which is represented as a call graph. The call chain indicates the system call goes through some functions and then finally reaches the Linux capability (i.e., the capability is supposed to be checked). The nodes between the system call and Linux capability represent different functions invoked within the call chain. The edges indicate the calling relationships (with some/no branch conditions) in the function call. If the system call entrance arguments cannot satisfy corresponding branch conditions, it means one cannot go through this call chain and reach the Linux capability.

Fig. 3 also shows that a system call is able to reach multiple Linux capabilities; meanwhile, different system calls are also able to trigger the same Linux capability. As a result, it is a many-to-many mapping between the system calls and Linux capabilities (**Observation 3**). With the same system call and different entrance arguments, the final result as to which Linux capability will be triggered will be different. As a user program can have a number of system calls, there can be different capabilities required by it. Hence the fine-grained and path-sensitive mapping relationship is necessary for developers and users to prevent programs over-privileged.

4 DESIGN

4.1 Overview

In this work, we propose a new approach to generate a fine-grained and path-sensitive mapping from system calls to Linux capabilities. The whole design overview is shown in Fig.4, where we have combined the technique parts (the top blocks) and the visualization (the bottom figures) of the result at each step. At a high level, LiCA has

```
1  struct file *alloc_empty_file(int flags, const struct
       cred *cred)
2  {
3    ...
4    if (get_nr_files() >= files_stat.max_files && !capable(
       CAP_SYS_ADMIN)) {
5      if (percpu_counter_sum_positive(&nr_files) >=
         files_stat.max_files)
6        goto over;
7    }
8    f = __alloc_file(flags, cred);
9    ...
10 }
```

Figure 2: Although this code snippets check CAP_SYS_ADMIN capability (line 4), this function still works successfully (line 8). With CAP_SYS_ADMIN, privileged users can go above max_files.

4 components: path extractor, branch condition analyzer, constraint equation generator, and user interface.

- **Path extractor.** The call chains are the backbones of the mapping from system calls to Linux capabilities, so the goal is to precisely extract the call chains from system calls to capabilities. Path extractor takes the LLVM intermediate representation (LLVM IR) [17] of the Linux kernel as the input and discovers all the functions invoked between the system calls and capabilities to build the paths. (Â§4.2)
- **Branch condition analyzer.** To generate a fine-grained and path-sensitive mapping from system calls to capabilities, we analyze the branch conditions within each function to extract and organize the branch conditions for each function through the paths. (Â§4.3)
- **Constraint equation generator.** After all the branch conditions are generated, we can derive constraint equations. The constraint equations are mathematical condition sets that indicate the requirements for entrance arguments to pass the branch condition and go through the path. (Â§4.4)
- **User interface.** With obtained the mapping from system calls to Linux capabilities, a user interface is needed to leverage the mapping and analyze the program in practical usage. (Â§4.5)

4.2 Function call paths extraction

How to extract the call chains from system calls to capabilities accurately and completely becomes the first challenge. Instead of analyzing the source code of the Linux kernel, we choose first to compile it into LLVM IR. There are two advantages: (1) The LLVM compiler translates the Linux kernel source code to a representation closer to machine instructions, which will be more accurate and accessible for analysis. This is because C/C++ languages use an archaic syntax and allow for nightmarish language constructs, which makes static analysis much complex. But IR will be simple as it eliminates the reliance on the concrete source language, involves no nesting and has fewer instructions [18, 35]. (2) We can run some optimization passes over the generated IR to reduce the works (such as the inline functions) in static analysis.

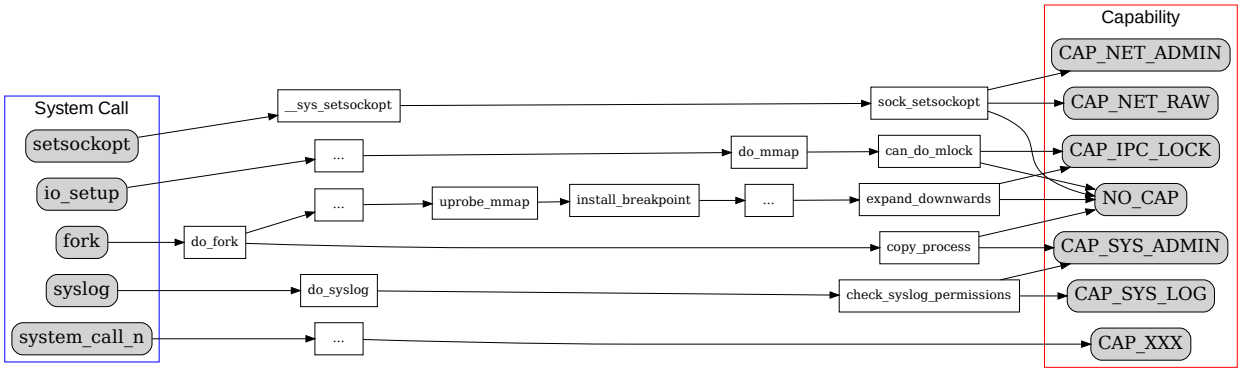


Figure 3: An Example of the Paths between System Call and Linux Capability

Backward program slicing. Prior studies ([28, 34]) have confirmed that enumerating all execution paths originated from the entry point of a system call may face the problem of path explosion. As a result, the analysis process can keep running for a long time. To address this problem, we propose a two-way analysis technique. As the capability macros have been compiled into its defined value (e.g., `CAP_NET_RAW` is compiled into 13), first we need to figure out where a given capability is used. After going through all the locations of capability macros in source code, we collected the functions which contains the capabilities as arguments (these functions will check if a capability is capable or not), such as *capable*, *file_ns_capable*, *ns_capable*, *inode_capable* and etc. Then we can precisely locate these functions in LLVM IR and figure out which capability is used according to the its defined value. We also go through all its basic blocks for each function and collect all the callees and generate a one-to-many call graph. For example, by going through the basic blocks in Fig. 5b, we can find that function `foo` will call function `p` by `"call void @p(i32)"` . With this start, we can do backward tracing along with the call graph from callee to the caller and so on until the definition of a system call is reached. During the backward tracing, we also perform program slicing by removing the functions that are irrelevant to the interested call trace.

Indirect call-based path extracting. In addition to direct calls, function pointers are also commonly utilized to facilitate dynamic program behaviour in the Linux kernel, which are known as indirect calls [31]. To deal with these kinds of calls, we followed the method in Pex [41] to find the indirect calls in the Linux kernel. We modify its source code to print out all the indirect calls and add them to the callee and caller pairs to generate the call chains with direct and indirect calls.

4.3 Branch conditions analysis

After the extraction of Linux function call chains, the next step is to construct branch conditions for each path. The branch conditions in the functions are mostly represented as "If-Else" statements, and they are mathematically represented as equality or inequality. For example, in Fig.3, the system call `setsockopt` will call the function`__sys_setsockopt` and `sock_setsockopt`. Depending on the entrance arguments, it may finally trigger the `CAP_SYS_RAW`, `CAP_SYS_ADMIN` or no capability.

The branch conditions are based on the function, so we first split the function call path generated in the last section into (caller, callee) pairs. For example, the path starting from the system call `setsockopt` to `CAP_NET_ADMIN` and `CAP_NET_RAW` in Section 3.1 can be divided into three pairs: (`setsockopt`, `__sys_setsockopt`), (`__sys_setsockopt`, `sock_setsockopt`) and (`sock_setsockopt`, `ns_capable`). To generate branch conditions for each function pair, we first need to analyze the Control Flow Graph (CFG).

Control flow graph analyzing. To build the CFG, firstly we iterate each basic block and analyze the terminator instruction. As each basic block has a list of successors - basic blocks to which control flow may transfer from these basic blocks. It is easy to obtain the control flow and the branch condition of it by looking at the terminator instruction of the basic blocks as the header `llvm/ADT/PostOrderIterator.h` offers iterators for going over basic blocks inside a function in the post-order traversal. This interlinking of basic blocks constitutes the CFG. However, while the basic block graph is directed, it's not necessarily cycle-free. Any loop in the code translates to a cycle in the basic block graph so that there can be errors. To deal with this problem, we mark each visited basic block as visited and would not go through it again.

The next step is to find the basic block location where callee's function is called in the caller's function and trace back the path how the caller reach the callee from the entry starting to this location. Fig. 5a shows a toy example. Function `foo` has 6 basic blocks: `entry, if.then, if.else, if.then2, if.end and if.end3`. Block `entry` is the entry point for `foo` and function `p` is called in block `if.end3`. For this example, there are 3 paths for `foo` to call function `p`: `(1) entry -> if.then -> if.end3; (2) entry -> if.else -> if.then2 -> if.end -> if.end3; (3) entry -> if.else -> if.end -> if.end3`. For each path, we record the branch conditions such as `entry (br i1 %cmp, True) -> if.then -> if.end3` for path `(1)`. As we only care about the entrance arguments - only %4, branch condition instructions, instructions related to %4 and branch condition instructions, we ignore

the irrelevant instructions. After applying use-define chain [25], the output for path (1) is shown below (in reversing order):

```
call void @p(i32 %4)
%4 = load i32, i32* %n.addr, align 4
True = br i1 %cmp
%cmp = icmp sle i32 %0, 2
%0 = load i32, i32* %b.addr, align 4
store i32 %b, i32* %b.addr, align 4
%b is the argument of function foo
```

However, two cases require special handling. The first case is that some functions use context information of the system, such as the total number of open files in the system in Fig. 2, or some information from current_task. As static analysis cannot give the value of these variables, we regard all the related instructions as True. The second one is that a system call might still work without the capability as shown in Fig. 2. For some user programs, it might be essential that the system call always works, so they need the capability, and for others, it can be fine that it fails sometimes. We assign this capability to the user program to make it always works. Fortunately, we found this case is rare in the Linux kernel and will not cause too much impact.

Branch condition pruning. When analyzing some complex functions, there are over hundreds of branch conditions. For instance, for function pair (`load_module, kobject_uevent`), there are 396 branch conditions before function `kobject_uevent` is called. Suppose one-tenth of them have 2 paths, and there will be $2^{40} = 1.0995116 * 10^{12}$ paths, which will lead to path explosion. To deal with this problem, we studied the functions with over 40 branch conditions in detail and found that (1) many of the branch conditions are the error checking; (2) many branch conditions will reach the same basic block no matter it is True or False. Fig.6 illustrate code snippets of the function (`load_module`, as shown in lines 5, 7, and 8, they will all form branch conditions to determine whether there is an error or not. If there is no capability checking in such a branch condition, we can just ignore this branch condition as this will not affect the final result.. The example in Fig. 5 shows that all the three paths will finally call function `p`, so that all the branch conditions can be eliminated then we can summarize that function `foo` will always call function `p` under all arguments settings. Note that if there is a capability checking in some paths, the capability checked will also be remarked as required by corresponding branch condition. With this insight in mind, it is obvious that we can disregard branch conditions if both 'then' and 'else' branches join together and prune these branch conditions. After pruning, we reduced branch conditions by 11% on average, and this number can reach 50% for functions with significantly more branch conditions that meet the conditions outlined above.

Leveraging OCaml bindings for LLVM [12], we collected all branch conditions through all paths. Each path includes massive serial or parallel branch conditions starting from the system call to the capability (as shown in Fig.4 Branch Condition Analysis).

4.4 Constraint Equation Generation

After extracting and organizing all the branch conditions, the path can be mathematically represented as a long expression with numerous constraints. Further, a single complete mapping is supposed to consist of all the possible paths from a specific system call to a particular capability. Therefore, the final generated mapping ought to include all the single complete mappings, which contain the paths from all the system calls and all Linux capabilities.

To achieve this, we need to analyze the serial or parallel branch conditions through a path, then combine them into a precise mathematical expression known as the constraint equation. Specifically, we focus on the constraints in terms of those symbols for the possible outcomes of each conditional branch, which requires the help of symbolic execution. In order to do this, we need to solve the challenges in handling common libraries and loops.

Loop and common library handling. In the preliminary studies, we have found that common libraries and loops may cause lots of trouble when performing the aforementioned backward-forward two-way analysis. Basically, there are two possible problems when handling common libraries and loops:

- Massive traces may be generated, which symbolic execution could not efficiently handle. One example is that a simple kernel function with only 20 lines of code will produce more than 4,000,000 lines of traces (mainly due to the deeply cascaded function calls), also known as the path explosion.
- Difficulty in handling pointer-oriented standard functions, these functions have the variables based on the pointer, making the indirect function call analysis more difficult. Commonly used build-in functions such as `strncpy` and `strncmp` are typical pointer-oriented standard functions, some of which are responsible for transferring the data from kernel space to the user space or vice versa. With various variable types and structs, the functionality of such pointer-oriented standard functions is difficult to be represented in mathematical expressions.

To address these problems, we propose using the technique of code modeling at the function level. The idea is inspired by function modeling (FM)[22], which aims to facilitate the communication as well as understanding between engineers of various disciplines and means to make use of computers for reasoning purposes. Meanwhile, we focus on the expressions in terms of those symbols for expressions and variables instead of considering their trivial properties, such as in the user space or kernel space. There are several ways to do the modeling: One is purely manual work by writing a model to describe the functionality of the target function at a high level. Specifically, this model is responsible for extracting the idea of "How-it-is-done" from the function and further representing the mathematical expression. An alternative approach is to replace it with a functional equivalent version but can be handled by the symbolic execution. The code modeling can effectively solve the path explosion and further handle the pointer-oriented standard functions in the symbolic execution.

We use case study to demonstrate the procedure and expected output of function modeling, Fig.7 shows an example of `strncpy` in Linux version 5.4.18. This function represents a typical method to traverse a string in C language. Its functionality is straightforward: From the starting position of the pointer, store the value pointed by `src` into the `dest`, and no more than `count` number of characters; in the case where the length of `src` is less than that of `count`, the remainder of `dest` will be padded with NULL.

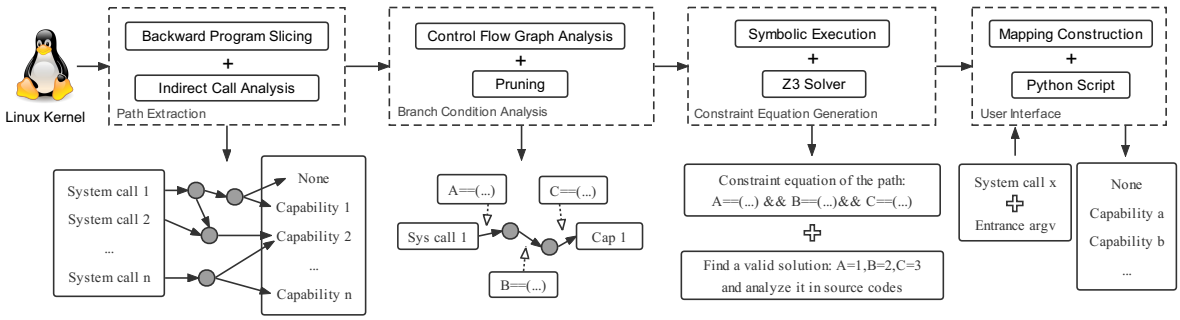
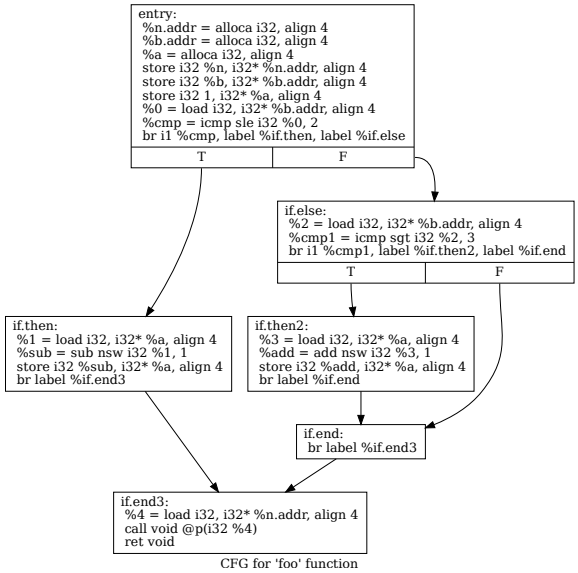


Figure 4: The Overview of Methodology and Visualized Examples



(a) CFG for `foo`

```
1   void foo(int n, int b)
2   {
3     int a = 1;
4     if (b <= 2)
5       a -= 1;
6     else if(b>3)
7       a += 1;
8     p(n);
9   }
10
```

(b) Source code for `foo`

Figure 5: CFG example

However, in the symbolic execution, we need to unfold the while loop and execute the branch condition in line 5 repeatedly. In the symbolic execution, each branch condition will fork a new path. As a result, this while loop will generate huge trace, the number of the paths is supposed to be exponentially grown, which is also known as the path explosion problem. Although there are massive branch condition in the function `strncpy`, at a high level, this function only has two branches: (1) if the length of the string from the `src` is more than `count`, only copy `count` characters to the `dest`; (2) otherwise, copy the all the characters from the `src` to the `dest`, the remainder of `dest` will be padded with NULL. Therefore, we only need to implement these two branches with a single if-else statement, instead of going to the while loop and executing branch condition repeatedly. That's the idea of function modeling. After manually writing a model to describe the functionality of the function `strncpy`, we can perfectly solve the path explosion problem in the while loop. As function modeling is a one-time cost work and most of the functions do not need function modeling while many functions have no changes or small changes when the Linux kernel version updates, we think it is acceptable to solve the path explosion problem leveraging function modeling.

```
1  static int load_module(struct load_info *info, const char
       __user *uargs, int flags)
2  {
3    ...
4    err = elf_header_check(info);
5    if (err)    goto free_copy;
6    err = setup_load_info(info, flags);
7    if (err)    goto free_copy;
8    if (blacklisted(info->name)) {
9      err = -EPERM;
10     goto free_copy;
11   }
12   ...
13 free_copy:
14   free_copy(info);
15   return err;
16 }
```

Figure 6: Pruning the error checking branches

```
1  char *strncpy(char *dest, const char *src, size_t count)
2  {
3      char *start = dest;
4      while (count && (*dest++ = *src++)) /* copy string */
5          count--;
6      if (count) /* pad out with zeroes */
7          while (--count)
8              *dest++ = '\0';
9      return(start);
10 }
```

Figure 7: Source code for `strncpy`

4.5 User Interface

After collecting the constraint equations in Â§4.4, we are able to generate the comprehensive mapping from the system calls (with their entrance arguments) to capabilities. The mapping consists

of all the paths (starting from system calls to the capabilities) together with the constraint equations through the paths. Therefore, we have proposed an algorithm that allows both the developers and application users to conveniently traverse the mapping, and further efficiently apply the mapping to practical usage. Both of them can use it to make sure the program is the least privileged. This algorithm is for the interaction between user programs and the mapping as shown below.

Algorithm 1: Interaction between user programs and the System call and Capability mapping

Input: *SysCallName*: System call name
Mapping: Mapping of System call & Capability
EA: Entrance arguments of the system call
Output: *CL*: Capability List to be triggered

```
1 CL = [];
2 cap_dict = Mapping[SysCallName];
  /* cap_dict =                                  */
  /* { CAP_1:Constraint_equations_1,             */
  /*   CAP_2:Constraint_equations_2,             */
  /*   ...}                                      */
3 foreach cap ∈ cap_dict.keys() do
4     constraint_equation = cap_dict[cap];
5     if constraint_equation(EA) then
6         CL.append(cap);
7     end
8 end
9 return CL;
```

The basic idea of this algorithm is traversing the whole mapping to find which capability will be triggered with a given system call and its entrance arguments. The capability settings are usually set only once when the user program is installed by `setcap` command with the root privilege as it assigns capabilities to a program. The user interface requires no privilege as it only reads the input system calls and arguments or scans the LLVM IR code of the user program to get the input. Then it can automatically construct the mapping of system calls and capabilities then calculates which capabilities are required by the user program. In practical usage, the developers and application users only need to find the system calls invoked in the user program together with the entrance arguments. This process can be done by the developers or by LiCA to analyze a given user program and apply our method again to the IR code of user programs to find out where the system calls are triggered and the arguments for them.

5 IMPLEMENTATION AND EVALUATION

LiCA was implemented using LLVM/Clang-10.0. It contains over 2000 lines of Ocaml/python code. We evaluated LiCA with the default kernel configuration defconfig, the (commonly-used) default configuration. All experiments were carried out on a virtual machine with Linux kernel 5.4.18. Fig. 8 shows the overview of the evaluation. We finished mapping between all 39 capabilities and 428 system calls. There are 46,094 functions and 164,627 function pairs in total, while only less than 100 functions had to be manually modeled. These functions can be decided automatically as they take much longer time in constraint equation generation than the others. We identified 16,239 call chains which can trigger capabilities. As a result, we attach a simple table of the capabilities that system calls might require in Appendix 3. We empirically evaluate it to address the following questions:

- **RQ1:** How about the correctness of LiCA with the mapping between system calls and capabilities?
- **RQ2:** How effective is LiCA in analyzing the user program?

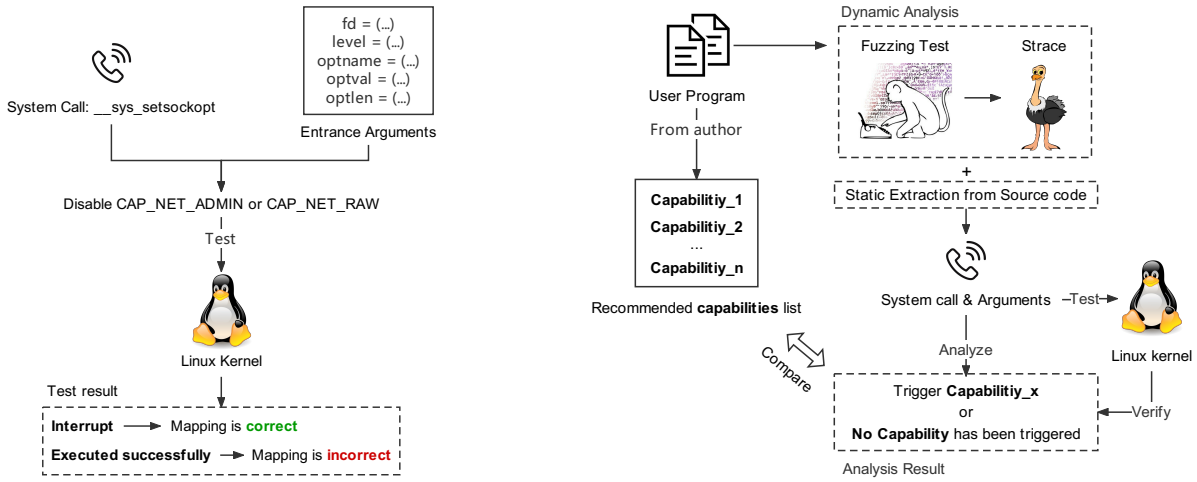
5.1 Correctness of the mapping

To answer RQ1, we prepared two approaches to verify the correctness of the mapping from system calls to Linux capabilities: the first method is static, which aims to explain the constraint equations in the mapping, further find the corresponding semantic meaning in the source code; the second one is dynamic, which leverages the Linux kernel to provide the ground truth and verify the correctness of the mapping (as shown in Fig.8a). The details of these two approaches are shown below.

Explaining the Constraint Equations In this method, we try to correspond the constraint equations we have collected to the source code in the Linux kernel, and further provides the semantic meaning of the constraints equations.

Here we use an example to explain the constrain equations in the mapping. In this part, we consider the paths that starting from the system call `setsockopt` and ending with two possibly required capabilities `CAP_NET_ADMIN` and `CAP_NET_RAW`. System call `setsockopt` is responsible for setting the option specified by the `optname` argument, at the protocol level specified by the level argument, to the value pointed to by the `optvalue` argument for the socket associated with the file descriptor specified by the `socket` argument, and it may require `CAP_NET_ADMIN` or `CAP_NET_RAW`. In the previous usage, the developer will always assign both the capabilities in case of the abnormal interruption, however, it is possible to cause an over-privileged problem. The constraint equations are shown in Table.1. We write a Python script to trace the equality and inequality in the mapping back to the source codes and validate if they can be matched. In addition, we manually check a random subset to ensure correctness. The results show that we can always find the corresponding branch conditions in the Linux kernel and we can further explain the purpose of such branch conditions. As shown in Fig.9, system call `setsockopt` will first call `__sys_setsockopt`, and then `sock_setsockopt` will be called, which is meant for all protocols to use and covers goings-on at the socket level, and everything inside is generic.

There are 7 cases to trigger `CAP_NET_ADMIN` and 2 cases to trigger `CAP_NET_RAW` (source codes are shown in Appendix). All of these cases have some common branch conditions to detect the invalid input, such as checking `fd` and return the prepared sock (line 20), or return error if the `optlen < 0` (line 17-18). Meanwhile, in `sock_setsockopt` there has branch condition to check the `optlen` (line 40). All these branch conditions have been captured during the mapping generation, hence the constraint equations can mathematically represent such branch conditions, for example, the equation `False == (optlen < 0)` corresponds to the code in line 17-18, and `False == (optlen < 4)` corresponds to the code in line 40.



(a) Evaluate Correctness of Mapping from System Call to CAP

(b) Evaluate Effectiveness in Analyzing User Program

Figure 8: Evaluation Overview

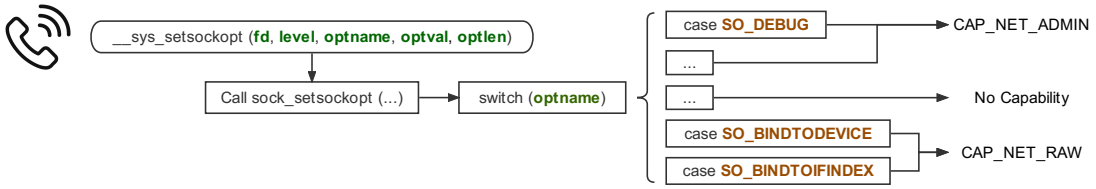


Figure 9: Real Case Example in Linux Kernel from System Call `setsockopt`

Therefore, in the mapping from the system call `setsockopt`, there has the constraint equations related to `optlen` as shown in Table.1.

Besides, considering different cases that may trigger the capability, the decisive factor is the value of the `optname`. Similar to the previous explanations, we can clearly align the constraint equations we have generated with the branch conditions in the source code. For example, when the arguments satisfy the condition `optname == SO_BINDTODEVICE == 25`, it will call the function `sock_setbindtodevice` (line 37-39). This function will further call the function `sock_setbindtodevice_locked` (line 83), which finally triggers the `CAP_NET_RAW` at line 88. This result verifies the constraint equations (The 3rd path in Table.1). In conclusion, the equality `optname == 25` or `optname == 62` correspond to triggering the `CAP_NET_RAW`; on the other hand, the `optname == 1`, or `optname == 32`, or `optname == 33`, or `optname == 36`, or `optname == 46`, or `optname == 61` or `optname == 62` correspond to triggering the `CAP_NET_ADMIN`. As we can see, these cases exactly match the constraint equations generated by LiCA.

Testing in Linux Kernel For verifying the correctness of our mapping, we use Linux kernel to provide the undeniable ground truth. In this method, we tried to directly test the system call with its entrance arguments in the Linux kernel by disabling the corresponding capability, which LiCA has found. There can be two different results after calling such system call with disabling corresponding capability: (1) If the system call can run smoothly and complete its task, it indicates the system call does not require the capability which LiCA has found, which means LiCA made a mistake in terms of the mapping. (2) If the system call interrupts during the process, it proves that the system call with these entrance arguments does need the capability, which means the mapping is correct and the result is exactly as what LiCA has analyzed.

We use an example to explain the details of verifying the correctness of our mapping with testing in Linux kernel, which is shown in Fig.8a. We used the same path as the last part. We collected the related constraint equations in the mapping (Table 1) and prepared a set of entrance arguments by enumeration. Some of the entrance arguments are struct defined in Linux kernel or not determinant in the branch condition, hence we retain the parameter names as the input arguments with mark %. We test the entrance arguments in Linux kernel by disabling one of the capabilities, and observe the execution performance in the Linux kernel, we can find that the results in Linux kernel match the results from the analysis of LiCA (Table.4 in Appendix shows part of the results).

5.2 Effectiveness in analyzing user program

The adoption of Linux capabilities has actually been very slow in the community and we find that only a small set of Linux programs used capabilities. Some programs required root privilege are highly-privilege and can benefit from the deployment of capabilities if we can identify the capabilities they used. As there is no such kind of

Path	Constrain Equations
`setsockopt -> __sys_setsockopt -> sock_setsockopt -> capable(CAP_NET_ADMIN)`	$False == (fd == 0)$ && $False == (optlen < 0)$ && $level == 1$ && $False == (optlen < 4)$ && $(optname == 1 \| optname == 32 \| optname == 33 \| optname == 46)$
`setsockopt -> __sys_setsockopt -> sock_setsockopt -> ns_capable(CAP_NET_ADMIN)`	$False == (fd == 0)$ && $False == (optlen < 0)$ && $level == 1$ && $(False == (optlen < 4)$ && $(optname == 12 \| \| optname == 36 \| \| optname == 61))$
`setsockopt -> __sys_setsockopt -> sock_setsockopt -> ns_capable(CAP_NET_RAW)`	$False == (fd == 0)$ && $False == (optlen < 0)$ && $level == 1$ && $(optname == 25 \| (False == (optlen < 4)$ && $(optname == 62)))$

Table 1: Constrain Equations of Path from System call `setsockopt` to Capability `CAP_NET_ADMIN` and `CAP_NET_RAW`

testing program set related to capability studies, we have to build our own and will later share it with the community. The evaluation program set was constructed with the following rules, to provide an accurate evaluation of LiCA:

- The program requires some capabilities or root privilege.
- The program is open-sourced so that it can be downloaded, built, and tested on our evaluation platform.

Based above rules, we then wrote some scripts to crawl data from Github automatically. After some manual vetting and removing of duplicates, we finally collected 100 programs requiring capabilities and another 10 requiring root privilege. Besides, we also add several well-known utility programs (like ping, traceroute, etc.) into our evaluation program set, serving as the ground truth to some degree. These selected programs from Github have 43.3 stars on average, and the most popular one has 953 stars.

To answer RQ2, we conducted an evaluation on the program set we collected as shown in Fig.8. First, we need to extract the system calls and their arguments. We applied two methods to do this. The first one is the dynamic approach. We leveraged radamsa [6] to randomly fuzz the input for a given program. For each program, we first generate a list of sample inputs (the length of this list depends on the number of combinations of different parameters of the program) and ensure the samples include every parameter of the program. After that, we run radamsa for each sample 10000 times and use strace [1] to trace the system calls and their arguments. In addition to this, if the result of the dynamic approach does not match the capabilities provided by the developer, we have the second approach to apply our method again to the user program and regard the system call as our target (the original target is capability). By tracing back the call chain for a system call, we can compute the changes for the arguments. Most times the arguments are constants defined in other functions, while little of them are related to the user input. For such cases, we assume that the argument will cover all input for the constraint equation. As for the external libraries, we can also collect their source code and compile it into LLVM IR. Given that the mapping is ready and only requires computing the results using the constraint equations, it takes less than two seconds to obtain the capability result when providing the system calls and their inputs for a user application.

After analyzing the user program and deriving the required capabilities by analyzing the system calls together with their entrance arguments, we use the output result to justify the correctness of the recommended capability list from the program author: if the capabilities reported by LiCA matches the recommended capability list from the author, it means the program has no privilege problem; if our output required capabilities consist of a capability that is not included in the recommended capability list, we find it's an insufficient privileges problem, it will lead to exceptions and break of normal functions; on the other hand, if there is a capability in the recommended capability list which is not included in our output results, we find it's an over-privileged problem, which is supposed to be avoided. Both the problems are double-checked by manually analyzing the program, including using gdb to run the program step by step and tracing back to the source code level to check each system call with its arguments. For the 114 programs we collected, we discovered three cases requiring more capabilities than they need, three cases asking users to assign root privilege, one inducing the user to assign root privilege without providing a capability list, and one program having different capability settings in different applications.

However, static analysis indeed may introduce false positives and false negatives. In this work, false positives indicate that LiCA report the capabilities which are not required by the user program; false negatives indicate that the program requires specific capabilities but not reported by LiCA. We aims to be conservative and try to remove all false positives, as false positives will bring unnecessary capabilities and lead to a wider attacking surface. We used A/B testing to evaluate the false positives: suppose LiCA reports N capabilities for a program, we run the program with each combination of (N-1) capabilities. If there is an error for every run, it means the list of capabilities reported by LiCA has no false positives. Otherwise, we can compute the false positives. In the evaluation of the collected program set, no false positives have been observed.

Meanwhile, false negatives are spotted during the evaluation. For example, we have come across false negatives caused by indirect calls (some function pairs missed in the mapping), even though we have used state-of-the-art techniques. There are 139,638 functions and 210,720 function call pairs in the default setting, and Pex reports 2,529 indirect call pairs. As Pex can cover over 92% of the indirect calls, the false negatives of LiCA should be less than 8%.

5.3 Capability setting problems case study

We identified multiple capability setting problems from the program set we collected:

Problem 1: Abusing root privilege. When searching key words like "run with root privilege" in Github, over 88 millions of code

Tool name	Description	Capabilities set by default	Capabilities reported by LiCA	Problem
Netsniff-ng [38]	Linux networking toolkit.	CAP_NET_RAW, CAP_NET_ADMIN, CAP_SYS_ADMIN, CAP_IPC_LOCK	part of the capabilities for each tool as shown above	over capability set
Mcsauna [16]	hottest keys tracking tool on memcached instances.	CAP_NET_RAW, CAP_SETPCAP	CAP_NET_RAW	over capability set
DOSBox-pigeos [9]	a modified version of DOSBox (emulator of DOS program).	CAP_NET_RAW, CAP_SETPCAP	CAP_NET_RAW	over capability set
iftop [13]	display bandwidth usage on interfaces	root	CAP_NET_RAW	requesting root privilege
probedhcp [4]	a tool for sending IPv4 DHCP packets with increasing TTL	root	CAP_NET_RAW	requesting root privilege
scan_iface [11]	a tool for scanning AP	root	CAP_NET_RAW	requesting root privilege
macchanger [5]	a tool for maniputation of MAC addresses of network interfaces	none	CAP_NET_ADMIN	no capability list provided

(a) Capability setting problems

Tool name	Description	# of system calls used	CAP1	CAP2	CAP3	CAP4
netsniff-ng	a zero-copy packet analyzer, pcap capturing/replaying tool	22	✓	✓	✓	✓
trafgen	a multithreaded low-level zero-copy network packet generator	25	✓	✓	×	×
mausezahn	high-level packet generator for appliances with Cisco-CLI	6	✓	×	×	×
ifpps	a top-like kernel networking and system statistics tool	8	×	✓	×	×
curvetun	a lightweight curve25519-based multiuser IP tunnel	16	✓	✓	✓	×
astraceroute	an autonomous system trace route and DPI testing utility	11	✓	✓	×	×
flowtop	a top-like netfilter connection tracking tool	16	×	✓	×	×
bpfc	a Berkeley Packet Filter compiler, Linux BPF JIT disassembler	12	×	×	×	×

✓: capabilities reported by LiCA ×: set by default but not reported by LiCA
CAP1: CAP_NET_RAW; CAP2: CAP_NET_ADMIN; CAP3: CAP_IPC_LOCK; CAP4: CAP_SYS_ADMIN

(b) Netsniff-ng toolkit result

Tool name	Description	# of system calls used	Capabilities set by default	Capabilities reported by LiCA
ping	a simple utility used to check whether a network is available and if a host is reachable.	12	CAP_NET_RAW	CAP_NET_RAW
traceroute	a tool to track the route packets taken from an IP network on their way to a given host.	8	CAP_NET_RAW	CAP_NET_RAW
tcpdump	a tool to analyze traffic sourced or destined to your own host or capture traffic between two or more endpoints	13	CAP_NET_RAW, CAP_NET_ADMIN	CAP_NET_RAW, CAP_NET_ADMIN
httpd(Apache2)	an open-source HTTP server	52	CAP_NET_BIND_SERVICE	CAP_NET_BIND_SERVICE

(c) Linux utility result

Table 2: Capabilities reported by LiCA for the programs in the wild

results are presented. Three millions of them use Markdown language, which indicates "run with root privilege" may be found in Readme or Makefile file. Here we present three programs which need to be run with root privileges (their authors write this in their readme file). It may be due to sometimes it is not clear what capabilities does a program need even for their authors so they just require root privilege for convenience. For instance, iftop [13] is a tool used for displaying bandwidth usage on an interface with "iftop must be run as root." is shown on its git main page. However, setting up `CAP_NET_RAW` capability has already fulfilled its needs.

Problem 2: Setting more capabilities than they need. For the programs from Github, discrepancies are observed in three programs. Manual inspections and testings show that some open-source programs are recommending more capabilities than actually needed. We will show the details below:

- Netsniff-ng [38] is a free Linux networking toolkit including 8 tools. In the installation description, it requires setting four capabilities: `sudo setcap cap_net_raw, cap_ipc_lock, cap_sys_admin, cap_net_admin=eip {toolname}` As a result, we must be a root user or we set above 4 capabilities for each tool if we want to run it. It is strange for a networking tool to have such a crucial capability `CAP_SYS_ADMIN`. So we evaluate the 8 tools in it and the results are shown in Table 2b. As expected, only one tool (netsniff-ng) needs all 4 capabilities for execution. Most of the tools only need one or two capabilities while there is one tool (tpfc) that needs no capability. It is used for translating Berkeley Packet Filter (BPF) assembler-like mnemonics into a numerical or C-like format, that can be read by tools such as `iptables`.
- Mcsauna [16] is a tool allows user to track the hottest keys on Memcached instances, reporting back in a graphite-friendly format. After analyzed by LiCA, we found that although it set cap_net_raw and cap_setpcap by default cap_setpcap is not necessary and not used in this program.
- DOSBox-pigeos [9] is a modified version of DOSBox [8], which is a free and open-source emulator of an Intel x86 personal computer designed for the purpose of running software created for disk operating systems on IBM PC compatibles, primarily DOS video games. It also set cap_net_raw and cap_setpcap by default and only cap_net_raw is reported by LiCA. It is dangerous to assign cap_setpcap capability to this program as it is an emulator which can run programs inside it. For example, CVE-2019-12594 [3] shows a program running inside DOSBox can access the contents of /proc (e.g. /proc/self/mem).

For Linux utilities, as many of them do not require any capabilities, we selected some most commonly used programs for evaluation. The results are illustrated in Table 2c. LiCA is able to report the same set of needed capabilities. Interestingly, by detailed checking the documents and settings of Apache2, we found that this program runs as a root user by default. Thus it will have full capabilities at most times, which does not follow the least privilege principle. As a normal user, only `CAP_NET_BIND_SERVICE` is required for binding to privileged ports such as 80 and 443.

Problem 3: Missing privilege settings. Apart from analyzing programs with capabilities settings, we also evaluated some programs with no capabilities set by default - neither mentioning the capabilities they need nor asking users to run as root. However, for one program Macchanger [5], LiCA reports `CAP_NET_RAW` capabilities. We manually checked this program and found an "operation did not permit" error when we tried to execute it as a normal user with no privilege. Such a case shows that some programs are missing privilege settings and leave this problem to users. Usually, the user will use root privilege to run them as they do not know which capabilities are used, violating the principle of least privilege.

Problem 4: Mismatch capability settings between users for the same program. Addition to problem 3 above, users may add the capabilities by themselves, for example, most of the users set `CAP_NET_RAW` and `CAP_NET_ADMIN` for tcpdump, but in one repository [10], `CAP_NET_BIND_SERVICE` is also set to it. Actually, it is hard to decide proper capabilities for users as they can only refer to the descriptions of the programs.

All these problems show that it is essential to set correct capabilities to user programs, regardless of developer and users and LiCA is able to fill the gap of this.

5.4 Comparison with prior works

We compare LiCA with related works TCLP [29]. TCLP is a dynamic analysis system that monitors the system calls in run time to find the least capabilities required for running the containers, which also generate a coarse mapping between system calls and capabilities. It monitors which capabilities are checked when system calls are triggered in run time. However, iIf a system call can trigger multiple capabilities and a program calls this system call, TCLP will report all the capabilities regardless whether these capabilities are really needed. Moreover, the dynamic analysis may not be able to cover all execution paths, which may result in false negatives. LiCA generates fine-grained mappings between system calls and capabilities and is able to decide the precise capability required by a system call under specific conditions. We found TCLP will generate more false positives than ours. For the 8 programs reported by LiCA, TCLP can report three of them with the same capabilities with LiCA but more capabilities for the other five. We manually analyzed the difference and found that TCLP reported them because some functions in the program can trigger those capabilities with some arguments but those arguments will never be fulfilled. A sample is the handling of the setsockopt system call in the testing program mausezahn [38]. TCLP will report the requirement of `CAP_NET_ADMIN` as it may appear in setsockopt system call, but LiCA will not because it will analyze the arguments to setsockopt and find that arguments provided inside this specific testing program mausezahn will eventually not trigger `CAP_NET_ADMIN`.

6 DISCUSSION

While LiCA has made a first step towards find-grained and path-sensitive Linux capability analysis with considering the indirect call, it still has a number of limitations.

First, some situations will fail the static analysis. When handling the common libraries, the function resource allocation may be beyond the scope of static analysis, as some functions will ask for the resource related to the user management of processes. One example is function `inode_owner_or_capable`, which checks current task permissions of the inode, and returns true if current either has `CAP_FOWNER` , or owns the file. Function `current_fsuid` is responsible for checking whether this inode owner owns the file, which is related to the user identification of the file system. This function is vital in determining whether trigger the capability `CAP_FOWNER`. However, for static analysis, we are not able to obtain the process or user information in terms of the file system. Hence, it is impossible to generate the constraint equations towards `current_fsuid` which represent the branch condition in line 4. Although we mark the branch condition and retain such function in the final constraint equations for the reminder, the static analysis cannot give the final result about whether it can trigger the capability. It is needed to be combined with further dynamic tests in practical usage.

```
1 bool inode_owner_or_capable(const struct inode *inode)
2 {
3   struct user_namespace *ns;
4   if (uid_eq(current_fsuid(), inode->i_uid))
5     return true;
```

```
  ns = current_user_ns();
  if (kuid_has_mapping(ns, inode->i_uid) && ns_capable(ns
      , CAP_FOWNER))
    return true;
  return false;
}
```

Second, when a program creates a child process by calling execve() to invoke another program, the capability of the child is the intersection between the capability set from the parent and the child. For these situations, both child and parent's programs must be considered, so that the capability list for the parent program must be added with the capabilities of the child, which may also cause overprivileged problems.

7 RELATED WORK

7.1 Linux kernel analysis

PeX [41] is 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. By analyzing discretionary access control (DAC), Capabilities, and Linux security module (LSM) permission checks in the latest Linux kernel v4.18.5 using PeX, 36 new permission check bugs are reported. Yamaguchi et al. [39] developed Chucky, a method to expose missing checks in source code. Since many vulnerabilities result from insufficient input validation, Chucky uses taint analysis to identify anomalous or missing conditions linked to security-critical objects. Sparse [37] is a semantic checker for C programs, which can be used to find a number of potential problems with kernel code.

7.2 Linux capability security analysis

There is a series of studies on Linux capabilities. Hallyn et al. [23] demonstrates how to use the Linux implementation of these capabilities and how capabilities enhance the security of the modern Linux system. AutoPriv [24] uses whole-program analysis to determine where programs use privileges and then transforms programs to remove unnecessary privileges during their execution, which helps programmers use capabilities more easily. Lin et al. [30] provide an in-depth analysis on the privilege escalation attacks, especially the effectiveness of kernel security mechanisms (i.e., Capability), container mechanisms, and CPU protection mechanisms on preventing privilege escalation against the container platform. PrivAnalyzer [20] is an automated tool that measures how effectively programs use Linux capabilities. It aims to help security-critical software developers to minimize privileges use. TCLP [29] is a dynamic analysis system that monitors the system calls in run time to find the least capabilities required for running the containers.

7.3 Indirect call analysis

In the Linux kernel analysis, handling the indirect call is a hard problem, as the modern compilers do not recognize indirect call targets by default. MLTA [32] is a new approach for effectively refining indirect-call targets for C/C++ programs. It relies on an observation that function pointers are commonly stored into objects whose types have a multi-layer type hierarchy; before indirect calls, function pointers will be loaded from objects with the same type hierarchy "layer by layer". By matching the multi-layer types of function pointers and functions, MLTA can dramatically refine indirect-call targets.

8 CONCLUSION

This paper presents LiCA, a fine-grained and path-sensitive Linux capability analysis framework, which can generate accurate mappings between system calls and capabilities. LiCA utilized LLVM, symbolic execution, and existing indirect call analysis methods to overcome the path explosion problem inside the Linux kernel source code. We evaluated LiCA on the standard Linux utility programs as well as open source projects from GitHub. For standard Linux utility programs, our framework reports all the needed capabilities that are same with what utility programs require, while for code from GitHub, discrepancies are observed. Manual inspections and testings show that those open source programs are recommending more capabilities than what they actually needed. We believe that LiCA allows developers to validate the capability setting for their programs to achieve the least privilege principle.

ACKNOWLEDGMENTS

We want to thank our shepherd Nathan Burow and all the anonymous reviewers for their valuable comments. This work was supported in part by National Key Research & Development Project of China (Grant No. 2019YFB1804400), and Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund (No. 14209720).

REFERENCES

[1] 1996. strace(1) - Linux man page. https://linux.die.net/man/1/strace
[2] 2018. https://nxnjz.net/2018/08/an-interesting-privilege-escalation-vector-getcap/
[3] 2019. CVE - CVE-2019-12594. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12594.
[4] 2021. aledwmorris/probedhcp: Simple program for sending IPv4 DHCP packets with increasing TTL. https://github.com/aledwmorris/probedhcp/tree/master.
[5] 2021. alobbs/macchanger: GNU MAC Changer. https://github.com/alobbs/macchanger.
[6] 2021. aoh/radamsa: a general-purpose fuzzer. https://github.com/aoh/radamsa.
[7] 2021. Becoming Root Through Overprivileged Processes | by Vickie Li | Better Programming. https://betterprogramming.pub/becoming-root-through-overprivileged-processes-f26f83e18059.
[8] 2021. DOSBox, an x86 emulator with DOS. https://www.dosbox.com/.
[9] 2021. hastho/dosbox-pigeos. https://github.com/hastho/dosbox-pigeos.
[10] 2021. jupyter. https://github.com/jsand66/jupyter/blob/master/setcap.txt.
[11] 2021. kongbiji/scan_iface. https://github.com/kongbiji/scan_iface/tree/master.
[12] 2021. The OCaml bindings distributed with LLVM. https://opam.ocaml.org/packages/llvm/
[13] 2021. Paul Warren / iftop · GitLab. https://code.blinkace.com/pdw/iftop
[14] 2021. PayloadsAllTheThings/Linux - Privilege Escalation.md at master · swisskyrepo/PayloadsAllTheThings · GitHub. https://github.com/swisskyrepo/PayloadsAllTheThings.
[15] 2021. Privilege escalation via Docker - Chris Foster. https://fosterelli.co/privilege-escalation-via-docker.html.
[16] 2021. reddit/mcsauna: Track hottest memcached keys by regex in a graphite-friendly format. https://github.com/reddit/mcsauna.
[17] 2022. The LLVM Compiler Infrastructure Project. https://llvm.org/.
[18] 2022. LLVM, Intermediate Representation, and Static Analysis! Oh My! - GaZAR. https://gazar.eu/2021/02/21/llvm-intermediate-representation-and-static-analysis-oh-my/.
[19] Canonical. 2019. http://manpages.ubuntu.com/manpages/precise/man7/capabilities.7.html
[20] John Criswell, Jie Zhou, Spyridoula Gravani, and Xiaoyu Hu. 2019. PrivAnalyzer: Measuring the Efficacy of Linux Privilege Use. *Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019* (2019), 593–604. https://doi.org/10.1109/DSN.2019.00065
[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[22] M.S. Erden, H. Komoto, T.J. van Beek, V. D'Amelio, E. Echavarria, and T. Tomiyama. 2008. A review of function modeling: Approaches and applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22, 2 (2008), 147–169. https://doi.org/10.1017/s0890060408000103
[23] Serge E Hallyn and Andrew G Morgan. 2008. Linux capabilities: Making them work. (2008).
[24] Xiaoyu Hu, Jie Zhou, Spyridoula Gravani, and John Criswell. 2018. Transforming code to drop dead privileges. *Proceedings - 2018 IEEE Cybersecurity Development Conference, SecDev 2018* February 2019 (2018), 45–52. https://doi.org/10.1109/SecDev.2018.00014
[25] Ken Kennedy. 1978. Use-definition chains with applications. *Computer Languages* 3, 3 (1978), 163–179. https://doi.org/10.1016/0096-0551(78)90009-7
[26] Michael Kerrisk. 2012. CAP_SYS_ADMIN: the new root. https://lwn.net/Articles/486306/
[27] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[28] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. 2010. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs. In *2010 19th IEEE Asian Test Symposium*. 59–64. https://doi.org/10.1109/ATS.2010.19
[29] Suyeol Lee, Jaehyun Nam, Junsik Seo, and Seungwon Shin. 2019. Poster: TCLP: Enforcing least privileges to prevent containers from kernel vulnerabilities. *Proceedings of the ACM Conference on Computer and Communications Security* (2019), 2665–2667. https://doi.org/10.1145/3319535.3363282
[30] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 418–429. https://doi.org/10.1145/3274694.3274720
[31] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. https://doi.org/10.1145/3319535.3354244
[32] Kangjie Lu and Hong Hu. 2019. Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. 1867–1881. https://doi.org/10.1145/3319535.3354244
[33] Novell. 2020. Home · Wiki · AppArmor / apparmor. https://gitlab.com/apparmor/apparmor/-/wikis/home
[34] Jan Obdržálek and Marek Trtík. 2011. Efficient Loop Navigation for Symbolic Execution. In *Automated Technology for Verification and Analysis*, Tevfik Bultan and Pao-Ann Hsiung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 453–462.
[35] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.
[36] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.
[37] Linus Torvalds. 2003. Sparse. https://www.kernel.org/doc/html/v4.14/dev-tools/sparse.html
[38] Maciej Treder. 2020. ng toolkit. http://netsniff-ng.org/
[39] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. *Proceedings of the ACM Conference on Computer and Communications Security*. https://doi.org/10.1145/2508859.2516665
[40] Z3Prover. 2008. Z3Prover/z3. https://github.com/Z3Prover/z3
[41] Tong Zhang, Wenbo Shen, Ahmed M. Azab, Dongyoon Lee, Changhee Jung, and Ruowen Wang. 2019. PEX: A permission check analysis framework for linux kernel. *Proceedings of the 28th USENIX Security Symposium* (2019), 1205–1220.

APPENDIX

```
#define SOL_SOCKET              0xffff
#define SO_DEBUG                0x0001
#define SO_SNDBUFFORCE          0x100a
#define SO_RCVBUFFORCE          0x100b
#define SO_PRIORITY             12
#define SO_BINDTODEVICE         25
#define SO_MARK                 36
#define SO_BUSY_POLL            46
#define SO_TXTIME               61
#define SO_BINDTOIFINDEX        62

static int __sys_setsockopt(int fd, int level, int
     optname, char __user *optval, int optlen)
{
  ...
  struct socket *sock;

  if(optlen < 0)
    return -EINVAL;

  sock = sockfd_lookup_light(fd, &err, &fput_needed);

    ...

  if(level == SOL_SOCKET)
    err = sock_setsockopt(sock, level, optname, optval,
     optlen);

  ...
out_put:
    ...
  }
  return err;
}

int sock_setsockopt(struct socket *sock, int level, int
     optname, char __user *optval, unsigned int optlen)
{
  ...
  if(optname == SO_BINDTODEVICE)
    /* Trigger the CAP_NET_RAW */
    return sock_setbindtodevice(sk, optval, optlen);
  if(optlen < sizeof(int))
    return -EINVAL;

    switch (optname) {
  case SO_DEBUG: /* Trigger the CAP_NET_ADMIN */
    if(val && !capable(CAP_NET_ADMIN))
    {...}
  case SO_SNDBUFFORCE: /* Trigger the CAP_NET_ADMIN */
    if(!capable(CAP_NET_ADMIN))
    {...}
  case SO_RCVBUFFORCE: /* Trigger the CAP_NET_ADMIN */
    if(!capable(CAP_NET_ADMIN))
    {...}
  case SO_PRIORITY: /* Trigger the CAP_NET_ADMIN */
    if((val >= 0 && val <= 6) || ns_capable(sock_net(sk)
     ->user_ns, CAP_NET_ADMIN))
    {...}
  case SO_MARK: /* Trigger the CAP_NET_ADMIN */
    if(!ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN))
    {...}
  case SO_BUSY_POLL: /* Trigger the CAP_NET_ADMIN */
    if((val > sk->sk_ll_usec) && !capable(CAP_NET_ADMIN))
    {...}
  case SO_TXTIME: /* Trigger the CAP_NET_ADMIN */
    if(!ns_capable(sock_net(sk)->user_ns, CAP_NET_ADMIN))
    {...}
  case SO_BINDTOIFINDEX: /* Trigger the CAP_NET_RAW */
    ret = sock_setbindtodevice_locked(sk, val);
    break;
  ...
}
```

System calls	Capabilities may require by some branch conditions
linkat	CAP_DAC_READ_SEARCH
mincore, mmap_pgoff	CAP_FOWNER
setitimer	CAP_FSETID
mlock, mlockall, mmap, io_setup, io_uring_setup, shmctl, unshare, get_mempolicy	CAP_IPC_LOCK
tkill, kill, rt_tgsigqueueinfo	CAP_KILL
mknod	CAP_MKNOD
mq_timedsend, mq_notify	CAP_NET_RAW
setreuid, setuid, setresuid, setfsuid	CAP_SETUID
setgid, setfsgid	CAP_SETGID
ptrace	CAP_SYS_ADMIN, CAP_SYS_PTRACE
add_key, mount, umount, quotactl, pivot_root, swapon, swapoff, sethostname, setdomainname, ioprio_set, accept, pipe, setns, madvise, fanotify_init, keyctl	CAP_SYS_ADMIN
chroot	CAP_SYS_CHROOT
vhangup, init_module, delete_module	CAP_SYS_MODULE
adjtimex	CAP_SYS_TIME
syslog, inotify_init, getitimer, recvmmsg, clock_gettime, getdents64, sendmmsg, sendmsg, lookup_dcookie, timerfd_gettime	CAP_SYSLOG, CAP_SYS_ADMIN
timerfd_settime	CAP_WAKE_ALARM
getxattr	CAP_FOWNER, CAP_SYS_ADMIN
setxattr, removexattr	CAP_FOWNER, CAP_SETFCAP, CAP_FSETID, CAP_SYS_ADMIN
shmat	CAP_FOWNER, CAP_IPC_LOCK
execve, execveat	CAP_FOWNER, CAP_SYS_ADMIN
futex	CAP_IPC_LOCK, CAP_SYSLOG
shutdown, close, sync	CAP_KILL, CAP_SYS_ADMIN
setsockopt	CAP_NET_ADMIN, CAP_NET_RAW
read	CAP_SYS_RESOURCE, CAP_SYS_ADMIN
getrusage, times	CAP_CHOWN, CAP_FSETID, CAP_SETUID
poll	CAP_SYS_RESOURCE, CAP_KILL, CAP_SYS_ADMIN
connect, nice, sched_setscheduler, sched_setattr	CAP_SYS_NICE, CAP_KILL, CAP_SYS_ADMIN
bind	CAP_NET_BIND_SERVICE, CAP_KILL, CAP_SYS_ADMIN
send	CAP_NET_ADMIN, CAP_SETGID, CAP_SETUID
utimes, ftruncate, sysctl, truncate, fchown, fchownat, fchmod, fchmodat, readv, preadv, writev, pwritev, pwrite64	CAP_CHOWN, CAP_FSETID, CAP_FOWNER, CAP_SETUID
fcntl	CAP_CHOWN, CAP_FSETID, CAP_LEASE, CAP_SETUID
fork	CAP_CHOWN, CAP_FSETID, CAP_NET_RAW, CAP_SETUID
dup3, sched_rr_get_interval	CAP_CHOWN, CAP_FSETID, CAP_FOWNER, CAP_SETUID, CAP_SYS_ADMIN
ioctl	CAP_SYSLOG, CAP_NET_ADMIN, CAP_SYS_ADMIN, CAP_SYS_TTY_CONFIG, CAP_SYS_RAWIO
open	CAP_CHOWN, CAP_SYS_RESOURCE, CAP_FOWNER, CAP_FSETID, CAP_SETUID, CAP_DAC_READ_SEARCH, CAP_SYS_ADMIN
write	CAP_CHOWN, CAP_SYS_RESOURCE, CAP_SYSLOG, CAP_FSETID, CAP_FOWNER, CAP_SETUID, CAP_SYS_ADMIN
exit	CAP_CHOWN, CAP_FSETID, CAP_KILL, CAP_SETUID, CAP_SYS_ADMIN, CAP_SYS_MODULE, CAP_NET_RAW, CAP_IPC_LOCK

Table 3: System calls and the capabilities may be required

fd	level	optname	optval	optlen	Trigger Capability	Execution without CAP_NET_ADMIN	Execution without CAP_NET_RAW
%fd	6	1	%0	4	Null	Success	Success
%fd	1	2	%0	4	Null	Success	Success
%fd	1	2	%1	4	Null	Success	Success
%fd	1	9	%0	4	Null	Success	Success
%fd	0	10	%0	4	Null	Success	Success
%fd	1	12	%0	4	CAP_NET_ADMIN	Interrupt	Success
%fd	263	20	%0	4	Null	Success	Success
%fd	1	25	%a	%conv	CAP_NET_RAW	Success	Interrupt
%fd	41	26	%0	4	Null	Success	Success

Table 4: Analysis Result with system call setsockopt and its Entrance Arguments when Disabling the CAP_NET_ADMIN or CAP_NET_RAW in Linux Kernel

```
71 static int sock_setbindtodevice(struct sock *sk, char
       __user *optval, int optlen)
72 {
73   ...
74 #ifdef CONFIG_NETDEVICES
75     ...
76   ret = sock_setbindtodevice_locked(sk, index);
77   ...
78 out:
79 #endif
80   return ret;
81 }
82
83 static int sock_setbindtodevice_locked(struct sock *sk,
       int ifindex)
84 {
85   ...
86 #ifdef CONFIG_NETDEVICES
87   ...
88   if(!ns_capable(net->user_ns, CAP_NET_RAW))
89     goto out;
90   ...
91 out:
92 #endif
93   return ret;
94 }
```

Source code from system call setsockopt