SysCap: Profiling and Crosschecking Syscall and Capability Configurations for Docker Images

Yunlong Xing*, Jiahao Cao‡, Xinda Wang*, Sadegh Torabi*, Kun Sun*, Fei Yan†, Qi Li‡
* *George Mason University, Fairfax, VA, USA*
† *Wuhan University, Wuhan, China*
‡ *Tsinghua University, Beijing, China*

***Abstract*—Due to its advantages of faster start-up speed and better resource utilization efficiency, container technology has been widely deployed in software deployment. However, the benefits of containers come at the cost of weak isolation for the underlying shared OS kernel. To enhance the security of containers, it is critical to customize secure configurations for each specific container, including the system call list and the capability list. However, existing solutions mainly focus on system call profiling and most of these approaches still demand huge human efforts to manually configure and successfully run each container. Moreover, the dependency between capability and system call has not been considered and cross-checked during the profiling process. In this paper, we develop a toolkit named SysCap to automatically customize required system calls and capabilities for Docker images. SysCap provides a static analyzer tool to construct a libc-to-syscall mapping via analyzing the libc and a syscall-to-capability mapping via analyzing the Linux kernel. When given a Docker image, SysCap parses the Docker image statically to obtain the binary-level called functions in the target layer and then queries them with the libc-to-syscall mapping to obtain the required system calls. Next, SysCap queries the obtained system calls with the syscall-to-capability mapping to obtain the required capabilities. Thus, SysCap can customize a secure configuration of system call and capability for a given Docker image. We test SysCap on the top 193 Docker images from Dockerhub, and the experimental results show that SysCap works on all images and can reduce the attack surface effectively.**

***Index Terms*—syscall profiling, capability profiling, container security**

I. Introduction

Containers have been widely deployed in cloud environments due to their convenience and flexibility. Compared to traditional virtual machines (VM) that run their own operating systems (OS), containers occupy fewer system resources, start-up much faster, and have better performance. Nevertheless, the benefits of containers come at the cost of weak isolation. When containers share the same underlying OS kernel, malicious tenants with access to a container may exploit kernel interfaces to escape into the host kernel space [1] and compromise other containers running on the same host.

Docker and similar technologies take a layered approach to security, using several distinct Linux security mechanisms to provide process isolation. Both capabilities [2] and seccomp filters [3] play a part in reducing the risks of container escape. The capability mechanism divides the traditional root privilege into multiple distinct units, which can be independently enabled and disabled. When running a Docker container, there are 14 out of 41 capabilities enabled by default [4]. Since system calls are the entry points for container processes to exploit kernel vulnerabilities, seccomp [3] has been adopted to limit available system calls in a syscall whitelist for containers. For Docker containers, there are 289 out of 333 system calls enabled by default [5].

Given the increasing number of diverse container images, it is crucial to automatically customize the capability whitelist and the system call whitelist for a given container. However, there exist two remaining challenges. First, the existing system call profiling solutions still require heavy human efforts when generating the system call whitelist for containers [6]–[10]. BOXMATE [6], DockerSlim [10], SPEAKER [7], and Cimplifier [8] track invoked system calls by manually configuring and then running the container for a time period, leading to an incomplete system call coverage. Confine [9] combines static analysis with dynamic analysis to improve the system call coverage for a container. It still requires specialists to manually configure and run each container for separating application binaries and required libraries from container images. Second, when profiling both system calls and capabilities, we should consider the dependency between capability and system call. We observe that the capability has been checked as a gatekeeper before certain system calls are triggered. For instance, system calls `sethostname` and `mount` require the `CAP_SYS_ADMIN` capability for their normal running. However, when those system calls are invoked, the additional capabilities are not listed in the default capability list. On the other side, the default capability list may still assign too much privileges for containers. To reduce the attack surface, it is important to remove the unnecessary capabilities.

In this paper, we develop a software toolkit named SysCap to help automatically profile both the necessary system calls and the required capabilities for Docker images. We adopt a static analysis approach to analyze Docker images and related libraries without running the containers. It targets automatically generating the correlated whitelists for both system calls and capabilities without involving heavy human efforts. It consists of three major modules, i.e., *docker image parser*, *libc-to-syscall mapper*, and *syscall-to-capability mapper*.

This work was partially supported by NSF CNS-1815650, Hubei Province Key Research and Development Program of China under Grant No. 2020BAA003, NSFC under Grant 62132011, and the Shuimu Tsinghua Scholar Program.

The docker image parser accurately locates target binaries of applications in Docker images and extracts the called functions. To accurately locate the required binaries of the target application in a hierarchical Docker image, we use the Dockerfile to reconstruct the linear relation between different layers that are stored disorderly in an image and automatically deduce the functionality of each layer. Then we extract the called functions in the binaries obtained from the target layer.

The libc-to-syscall mapper constructs a mapping table between libc functions and the related system calls and then obtains the required system calls for functions extracted in Docker image parsing, via identifying system calls in libc and building a complete calling relation. First, when invoking a system call, the corresponding system call number is usually passed to specific registers (i.e., `eax` or `rax`), and a software interrupt will be triggered to complete the kernel service. However, there are different ways to pass the system call numbers to the registers, increasing the difficulty of identifying a specific system call. Thus, we propose a method named Register Value Backtracking to identify the passing patterns of system call numbers. Second, after identifying all system calls, we construct a complete calling graph for libc to determine what system calls are invoked by each libc function. Considering a large number of libc functions and complex calling relations, the traditional directed graph construction algorithms may not work due to the huge performance overhead. Instead, we propose an efficient algorithm to improve the performance and completeness when building the calling graph. Finally, we obtain the required system calls by comparing the called functions from the image parser and the libc-to-syscall mapping.

The syscall-to-capability mapper is responsible for generating a mapping table between system calls and their required capabilities. It is done by identifying all positions that check for capabilities and their corresponding parameters and performing system call reachability analysis. First, it compiles the Linux kernel into one bytecode using `wllvm` [11] and resolves indirect calls based on struct-type [12]. Then it identifies all capability checking functions that take a specific capability as a parameter in the kernel IR and locates functions calling these capability checking functions. Meanwhile, it records the related parameters for capability. Next, it performs control-flow analysis based on the kernel control flow graph. If there is a path from a specific system call to the capability checking positions, we keep this correspondence to the syscall-to-capability mapping. Finally, after comparing the invoked system calls with the syscall-to-capability mapping, we can obtain the required capability list.

We conduct extensive experiments to evaluate the performance of our tool over the top 193 popular official Docker images that cover different application categories. SysCap can automatically extract system calls for all 193 images and SysCap reduces about 62% unnecessary system calls on average. By comparing with the results from dynamic analysis methods, we find that all the system calls obtained by dynamically tracking are subsets of those obtained by SysCap. Meanwhile, SysCap increases 6.6% more system calls on average than the dynamic analysis methods. As for the generated capability list, 78.5% of containers require less than 7 capabilities. On average, 5.3 capabilities are required and only about 4% of containers require more than 14 capabilities.

In summary, we make the following contributions:

- We develop a toolkit called SysCap to statically profile both system calls and capabilities of Docker images by considering the dependency between syscalls and capabilities, reducing the attack surface of containers hugely.
- We implement our toolkit by building the connection between libc-to-syscall and syscall-to-capability mapping. It is the first work to take the capability mechanism into consideration when profiling the Docker configurations.
- We evaluate our toolkit using 193 popular official Docker images the result shows that SysCap can automatically and successfully generate both the system call whitelist and required capabilities for various Docker images.

II. Background

A. Seccomp

The latest Linux kernel (v5.18) provides about 350 system calls. However, most of them are not needed for running a specific application. To prevent attackers from abusing extra system calls, secure computing mode (seccomp) has been integrated into the Linux kernel to limit available system calls. There are two working modes for seccomp, namely, strict mode and filter mode. When working in the strict mode, a user process can only invoke 4 system calls, i.e., `read`, `write`, `exit`, and `sigreturn`, and other system calls tried by user processes will be blocked. In the filter mode, a customized system call whitelist is enforced on a target process.

B. Syscall and Capability

The implementation of syscall is gated by corresponding capabilities. That is, enabling a syscall and disabling the required capabilities cannot complete requesting the kernel services, and vice versa. The latest Linux kernel contains 41 capabilities and each capability represents a bit in a bitmap of a process. When running a container, there are 14 capabilities enabled by default [4]. However, for some operations, it will require the additional capability that is not in the default list, e.g., `sethostname` requiring the privilege of `CAP_SYS_ADMIN`. For some containers, a few capabilities in the default list will not be required and thus can be removed.

III. System Overview

SysCap aims to automatically generate a secure configuration for a Docker image to disable the unused system calls and capabilities without impacting the normal operations of the application. Fig. 1 shows the architecture of SysCap that consists of three main parts: *docker image parser*, *libc-to-syscall mapper*, and *syscall-to-capability mapper*.

The docker image parser extracts the called functions from the given Docker image. The main challenge in this step is to accurately locate the required binaries of the target application in a hierarchical Docker image. To solve it, we use the

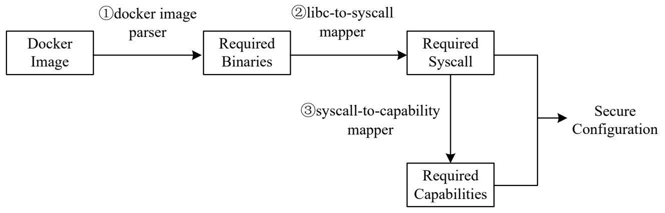


Fig. 1: The Architecture of SysCap

Dockerfile of the target Docker image to rebuild the linear relation between different layers that are stored disorderly in a Docker image and infer the functionality of each layer automatically. Then, we extract the called functions in the binaries obtained from the target layer in the Docker image.

The libc-to-syscall mapper builds a mapping table between libc functions and their corresponding system calls (only once for a specific libc version) and then transforms the called functions obtained from *docker image parser* into required system calls. To construct calling relations between libc functions and corresponding system calls, this module solves two challenges: accurate system call identification and complete calling relation construction in libc. Since the ways to pass the system call number to registers vary, we propose a method called *Register Value Backtracking* to accurately track the system call numbers. After identifying all system calls, we develop an efficient algorithm to obtain a complete calling relation from libc functions to their corresponding system calls since the number of libc functions is large and the calling relations are complex. Finally, using the libc-to-syscall mapping table, we transform the called functions obtained from *docker image parser* into required system calls.

The syscall-to-capability mapper is responsible for generating a mapping table between system calls and their corresponding capabilities. To achieve this, the main challenge is to identify all positions that check for capabilities in the Linux kernel and to execute system call reachability analysis. In our solution, we first compile the Linux kernel into a single-file LLVM bytecode using `wllvm` [11] and identify capability checking functions that take a specific capability as a parameter. Then, we check all positions that call these capability checking functions and mark the corresponding capabilities. Next, we execute the system call reachability analysis, which checks whether there exists a path from system call entries to these capability checking positions. Finally, using the syscall-to-capability mapping table, we obtain the required capabilities via transforming the invoked system calls obtained from the *libc-to-syscall mapper*.

IV. Docker Image Parser

The docker image parser extracts the called functions in the binaries obtained from the target layer of the Docker image. First, it obtains the layer relations of the image via analyzing the configuration file. Second, it obtains the semantic of each layer by comparing the layer relations with the Dockerfile. Finally, it locates the target layer in the image and obtains all binaries in the target layer.

A. Layer Relation Construction

Since the organizational structure of a Docker image is complicated and layers in a Docker image are stored disorderly, we cannot directly build the relation between layers from the original image. Fortunately, we can extract all layers of a Docker image by applying the `docker save` command and each layer is within a folder, containing a `json` configuration file. As each `json` file contains the current layer ID and its parent layer ID, i.e., hash values, the relation between different layers can be constructed.

Table I shows an example on constructing the layer relation for the `redis` image. The image contains six layers in total. Starting from layer `6102` whose in-degree is `0`, we know its parent layer is `ae36`. Similarly, we know that the parent layer of `ae36` is `d11c`. By analyzing all the layer relations recorded in the `json` files, we get the chain relation for the six layers. Note that no matter what the Docker image is, we always obtain a one-way and acyclic chain relation between different layers due to the hierarchical structure between layers. Hence, the layer with an in-degree of 0 is always the start layer, and the layer with an out-degree of 0 is the end layer.

TABLE I: Dockerfile commands and layer dependency construction for the `redis` image.

Dockerfile Commands		**Layer**	**Parent**
FROM	debian:buster-slim	`9297`	`null`
RUN	groupadd -r -g 999 redis && useradd -r -g redis -u 999 redis	`1a7d`	`9297`
RUN	wget -O /usr/local/bin/gosu "/download/$GOSU_VERSION"; \	`afac`	`1a7d`
RUN	wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL"; \ ...	`d11c`	`afac`
RUN	mkdir /data && chown redis:redis /data	`ae36`	`d11c`
COPY	docker-entrypoint.sh /usr/local/bin	`6102`	`ae36`

B. Layer Function Extraction

Due to the multiple files in each layer, it is time-consuming and difficult to manually analyze the main function of each layer. Fortunately, a text document named Dockerfile [13] records how each layer is constructed from Dockerfile commands, showing the main function of each layer. For example, Table I shows a Dockerfile of the `redis` image that consists of six Dockerfile commands with all capital letters, e.g., `FROM` and `RUN`. According to the building process of a Docker image, each building command forms a layer in the image one after another. After knowing the chain relation between the six layers, we can establish a one-to-one correspondence between the Dockerfile commands and the image layers. For instance, the first image layer named `9297` builds a Debian operating system with the buster-slim version, and the second image layer named `1a7d` adds a user called `redis` to a group.

Since a Dockerfile command can span multiple lines, we can split those commands based on two observations: (1) the Dockerfile commands are all at the beginning of a line, and other general commands follows them; (2) all the letter in the keywords of Dockerfile commands are capitalized, and other general commands are usually lowercase. By scanning the Dockerfile line by line, we can split Dockerfile commands and build a one-to-one correspondence between Dockerfile commands and image layers. Thus, the main function of each layer can be known from related Dockerfile commands.

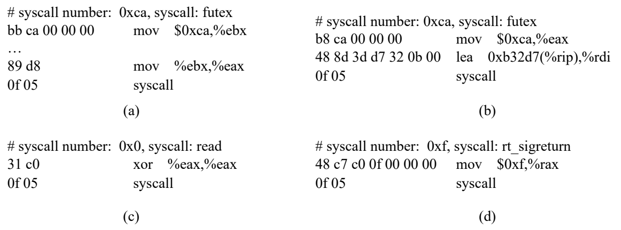
After building the correspondence between Dockerfile commands and image layers, we can locate the target layer in the image and obtain all binaries in this layer. Since the Docker image uses the UnionFS file system, each layer is relatively independent and has complete files to execute its function. Hence, we only need to extract all binaries located in the target layer which realizes the core function of an image. Since we have obtained the function of each layer, it is easy to identify the target layer from all layers. Typically, the target layer installs the target application with the `RUN` Dockerfile command. As shown in Table I, the target layer is layer `d11c`, which installs the `redis` application in the `redis` image using the `RUN` command.

V. Libc-to-Syscall Mapper

The libc-to-syscall mapper analyzes the library to construct a mapping table between libc functions and their corresponding system calls. We first solve the difficulty of identifying system calls in a library using *Register Value Backtracking*. Next, we propose an efficient algorithm to improve the performance when constructing the calling graph for libc.

A. Syscall Identification

We need to identify which system calls, i.e., system call numbers, are invoked of libc functions. A system call is usually invoked in two steps: (1) a system call number is passed to the register `eax` or `rax`; (2) a system call instruction (i.e., `syscall`) is called to complete related kernel functions. However, there are various ways to pass system call numbers to registers. Fig. 2 shows several typical ways of passing system call numbers.



```
# syscall number:  0xca, syscall: futex
bb ca 00 00 00        mov    $0xca,%ebx
…
89 d8                 mov    %ebx,%eax
0f 05                 syscall
```

(a)

```
# syscall number: 0xca, syscall: futex
b8 ca 00 00 00           mov    $0xca,%eax
48 8d 3d d7 32 0b 00     lea    0xb32d7(%rip),%rdi
0f 05                    syscall
```

(b)

```
# syscall number:  0x0, syscall: read
31 c0                 xor    %eax,%eax
0f 05                 syscall
```

(c)

```
# syscall number:  0xf, syscall: rt_sigreturn
48 c7 c0 0f 00 00 00     mov    $0xf,%rax
0f 05                    syscall
```

(d)

Fig. 2: Different Ways for Passing System Call Numbers

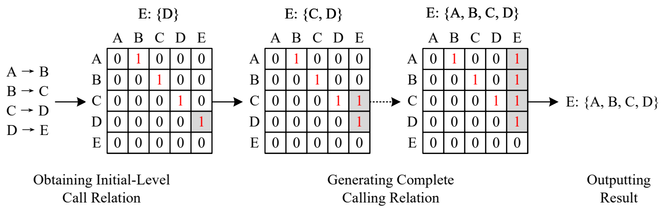
To accurately identify the required system calls of libc functions, we develop a method called Register Value Backtracking. We first disassemble the library into the assembly code using the disassembly tool `objdump`. Next, we split the assembly code into pieces according to the system call invoking instruction `syscall`. Finally, we backtrack the position of register `eax` or `rax` from the `syscall` instruction to obtain correct syscall numbers. For patterns in Fig. 2(b) and Fig. 2(d), if a constant is passed to `eax` or `rax`, we keep this number and continue backtracking another system call. If there is another register when tracking back to `eax`, we backtrack the value of this register until we get the final syscall number, for pattern in Fig. 2(a). If there is a numerical calculation like `xor`, we calculate this result and assign it to `eax`, solving the problem in Fig. 2(c).

We change obtained system call numbers to corresponding system call names with a system file named `unistd_64.h` that records the correspondence between a system call number and its name. To help construct the mapping table, we insert these system call names into original assembly codes with special marks. For example, we mark `syscall read` in the position where the `read` system call is invoked in the assembly codes.

B. Syscall Mapping Table Construction

We observe that most libc functions call other libc functions to invoke system calls, instead of invoking system calls directly. Also, the calling relation among libc functions can be complex and may generate calling loopbacks. To model the calling relation between functions, we develop an efficient algorithm to generate a calling graph for libc. By analyzing the nodes in the calling graph, we know what system calls are invoked by a libc function.

Building a calling graph requires the direct calling relation between functions, i.e., directed edges in the graph. Since function calling has a fixed format in assembly code, when we disassemble libc into assemble code, each caller function is included inside '⟨' and '⟩', followed by a ':' symbol and each callee function exists in the implementation of the caller function and inside '⟨' and '⟩'. Thus, to obtain the calling relation, we scan the assembly code with '⟨', '⟩', ':', and syscall marks as keywords. If '⟨' and '⟩' are followed by ':', the function inside '⟨' and '⟩' is the caller function. From the line of the caller function to the line of the next caller function, all the functions and the system calls between the two lines are the callee functions. By scanning all the assembly codes, we can get the direct calling relations of all functions.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 0 & 0 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array} \quad \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 0 & 0 \\ C & 0 & 0 & 0 & 1 & 1 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array} \quad \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 1 \\ B & 0 & 0 & 1 & 0 & 1 \\ C & 0 & 0 & 0 & 1 & 1 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

Fig. 3: An Example of Building Function Call Relation

Due to the large number and the complex calling relation, the traditional algorithms to check if there are indirect calling relations between two function nodes will suffer from huge performance overhead. For example, to solve the *transitive closure* problem in the directed graph, the time complexity of the classic Floyd–Warshall algorithm [14] is O(n^3). To solve this problem, we propose an efficient algorithm to extract what system calls are invoked for each function in libc. Specifically, we number all libc functions and system calls using consecutive integers (the sequence numbers of all system calls will be in a centralized range due to special marks), and apply a Boolean adjacency matrix to represent the calling relations. Since what we want to know is the calling relation from libc functions to the corresponding system calls, intermediate nodes which chain the calling can be ignored and only the callings to system calls will be focused.

We use an example to illustrate how we find calling relations from libc functions to the corresponding system calls. As shown in Fig. 3, A, B, C, and D are libc functions and E is a system call. Function A calls function B, which calls function C, which calls function D, and finally function D invokes system call E. Based on the initial calling relation, the corresponding bits are set to `1` in the Boolean adjacency matrix. Since only the function D can invoke system call E, the reachable set of system call E has an element of D. Later, any function calling functions in the reachable set will be added. Repeat this step until all reachable functions are added to this set. Finally, the complete calling relations can be obtained in the final adjacency matrix from libc functions to the corresponding system calls. The detailed algorithm is shown in Algorithm 1.

Algorithm 1 Building the Mapping Table

```
Input: A pre-processed assembly code of libc
Output: A mapping table from libc functions to corresponding system calls
 1: // Relation Initialization
 2: number all functions and system calls using consecutive integers (total $N$)
 3: initialize the Boolean adjacency matrix ($N * N$) to 0
 4: for node $i \in N$ do
 5:     for node $j \in N$ do
 6:         if existing calling relation from node $i$ to $j$ then
 7:             $matrix[i][j]$ = 1
 8:         end if
 9:     end for
10: end for
11: // Complete Calling Relation Generation
12: for node $j \in$ number scope of system calls do
13:     for node $i \in N$ do
14:         if $matrix[i][j] == 1$ then
15:             add $i$ to reachable set of system call $j$
16:             add all functions that call functions in the reachable set
17:         end if
18:     end for
19: end for
20: // Calling Relation Filtering
21: for node $i \in N$ do
22:     for node $j \in$ number scope of system calls do
23:         if $matrix[i][j] == 1$ then
24:             output function $i$: syscall $j$
25:         end if
26:     end for
27: end for
```

C. System Calls Extraction

We extract the required system calls for the binaries obtained from the Docker image. Since programs invoke system calls by calling the libc functions, we first identify function names in target binaries and then obtain the corresponding system calls from the constructed libc-to-syscall mapping table. We generate binary symbol tables, containing symbol types and names, using the `readelf` tool to help identify function names in target binaries. After comparing the obtained functions with the libc-to-syscall mapping table, we can filter out libc functions and their corresponding system calls.

When the binaries leverage a third-party library that calls libc functions to invoke system calls, we cannot directly obtain the invoked system calls via the libc-to-syscall mapping. To solve this problem, we apply the `ldd` tool to obtain all dependency libraries of the binary and then build a mapping for each library. The building procedure is similar to that in Section V. In a few cases where the binary may directly invoke system calls via passing system call numbers to `eax` or `rax` instead of calling libc functions, we extract these system calls using *Register Value Backtracking* method described in Section V-A. Therefore, all invoked system calls can be extracted no matter if they are directly invoked by passing constants to the specific registers or indirectly invoked by calling libc functions. There are some system calls that are used for creating the running environment only and cannot be obtained from profiling a Docker image statically. To obtain these system calls, we monitor the startup process of different containers and dump system logs that record the required system calls in this phase. We find that the startup of Docker containers is mainly done by the `runc` process and such system calls are fixed when specifying a `runc` version, as shown in Table II.

TABLE II: Required syscalls to create container running environment.

Functionality	System Call
ID	getppid, setgid, setgroups, setuid
File	chdir, close, epoll_ctl, epoll_pwait, fchown, fcntl, fstat, fstatfs, getdents64, newfstatat, openat, read, write
Process	execve, prctl, sched_yield
System	futex, nanosleep
Capability	capget, capset

VI. Syscall-to-Capability Mapper

Customizing required system calls for an image can enhance the security; however, some system calls will require specific capabilities to guarantee their normal running. The syscall-to-capability mapper is responsible for generating a mapping table between system calls and their required capabilities.

To obtain syscall-to-capability mapping table, we first compile the Linux kernel into one LLVM bytecode and translate the bytecode into LLVM IR using `wllvm` [11]. After resolving indirect calls using struct-type [12], over 4 million lines of IR codes are generated, in which detailed implementation of system calls and kernel functions are included. Then we identify capability checking functions that take a specific capability as a parameter. There are 3 security functions used for capability checking, i.e., `capable`, `ns_capable`, and `avc_has_perm_noaudit` [12]. When analyzing the kernel IR code, we extend the capability checking functions to 34, in which one of the parameters is `i32 %cap`. For instance, the function `has_capability` passes two parameters: `%struct.task_struct* %t` and `i32 %cap`. We mark function `has_capability` as a capability checking function and the position to check the capability is parameter 2. Table III lists the information of capability checking functions and capability position in parameters.

After identifying all capability checking functions and capability positions in parameters of these functions, we need to know which kernel functions call these capability checking functions and which capabilities are checked. Thus, we

TABLE III: Capability checking functions and position for capability in parameters.

Function	Pos	Function	Pos	Function	Pos
capable	1	security_capable	3	has_capability_noaudit	2
ns_capable	2	amd_iommu_capable	1	pci_find_ext_capability	2
sk_capable	2	ns_capable_noaudit	2	pci_bus_find_capability	3
cap_capable	3	netlink_ns_capable	3	security_capable_noaudit	3
file_ns_capable	3	intel_iommu_capable	1	pci_find_next_capability	3
has_capability	2	netlink_net_capable	2	capable_wrt_inode_uidgid	2
sk_ns_capable	3	sk_filter_trim_cap	3	has_ns_capability_noaudit	3
sk_net_capable	2	cred_has_capability	2	snd_seq_event_port_attach	3
netlink_capable	2	pci_find_capability	2	snd_hda_override_amp_caps	4
selinux_capable	3	__vm_enough_memory	3	ieee80211_ie_build_vht_cap	3
iommu_capable	2	__netlink_ns_capable	3	pci_find_next_ext_capability	3
has_ns_capability	3				

take the capability checking functions as the keywords and the capability positions as the matching patterns to locate all positions that check capabilities and mark these capabilities specially, e.g., mark `CAPABILITY_0` for capability `CAP_CHOWN`. In the implementation of system call `setgid`, function `ns_capable` is called with parameter `6` in the capability position, thus we mark this function calling statement as `CAPABILITY_6`. By inquiring the official document [15], we see capability `6` corresponds to `CAP_SETGID`, allowing system call `setgid` manipulation.

With the identified capability checking positions, we propose system call reachability analysis to ensure these capability checking positions can actually be reached from a specific system call. Our idea of system call reachability analysis is two-fold. First, we process the kernel IR code into a specific format, which means keeping function names and special capability marks only. To achieve this, we use the regular expression and script to delete all other information. Second, we perform control-flow analysis based on the kernel control flow graph. If there is a path from a specific system call to the capability checking positions, we keep this correspondence to the syscall-to-capability mapping table. After comparing the invoked system calls obtained from the libc-to-syscall mapper with the syscall-to-capability mapping table, the required capabilities can be extracted and a secure configuration for a specific container is generated, including the customized whitelists for both system call and capability.

VII. Implementation

We implement the docker image parser with Shell scripts. First, we extract all layers of the image using `docker save` command and obtain the relations of all layers based on parsing configuration file in each layer. By comparing the Dockerfile with all layers in the image, we obtain the function of each layer and then extract all binaries in the target layer.

We implement the libc-to-syscall mapper with C/C++ and Python. We first disassemble libc into assembly code and preprocess it to delete function-irrelated codes. Then we split the preprocessed code into pieces according to system call invoking instructions. Furthermore, we use the Register Value Backtracking method to obtain the system call numbers. By comparing system call numbers with a system call table in `unistd_64.h`, we obtain the corresponding system call names. Next, we output the libc-to-syscall mapping table based on Alg. 1. Finally, we compare the called functions obtained in the binaries obtained from docker image parser with libc-to-syscall mapping table to obtain the required system calls. To get the system call list for container startup, we set seccomp filter as the logging mode with the `SCMP_ACT_LOG` parameter to record all system calls filtered by seccomp.

Applications in containers may be constructed by different programming languages and most of these languages rely on libc to implement the system call invoking. We can apply the methods in Section V to parse all dependency libraries and build a mapping table for each library. Then, the required system calls can be obtained by chaining all mapping tables until libc-to-syscall mapping. However, for Go, it has its own way to invoke system calls by calling `syscall.Syscall` and `syscall.RawSyscall` functions. Through observation, for a go binary, the system call is invoked by passing a constant to `rsp` register, e.g., "`mov 0x27, (rsp)`" to invoke `getpid` system call, and then calling system call invoking functions. Thus, we obtain the invoked system calls for go applications by matching this fixed pattern.

We write a pass based on LLVM to implement the syscall-to-capability mapper. First, we compile the Linux kernel into an LLVM bytecode using `wllvm` [11] and resolve indirect calls based on struct-type [12]. Then we extend capability checking functions and locate the capability location of parameters in these functions to identify all capability checking positions and their corresponding capabilities. Next, we delete all function-unrelated IR codes and generate a kernel control flow graph. Finally, we perform system call reachability analysis and generate the syscall-to-capability mapping table. After comparing the required system calls for a Docker image with the syscall-to-capability mapping table, we obtain the capabilities that should be enabled when running the container.

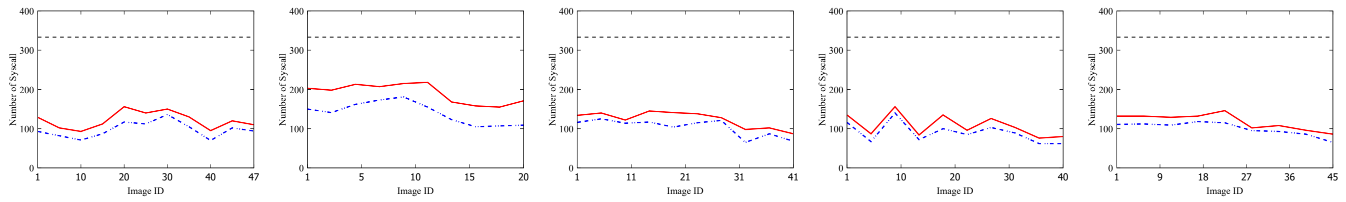
VIII. Experimental Results

A. Experiment Setup

We conduct experiments on a computer with an Intel i5-8265U 1.6GHz 4-core processor and 8GB of RAM. It runs Ubuntu 18.04 with Linux kernel 4.10, GNU C Library (glibc) 2.29, and Docker 20.10.2. To facilitate the comparison with previous works, we adopt the same container image dataset used by the Confine [9]. It consists of 193 popular Docker official images pulled from the Docker Hub [16], whose downloads reached over 10 million. These images can be generally classified into 5 categories in terms of their functionality, i.e., database, OS, DevOps, language, and infrastructure.

B. System Call Reduction

Fig. 4 compares the system call whitelist obtained via SysCap and dynamic tracking method over 5 categories of Docker images. We can see that for each tested container, the system call whitelist obtained by SysCap is much smaller than the default list and contains the system call whitelist



(a) Database (b) OS (c) Infrastructure (d) Language (e) DevOps

Fig. 4: Numbers of Required Syscall. The **dashed** gray line is the default setting, the **solid** red line is SysCap, and the **dotted** blue line is dynamic tracking.

by the dynamic tracking method. Moreover, the difference between SysCap and dynamic tracking method is small for four categories, i.e., Database, Infrastructure, Language, and DevOps. It shows that our static analysis solution can achieve completeness without sacrificing too much soundness. For the OS category, the larger difference is because OS containers provide many general commands and programs for all applications, but many commands and programs may not be executed during the dynamic tracking process.

Overall, dynamic tracking and SysCap finds 105 and 127 system calls on average, respectively. It is reasonable to finds more system calls by static analysis techniques than the dynamic tracking techniques. When performing dynamically tracking, it is challenging to cover all branches for containers even if with the assistance of the fuzzing technique [17]. In contrast, our static analysis work can ensure a complete whitelist that includes around 20 system calls on average more than the dynamic tracking.

SysCap can reduce up to 78% system calls and 62% system calls on average for all 193 containers (see Fig. 5). The average system call reduction rate for each category is 63.3% (Database), 42.6% (OS), 67.6% (Infrastructure), 62.8% (Language) and 64.6% (DevOps), respectively. Even for the worst OS category, the system call reduction rate can reach about 35% at least and 53% at most. For other four categories, system call reduction rate shares the similar results, ranging from 45% to 78%.

C. Vulnerability Reduction

We explore a set of exploitable vulnerabilities that can be mitigated by leveraging the system call whitelist generated by SysCap. We collect all vulnerabilities from Common Vulnerabilities and Exposures (CVE) [18] in recent six years, which leverage specific system calls to launch attacks.

We define the vulnerability reduction rate for a container as the number of vulnerabilities that can be disabled by eliminating unnecessary system calls with SysCap to the total number of vulnerabilities. As shown in Fig. 6, we summarize the vulnerability reduction rate for tested container dataset, obtained via limiting the system calls that may trigger the vulnerability in containers. From the figure, we can see that almost all the vulnerabilities reduction rate are stable between 60% and 80%, which means that by setting the system call whitelist for the containers, more than 60% of the software vulnerabilities can be directly blocked. Specifically, SysCap can achieve more than 55% vulnerability reduction rate for all these containers, so over half number of existing vulnerabilities can be prevented. The vulnerability reduction rate for 70% OS-related containers can reach to at least 60%. Moreover, SysCap performs the best for containers in the language category, i.e., more than 80% vulnerability reduction rate for about 40% containers. Indeed, our experimental results prove that disabling unused system calls with SysCap can reduce the attack surface of containers.

D. Comparison with State-of-the-Art work

Existing studies [6]–[8] dynamically profile system calls for containers. However, as demonstrated by Confine [9], they may miss a number of required system calls. Hence, we compared our tool with the state-of-the-art system Confine [9] that combines dynamic and static analysis in aspects of the efficacy and accuracy. Confine extracts required system calls for containers in two steps: (1) it needs to manually configures and run a container in a period to monitor called binaries and libraries; (2) it statically analyzes called binaries and libraries to extract required system calls. However, 43 out of 193 images cannot be analyzed by Confine [9] due to complex configuration settings and prerequisite containers, which denotes a success rate of 77.7%. In contrast, SysCap statically extracts binaries from Docker images and our experiments demonstrate that SysCap can fully automatically extract system calls from all the 193 images, with a 100% success rate.

We further compare the number of system calls filtered by SysCap and Confine [9] on the top 15 most popular official images. As shown in Table IV, we find that SysCap can exclude more unnecessary system calls for 13 out of 15 images. The only exception is on OS-related images (i.e., Ubuntu and CentOS) where Confine filters more system calls. We figure out two main reasons. First, in the source code of Confine [19], they maintain a system call blacklist and use two except lists (i.e., `exceptList` and `javaExceptList` [20]) for all containers. Since these two except lists contains 87 system calls, no matter what containers they run, its required system call list would have at least 87 system calls. However, Confine does not prove the necessity of these 87 system calls. When we implement SysCap, we extract the required system call based on the target binaries. In this way, we can obtain a more precise system call list that filters more unnecessary system calls.

Second, Confine may miss some system calls since it only records the called binaries and libraries in a fixed running

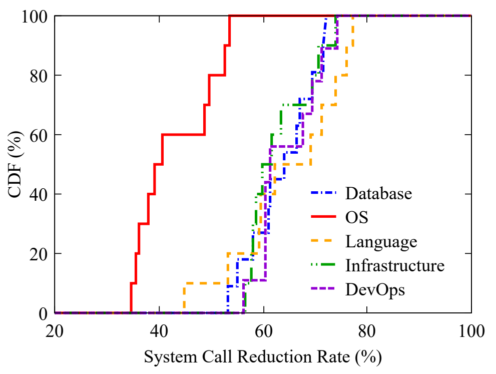


Fig. 5: CDF of System Call Reduction

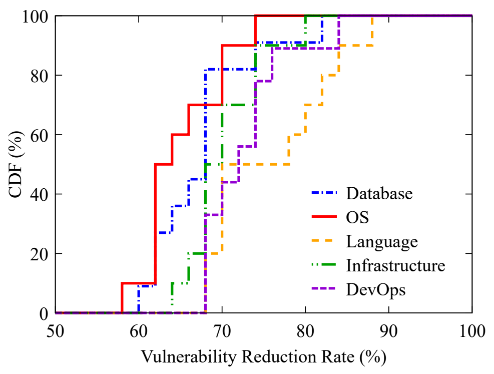


Fig. 6: CDF of Vulnerability Reduction

TABLE IV: Comparison Between SysCap and Confine [9] on number of filtered system calls and vulnerability reduction rate for the top 15 most downloaded images.

Image*	Filtered System Calls		Vulnerability Reduction	
	Confine	SysCap	Confine	SysCap
couchdb	157	240 (+83)	50%	68% (+18%)
ubuntu	198	135 (-63)	68%	66% (-2%)
alpine	162	165 (+3)	65%	70% (+5%)
redis	179	223 (+44)	58%	62% (+4%)
node	170	192 (+22)	62%	64% (+2%)
postgres	141	213 (+72)	52%	66% (+14%)
mysql	149	183 (+34)	60%	64% (+4%)
nginx	177	188 (+11)	67%	68% (+1%)
mongo	152	193 (+41)	53%	60% (+7%)
python	154	177 (+23)	66%	70% (+4%)
traefik	211	246 (+35)	59%	66% (+7%)
mariadb	139	177 (+38)	62%	68% (+6%)
httpd	175	199 (+24)	66%	70% (+4%)
golang	197	198 (+1)	78%	78% (+0%)
centos	199	120 (-79)	76%	74% (-2%)

*Images were ranked based on total downloads as of May 27, 2021.

period of containers, especially for OS-related ones. In the operating system, although there are hundreds of general commands, only about ten commands would be executed during startup period. Failing to consider binaries executed during the running time will lead to an incomplete invoked system call list, which disables some normal functionalities. For instance, in `ubuntu` container, a user can create a directory using `mkdir` command, which invokes `mkdir` system call. However, when Confine customizes the required system calls for `ubuntu`, since the `mkdir` command does not run during startup period, Confine blocks the `mkdir` system call [21], causing users to be unable to use the `mkdir` command.

E. Capability Mapping

We conduct experiments to verify the correctness of the syscall-to-capability mapping table, which consist of comparing it with official description and running customized system call and capability list for a specific container.

We first compare the syscall-to-capability mapping table with the official document [15], which describes the functionality and possible system calls for each capability. For example, in the description of capability `CAP_SETGID`, it allows system call `setgid` and `setgroups` manipulation, and it allows gids on socket credentials passing. Meanwhile, in the syscall-to-capability mapping table, the required system calls for capability `CAP_SETGID` are `setfsgid`, `setgid`, `setgroups`, `setregid`, and `setresgid`. By comparing the official description and the syscall-to-capability mapping table for capability `CAP_SETGID`, we can know that our mapping result is reasonable and we can help make official descriptions clear. Similarly, we compare the syscall-to-capability mapping table with the official description for each capability, and the result shows the correctness of our mapping.

Next, we run all containers in the image dataset with generated system call and capability list (obtained from comparing the generated system calls with the syscall-to-capability mapping table), all containers run normally under artificial loads. For example, for `hello-world` container, it requires `26` system calls for normal running and requires `0` capabilities; and for `redis` container, it invokes `91` system calls and requires `3` capabilities, i.e., `CAP_SETGID`, `CAP_SETUID`, and `CAP_SYS_RESOURCE`. In general, the required capabilities for all containers range from `0` to `16`, and 78.5% of containers require less than 7 capabilities. On average, 5.3 capabilities are required and about 4% of containers require more than 14 capabilities. The experiment shows that the attack surface can be further reduced with the help of capability customization.

IX. Related Work

System Call Profiling for Containers. Removing unnecessary system calls can significantly reduce the attack surface for containers. Wan et al. [6] dynamically profile invoked system calls for containers with different workloads. Lei et al. [7] split the container running process into phases and dynamically track invoked system calls. Rastogi et al. [8] divide a container into multiple isolated sub-containers and dynamically profile required system calls. As dynamic analysis cannot guarantee completeness for all required system calls, Ghavamnia et al. [9] combine dynamic and static analysis by dynamically launching a container to obtain its running binaries and libraries and statically analyzing the required system calls from the extracted binaries. Different from existing studies, SysCap statically profiles container system calls directly from images without dynamically running containers.

System Call Profiling for Programs. Besides profiling system calls for containers, most prior studies focus on profiling system calls for programs. Zeng et al. [22] construct a stateless model to generate a system call whitelist. Provos et al. [23] design system call policies to run with different levels of privilege. Ghavamnia et al. [24] manually divide the program running state into two phases and statically profiles required system calls. DeMarinis et al. [25] statically extract required system calls from binaries and restricts access by rewriting binaries. These studies effectively profile system calls from programs and possibly inspires profiling system calls for containers. However, they fail to consider the complex structures in container, thus, we provide SysCap to automatically parse Docker images to extract the target binaries.

Traditonal Program Analysis. Static program analysis can generate data flows and control flows for programs, used to detect bugs and vulnerabilities [26]–[28], validate patches [29], and reduce unnecessary path access in programs [30]. Dynamic program analysis is used to monitor and slice programs [31], [32], locate bugs in programs [33]–[35], and perform code coverage fuzz testing [36], [37]. We apply static analysis in SysCap to automatically obtain the required system calls and capabilities for a specific container.

X. Conclusion

In this paper, we design SysCap to automatically generate required system calls and capabilities for Docker images. It statically analyzes libc to build a mapping between libc functions and their corresponding system calls and analyzes the Linux kernel to construct a mapping between system calls and the related capabilities. Given a Docker image, it extracts the called functions of the binaries in the target layer and compares them with these two mappings to obtain the required system calls and capabilities. Our experiments with 193 popular images demonstrate that SysCap can filter 62% unnecessary system calls and reduce more than a half default capabilities on average, reducing the attack surface hugely.

References

[1] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma *et al.*, "Demons in the shared kernel: Abstract resource attacks against os-level virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[2] "capability," https://man7.org/linux/man-pages/man7/capabilities.7.html.

[3] "Seccomp," https://man7.org/linux/man-pages/man2/seccomp.2.html.

[4] "DefCap," https://github.com/moby/blob/master/oci/caps/defaults.go.

[5] "Seccomp profiles," https://docs.docker.com/engine/security/seccomp/.

[6] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *IEEE ICST*, 2017.

[7] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 230–251.

[8] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[9] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020.

[10] "Dockerslim," https://github.com/docker-slim/docker-slim.

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

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

[13] "Dockerfile Reference," https://docs.docker.com/engine/reference/builder.

[14] "Transitive closure," https://en.wikipedia.org/wiki/Transitive_closure.

[15] "capability definition," https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/capability.h#L146.

[16] "Docker Hub," https://hub.docker.com/.

[17] "Fuzzing," https://en.wikipedia.org/wiki/Fuzzing.

[18] "Common Vulnerabilities and Exposure," https://cve.mitre.org/.

[19] "Confine source code," https://github.com/shamedgh/confine.

[20] "The core implementation of confine source code," https://github.com/shamedgh/confine/blob/master/containerProfiler.py.

[21] "Confine customizes the system call blacklist for `ubuntu` image," https://github.com/shamedgh/confine/blob/master/sample.results/ubuntu.seccomp.json.

[22] Q. Zeng, Z. Xin, D. Wu, P. Liu, and B. Mao, "Tailored application-specific system call tables," *The Pennsylvania State University*, 2014.

[23] N. Provos, "Improving host security with system call policies." in *USENIX Security Symposium*, 2003, pp. 257–272.

[24] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[25] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 459–474.

[26] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[27] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015.

[28] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, 2016.

[29] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 49–64.

[30] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 619–629.

[31] B.-A. Stoica, S. K. Sahoo, J. R. Larus, and V. S. Adve, "Wok: statistical program slicing in production," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 324–325.

[32] M. Ward and H. Zedan, "The formal semantics of program slicing for nonterminating computations," *Journal of Software: Evolution and Process*, 2017.

[33] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, pp. 707–740, 2016.

[34] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[35] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[36] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*, 2019.

[37] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, 2014.