Computer Systems Science and Engineering

Tech Science Press

DOI: 10.32604/csse.2024.046615

ARTICLE

FFRA: A Fine-Grained Function-Level Framework to Reduce the Attack Surface

Xingxing Zhang[1], Liang Liu[1,*], Yu Fan[1] and Qian Zhou[2]

[1]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

[2]School of Modern Posts, Nanjing University of Posts and Telecommunications, Nanjing, China

*Corresponding Author: Liang Liu. Email: liuliang@nuaa.edu.cn

Received: 09 October 2023 Accepted: 28 February 2024 Published: 17 July 2024

ABSTRACT

System calls are essential interfaces that enable applications to access and utilize the operating system's services and resources. Attackers frequently exploit application's vulnerabilities and misuse system calls to execute malicious code, aiming to elevate privileges and so on. Consequently, restricting the misuse of system calls becomes a crucial measure in ensuring system security. It is an effective method known as reducing the attack surface. Existing attack surface reduction techniques construct a global whitelist of system calls for the entire lifetime of the application, which is coarse-grained. In this paper, we propose a Fine-grained Function-level framework to Reduce the Attack surface (FFRA). FFRA employs software static analysis to obtain the function call graph of the application. Combining the graph with a mapping of library functions generates each function's legitimate system calls. As far as we know, it is the first approach to construct the whitelist of system calls for each function of the application. We have implemented a prototype of FFRA and evaluated its effectiveness with six popular server applications. The experimental results show that it disables 33% more system calls compared to existing approaches while detecting 15% more shellcode vulnerabilities. Our framework outperforms existing models by defending against a broader range of attacks. Integrated into antivirus software and intrusion prevention systems, FFRA could effectively counter malware by precisely restricting system calls.

KEYWORDS

Software security; attack surface reduction; system call restriction

1 Introduction

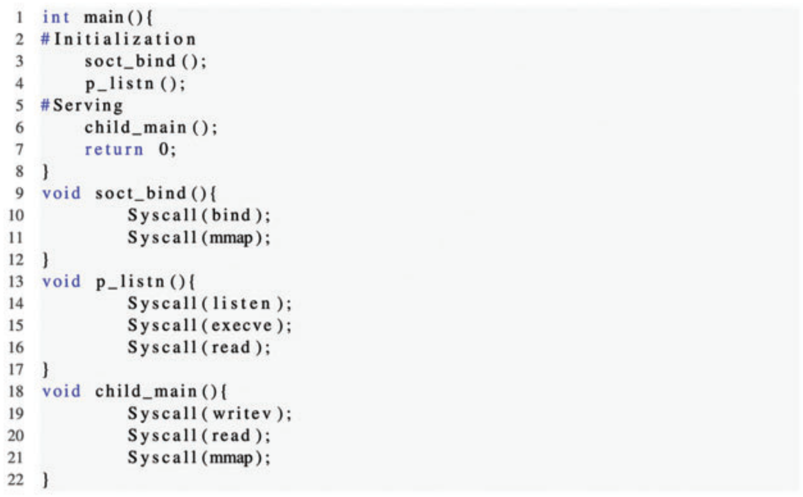
System calls play a vital role in ensuring the security of software and operating systems [1,2], They act as the exclusive bridge that enables applications to access essential services and resources provided by the operating system. However, this also makes them a target for security vulnerabilities, as attackers often exploit system calls to execute malicious code [3].

To mitigate these risks, extensive research has focused on limiting system call usage. This approach, known as reducing the attack surface, is essential for system security. By limiting the legal system calls that an application can use, the attack surface is reduced. Specifically, software debloating and specialization [4] identify and eliminate code unused during the application's entire lifetime. In a distinct approach, Temporal Specialization [5] (TEMP) divides an application into initialization and serving phases, constructing two whitelists of system calls for the above phases, respectively.

While the aforementioned methods have significantly limited the number of legal system calls, they remain too coarse-grained. This approach has inherent limitations. Firstly, it often fails to accurately distinguish between legitimate and malicious uses of the same system calls, which can lead to either overly restrictive controls that hinder legitimate functionalities or insufficiently stringent measures that leave security gaps. Coarse-grained methods lack the granularity to tailor security controls to the specific behavior and needs of individual software components. In this paper, we propose a fine-grained function-level framework to reduce the attack surface. Our approach offers a fresh perspective on system call restrictions. Rather than considering the entire application lifecycle, we establish a legal set of system calls for each function of the application.

FFRA represents a shift toward a more fine-grained approach in system call restriction. Unlike the coarse-grained methods, FFRA operates at the function level within applications. It employs a combination of static and dynamic analyses to identify the minimal, necessary system calls for each specific function. This precision allows FFRA to provide more targeted and effective security controls. By focusing on the function level, FFRA can allow or restrict system calls in a way that aligns closely with the actual needs and behaviors of the application, thereby reducing false positives and false negatives. Furthermore, the fine-grained nature of FFRA enables more adaptive and responsive security measures. It can adjust to changes within the application, such as updates or modifications, with greater agility. By addressing the specific limitations of coarse-grained methods, FFRA offers a more nuanced, accurate, and effective way of reducing the attack surface through system call restriction.

To illustrate the distinction between the TEMP framework and our proposed FFRA framework, take the Nginx application as an example. Fig. 1 displays a simplified pseudocode of Nginx, which consists of two phases: Initialization and serving. The initialization phase calls soct_bind and p_listn to bind and listen to sockets. The child_main serves as the entry point for the serving phase, handling tasks like buffer allocation and I/O operations.



```
int main(){
#Initialization
    soct_bind();
    p_listn();
#Serving
    child_main();
    return 0;
}
void soct_bind(){
        Syscall(bind);
        Syscall(mmap);
}
void p_listn(){
        Syscall(listen);
        Syscall(execve);
        Syscall(read);
}
void child_main(){
        Syscall(writev);
        Syscall(read);
        Syscall(mmap);
}
```

Figure 1: Simple example

According to the TEMP framework, it constructs two whitelists of system calls for initialization and serving phases, respectively. The whitelist of sys-call set during the serving phase (denoted as SS) includes those called by child_main, SS = {writev, read, mmap}. The whitelist of sys-call set during the initialization phase (denoted as IS) includes those called by all application functions. Thus, IS = {bind, listen, execve, writev, read, mmap}. Consequently, upon entering the serving phase, system calls such as bind, listen and execve are effectively disabled. This restriction narrows the attack surface by limiting the range of system calls that can be exploited by an attacker. FFRA constructs the white list of system calls for each function of the application: WS (soct_bind) = {bind, mmap}, WS (p_listn) = {listen, execve, read}, WS (mk_child) = {writev, read, mmap}, and WS (apr_palloc) = {mmap}. FFRA is a more fine-grained framework compared to TEMP.

Our motivation stems from the fact that the system calls between diverse functions can vary significantly. Some potentially hazardous system calls may only be present in a limited number of functions. If there is no restriction, these hazardous system calls could be exploited by attackers, creating vulnerabilities in functions where they are not legitimately needed. Our fine-grained Function-level Framework effectively resolved this issue.

We implemented a prototype of FFRA and evaluated it with six popular applications (Nginx, Apache httpd, Lighttpd, Bind, Memcached, and Redis). The results show that each function consists of at most 78 system calls on average and FFRA disables 55% more security-critical system calls compared to existing approaches.

The main contributions of this paper are as follows:

- We first explore the topic of reducing the attack surface at the application function layer and present a fine-grained function-level system call constraint model to enhance application protection.
- We evaluated our prototype implementation using six popular applications and a diverse collection of 567 shell codes to demonstrate its effectiveness and efficiency in blocking exploit code.

The rest of this paper is organized as follows. In Section 2, we introduce the related work to software debloating and specialization and syscall-based anomaly detection. The motivation and background are present in Section 3. In Section 4, we introduce the proposed solutions and algorithms in detail. In Section 5, we show the experimental results. We summarize the full paper in Section 7.

2 Related Works

In this section, we mainly present the related works from two aspects: Software debloating and specialization, and system call restriction.

2.1 Software Debloating and Specialization

Software debloating and specialization focus on eliminating or limiting unused components of applications. It can decrease the amount of code and features vulnerable to attackers. Existing algorithms can be divided into two categories: Static analysis, used to pinpoint unused sections of shared libraries, and dynamic analysis, which relies on targeted training to detect unnecessary parts within the application.

2.1.1 Static Analysis

Static analysis plays a pivotal role in software debloating. The study shows that only 5% of libc is used on average across the Ubuntu Desktop environment; the heaviest user, vlc media player, only needed 18%. When an application loads a library, as noted by Quach [4], the entirety of that library's code is mapped into the address space, even if only a single function is needed. This approach uses static analysis to achieve significant reductions in library use within the Ubuntu Desktop environment.

Nibbler [5] identifies and erases unused functions within shared libraries at the binary level, showcasing a practical approach to reducing bloat in scenarios where source code might not be accessible.

Static analysis can lead to false positives, as it lacks context on dynamic execution environments. Additionally, it may miss subtle vulnerabilities, due to its limitations in handling complex software structures. Considering these limitations, it becomes increasingly important to explore dynamic analysis.

2.1.2 Dynamic Analysis

Dynamic analysis addresses code debloating by focusing on unnecessary features for specific users. A study on four PHP applications [6] analyzed server-side code activated by client requests. It found that de-bloating can reduce vulnerabilities and attack surfaces by removing extra external packages and PHP gadgets, thereby highlighting the effectiveness of dynamic analysis in reducing vulnerabilities in PHP applications. This approach demonstrates the potential for targeted debloating in various software environments.

RAZOR [7] employs control-flow heuristics to keep vital code, effectively cutting over 70% of excess in bloated binaries. RAZOR effectively reduces the size of bloated binaries, finely adjusting them to meet specific users' needs, demonstrating the value of personalized binary customization.

Ullah et al. [8] and Mani et al. [9] provide perspectives on securing advanced computational models against emerging threats. These studies underscore the importance of proactive and innovative defenses in protecting sophisticated computational systems.

Mishra et al. [10] and Singh et al. [11] offer insights into the evolving nature of DDoS attacks and defense strategies in cloud and SDN environments. These works highlight the importance of robust defense mechanisms against increasingly sophisticated cyber threats.

Sharma et al. [12] shed light on the unique challenges faced by IoT devices, which often have constrained resources, making them vulnerable to specific attack vectors. This paper's insights are crucial for understanding the security implications in IoT environments.

Although dynamic analysis can find most system calls, it still has some limitations. First, due to the incompleteness of dynamic tracing, tracing is not guaranteed to identify all required system calls. Second, dynamic tracing of different paths of a process requires strict input of construction conditions, in particular, user processes usually call various wrapper functions in the standard library to call system calls, which greatly increases the complexity of tracing.

2.2 System Call Restriction

System call restrictions limit the actions a program can request from an operating system for security purposes. It can prevent unauthorized access and ensure system security.

Wan et al. [13] track applications running in containers and generate corresponding seccomp rules, pioneering the use of dynamic analysis for container security and creating tailored seccomp rules for enhanced protection.

DockerSlim [14] generates seccomp filters and revolutionizes Docker security. It intelligently streamlines Docker images by removing redundant files, significantly enhancing the overall security and efficiency of Docker containers.

Lei et al. [15] innovatively divided container operations into two distinct phases, employing a benchmark tool, HammerDB [16] utilizes dynamic tracking to ensure robust monitoring of system calls, enhancing the safety and efficiency of container operations.

Cimplifier, introduced by Rastogi et al. [17], makes a notable contribution by using dynamic analysis to efficiently split containers running multiple applications into single-purpose containers. Following this, Rastogi et al. [18] further enhanced Cimplifier with the introduction of symbolic execution, which refined its functionality and effectiveness in container management.

Ghavamnia et al. [19] made a significant contribution by proposing a hybrid approach combining dynamic and static analysis for container security. Their method starts with dynamically monitoring binaries within a container and then employs static analysis to compile a comprehensive system call whitelist. This approach addresses the limitation of not capturing all necessary system calls during the container's startup phase, ensuring a more complete and secure container environment.

Brown et al. [20] primarily focused on the impact of software debloating on malware detection. It reveals how reducing the size of malware through debloating techniques can decrease its detectability by antivirus systems. This research underscores the potential security risks of debloating, especially in the context of malware defense. The study's main contribution is in highlighting the balance between software efficiency and cybersecurity risks related to malware detection.

Wang et al. [21] introduces Picup, a method that dynamically customizes software libraries based on specific inputs to mitigate code-reuse attacks. This approach uses a convolutional neural network to predict necessary library functions for each input, significantly reducing the software's code size and enhancing security against attacks. The study's main contribution is to develop a more secure and efficient way of debloating software libraries, thus improving overall software security.

Existing application de-redundancy schemes are mainly based on removing unnecessary code within a single process. Ghavamnia et al. deleted all non-imported functions at program load time. Ghavamnia et al. [22] constructed a function call graph for the binary system based on a Low-Level Virtual Machine (LLVM) and obtained the list of system calls required for each stage according to the phase segmentation points of the function call graph.

These papers collectively contribute to advancing application de-redundancy and security. They focus on removing unnecessary code, optimizing compiler technology, and customizing binary libraries for efficiency. Additionally, they delve into advanced system call management, employing LLVM techniques for detailed function call analysis. Each study introduces innovative methods to enhance software performance and security.

While previous studies have made strides in system call restriction and application de-redundancy, they often employ a global, coarse-grained approach. Our research introduces a significant innovation with the FFRA. This novel method provides a function-specific system called whitelists, offering a more precise and effective attack surface reduction.

3 Background and Motivation

This section discusses the motivation and necessary background for this work.

3.1 Background

In recent years, the exploitation of system calls by malicious software has become a significant security issue. Various studies and reports indicate a substantial increase in malicious activities executed through system calls in the past few years, leading to losses amounting to billions of dollars. Notable instances, such as the WannaCry ransomware attack, demonstrate this trend. In this attack, exploiters leveraged system call vulnerabilities to affect over 200,000 computers worldwide, resulting in millions of dollars in damages. These incidents underscore the urgency of effective system call control and the necessity of more fine-grained approaches to reduce attack surfaces and enhance overall software security.

One of the primary challenges in this domain is the inherent difficulty in differentiating between legitimate and malicious use of system calls. Traditional, coarse-grained methods of system call restriction often lack the sophistication to make this distinction, which can lead to two major pitfalls: Overly restrictive controls that impede legitimate application functionalities, and under-protective measures that leave exploitable vulnerabilities open.

Contrasting with coarse-grained approaches, fine-grained system calls restriction methods offer more targeted and effective security control. By precisely regulating system calls required for each function or module, these methods adeptly identify and mitigate malicious activities while minimizing restrictions on legitimate software functionalities. For instance, in complex applications, fine-grained strategies adapt security policies to specific functional requirements, ensuring operational integrity while effectively reducing potential attack surfaces. The flexibility and precision of this approach are pivotal in countering contemporary, sophisticated attack scenarios, making it an indispensable tool in enhancing software security.

3.2 Motivation

The system calls between diverse functions can vary significantly. The Linux kernel currently provides approximately 345 distinct system calls, but research shows that many pieces of software only utilize a subset of these calls. As a result, numerous studies have analyzed the set of system calls utilized throughout a software's entire lifecycle and restricted the rest. However, the system calls invoked by many functions are vastly different. Assuming that a particular system call is used by only one function, it will be preserved and allowed in all other functions according to application-level methodologies. This, however, leaves potential gaps for attacks.

Therefore, this paper proposes a function-level restriction on system calls. As illustrated in Fig. 2, the execve syscall is only invoked within the function Ngx_process_event. If it is accessible from functions Ngx_init_cycle and Ngx_single_process_cycle, it poses a risk. By restricting the use of execve to function Ngx_process_event only, the security of the software can be significantly enhanced.

3.3 Strace

Strace is frequently used to trace system calls and received signals during the execution of a process. In Linux [23], processes do not have direct access to hardware devices. When a process needs to access hardware devices (such as reading disk files or receiving network data, etc.), it must switch from user mode to kernel mode and then use system calls to access hardware devices. Strace can monitor system calls generated by a process, including their parameters, return values, and execution time.

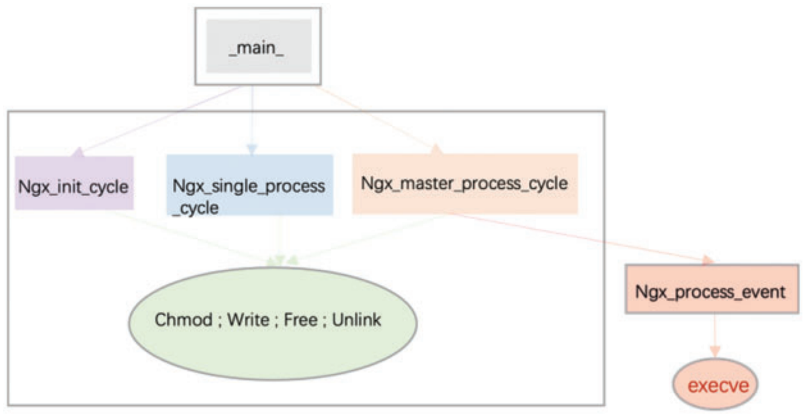


Figure 2: Invocation of system calls

3.4 Instrumentation

Professor Morris first introduced program instrumentation, which consists of inserting probes (also known as "detectors") into the program to guarantee the original logic integrity of the program being tested, i.e., to collect data [24]. The code segment, which can be an assignment statement or a function call to collect coverage information, through the execution of the probe and throws out the characteristic data of the program running, through the analysis of these data, the control flow and data flow information of the program can be obtained, and it is a method for obtaining dynamic information such as logic coverage, to achieve the testing objective.

3.5 Syscall Filtering

Typically, two types of programs run on a computer system: System programs and application programs, as Fig. 3 shows. To prevent application programs from accidentally or intentionally harming the system programs, the computer is configured with two states: System state and user state. The operating system operates in system mode, whereas only applications can operate in user mode. The processor will transition between the system state and the user state during operation. When the application requires services from the operating system, such as requesting I/O resources or performing I/O operations, it must rely on a system call. For programs to do crucial jobs, they must interface with the kernel wholly via the syscall API.

Indicatively, the Linux kernel (v5.5) provides support for 347 syscalls on x86-64 at the time of writing. This amount does not include the syscalls required to run 32-bit x86 programs on a 64-bit kernel or 64-bit processes that conform to the x32 ABI. The OS kernel offers complete and unfettered access to the whole collection of syscalls, even though programs only need access to a portion of the aforementioned API to operate effectively.

Syscall filtering can be employed to constrain a process (i.e., sandbox) such that access to certain syscalls and kernel resources is blocked. Thus, it can also reduce the available attack surface by limiting exposure to potential kernel bugs. In Linux, many different mechanisms can be used to filter syscalls, which we explain in the following.

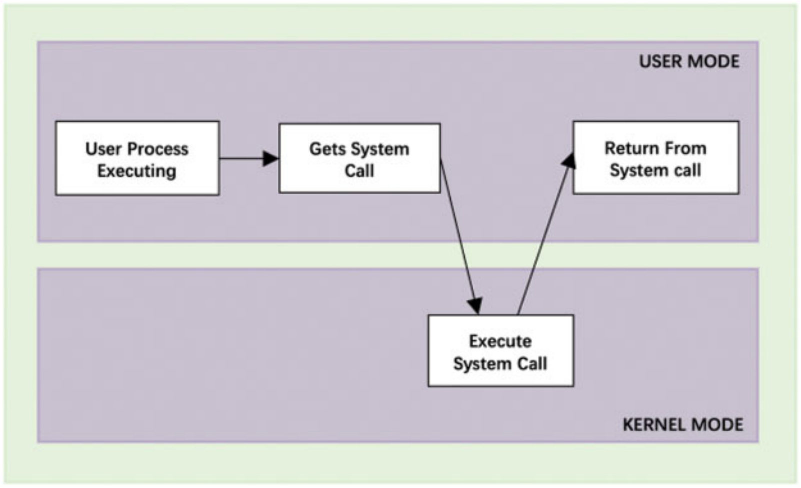


Figure 3: Working of a system call

4 Design and Implementation

Our proposed FFRA aims to curtail the usage of system calls with a heightened degree of precision. Through a combination of static and dynamic analyses, we deduce the minimal set of system calls requisite for each function within the program. This approach allows us to apply constraints with increased accuracy, outperforming traditional methodologies.

FFRA marks a significant departure from traditional coarse-grained methods by focusing on individual functions within applications. This approach is underpinned by a design philosophy that emphasizes targeted analysis and control of system calls at a granular function level. Unlike conventional methods that implement broad, uniform policies applicable to the entire application, FFRA's architecture is tailored to assess and manage system calls specifically for each function. This nuanced approach enables more precise and effective management of system calls, enhancing security without compromising the functionality of the application.

4.1 Overview

The architecture of our system is depicted in Fig. 4, which consists of two main phases. The initial phase employs static analysis to generate a legitimate sys-call set for each application function. Due to the limitations of static analysis, it cannot detect dynamic behavior and may generate false positives, etc. Therefore, we further employ dynamic analysis to compensate for the limitation of static analysis. The combination of dynamic and static analysis increases the precision of the legitimate sys-call set we construct. In the second stage, the legitimate sys-call set generated in the first stage is used as a benchmark to determine if an exception exists during the actual running of the program.

It is challenging to construct the legitimate sys-call set of each application function, because system calls may be invoked directly or indirectly. To meet this challenge, we first perform a static analysis of the application to obtain its function call graph. Not only does this graph contain all required functions and their function call relationships, but it can also map functions to library functions which indirectly invokes the system call.

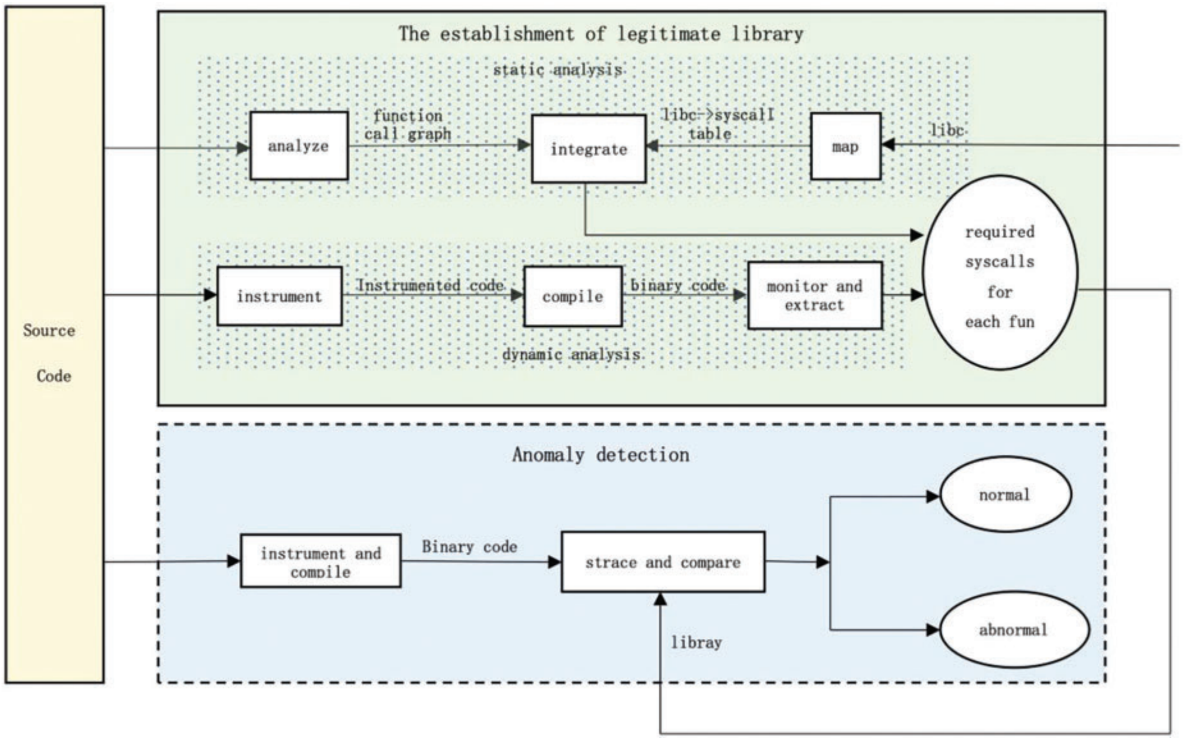


Figure 4: Overview of the architecture of our FFRA framework

Through our proposed Algorithm 1, we can obtain the library functions required by each function. Next, we analyze several popular libraries to obtain the mapping table between library functions and system calls. Finally, we generate the legitimate sys-call set based on the foremost function call graph and the mapping between library functions and system calls.

Even though static analysis is very thorough, some special system calls are called when the program runs dynamically, such as "brk" (which cannot be analyzed by static analysis). If these system calls are disregarded, then when the program runs normally, there will be a lot of false positives. Therefore, we gather system calls called dynamically as a supplement to enhance the accuracy of legitimate sys-call sets.

The second phase involves anomaly detection during the program's actual execution. We are also faced with the challenge of obtaining system calls that are currently being invoked at the application function layer. We employ marking at the entry and exit of application functions to solve the problem and then compare the actual system calls of each application function with the previously established legitimate sys-call sets to identify any possible deviations.

4.2 Legitimate Sys-Call Set Establishment

Constructing the Function-Call Graph, we restrict the use of system calls at the application function layer due to our framework's focus on fine-grained anomaly detection. Therefore, obtaining legal system calls at this layer is of utmost importance. To achieve this, we first obtain the function call graph of the program via SVF which is a static analysis tool. The function call graph offers a visual representation of the relationships between all function calls within an application. Each node in this graph represents a specific function, while edges symbolize the invocation of one function by another.

This graphical construct provides a lucid comprehension of the control flow within the application, essentially the sequence and pathways that govern the execution of the program.

Fig. 5 depicts the corresponding call graph of the above example, there are two types of node shapes in the function call graph: Circle and mrecord, the circle represents the application function, and mrecord represents the library function which generally appears in leaf node, all functions and their required library functions of the program are included.

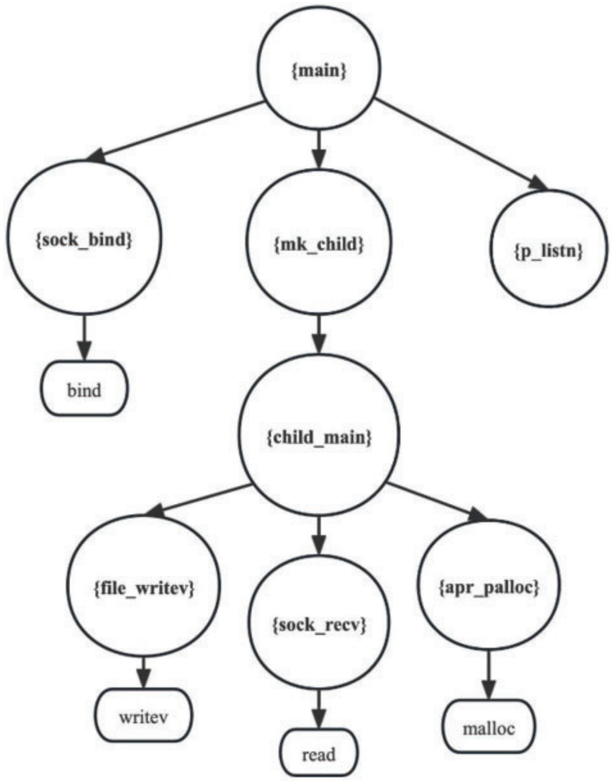


Figure 5: Function call graph

However, the static analysis might still lead to the imprecision of the function-call graph due to potential over-approximations, such as unreachable paths due to indirect function calls, or "dead code". To refine the function-call graph and make it more precise, we propose pruning techniques. These techniques identify and remove nodes and edges that correspond to dead code or unexecuted indirect calls, thus reducing inaccuracies. We also employ the points-to-analysis technique to further enhance the precision of the call graph by solving the aliases of pointers.

The iterative refinement through SVF and the precision enhancement techniques ultimately result in a robust, accurate function-call graph. This graph accurately reflects the interdependencies between various functions and serves as a fundamental stepping stone for the subsequent steps in the establishment of the legitimate sys-call set.

Mapping Library Functions to System Calls System calls are predominantly invoked through library functions. Before determining the legitimate system call set for application functions, we need to establish the mapping relationship between library functions and system calls. This mapping plays a fundamental role in refining our legitimate system call set, substantially enhancing the accuracy of our system call restriction methodology.

However, this mapping process is replete with challenges. Firstly, a single library function might invoke multiple system calls, leading to many-to-many relationships in our mapping. To manage this complexity, we use a relational database model to store and organize the information. Secondly, given the probability of library functions employing dynamic loading and runtime linking in specific contexts, the reliability of static analysis may be undermined. To mitigate this limitation, we incorporate dynamic analysis into our methodology, ensuring a more comprehensive assessment.

Specifically, we analyze the source code of the application to identify all the invoked library functions. Based on the GNU C Library glibc the prevalent C library for Linux applications, we map each library function to the system calls it can potentially invoke. The result of this step is a comprehensive mapping between library functions and system calls. This map provides a granular view of how different parts of an application interact with the operating system, thereby contributing to the establishment of a more precise legitimate sys-call set.

Fig. 6 displays our outcome. The red rectangle indicates a system call. Among them, certain application functions directly invoke system calls. For instance, p_listn directly calls the listen system call. Also, library functions may invoke numerous system calls, and several library functions are just served by a single system call.

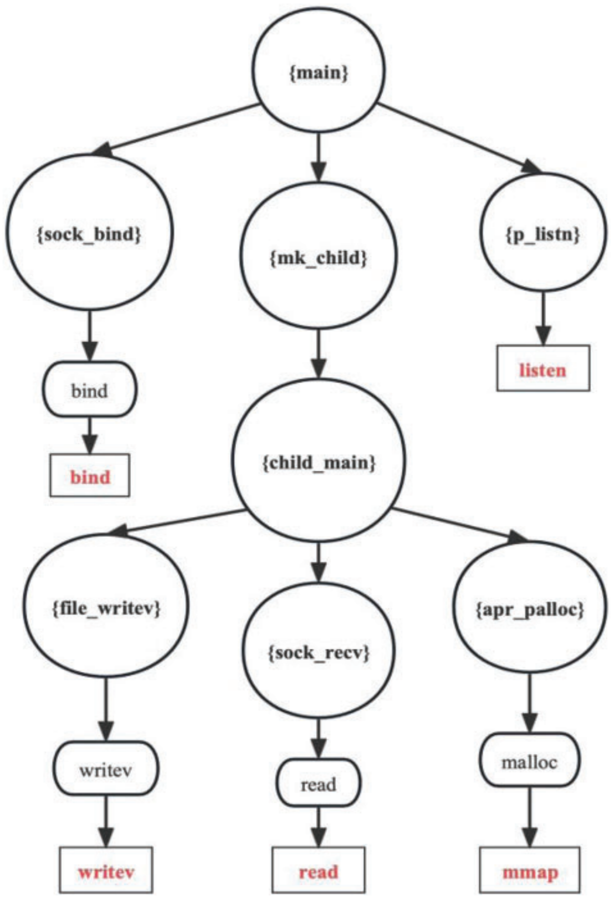


Figure 6: Function call graph and mapping of library functions to system calls

Constructing the Sys-call Set for Each Function Based on the precise function call graph and the mapping relationship between library functions and system calls, we employ the algorithm1 to construct a legitimate sys-call set for each function in the application. The detailed steps of this algorithm are provided below:

1. Initialization: Through our prior analysis, we have obtained the application's function-call graph and the mapping table between library functions and system calls. During this phase, three crucial data structures are set up:

- The Graph: This represents the function call graph of the target application, as determined by SVF.
- The Hash Table: This facilitates the mapping between library functions and system calls.
- Whitelist Sets: These are curated for each function within the application.

Initially, the whitelist sets are empty, and the call graph contains all functions which are to be processed.

2. In general, library functions are primarily leaf nodes, and we start the analysis from the leaf nodes. If the shape of the node is a circle (This means that the graph node is an applied function), we set its corresponding set to be an empty set. If the shape of the node is a rectangle (This means that the graph node is a library function), we just map the library function to sys-call. If the whitelist set no longer expands, the algorithm ends.

3. While the set of functions is not empty, we add it to its caller's whitelist set. Each node will be processed in this way until the parent node is empty. Go back to Step 2.

The algorithm we developed for building software's system call sets at the function level distinguishes itself from traditional methods by its focus on fine-grained analysis. It starts with initializing data structures like the function call graph and a mapping table for library functions and system calls. The process involves analyzing leaf nodes (library functions) in the graph and mapping them to system calls. It then recursively expands the whitelist set for each function, adapting to the function's call hierarchy.

Algorithm 1: Form a legal set of syscalls for each function

```
Require: source code of the target application source code of glibc
Ensure: Set of syscalls for each function
1: Run SVF to get the function call graph fcg of the target app
2: Run Egypt to get the map m from library functions to system calls
3: for each leafnode in fcg do
4:     if leafnode. Shape == circle then
5:         this.Set = Ã
6:     else
7:         value = map.get(this.label)
8:         this.set.add(value)
9:     end if
10:    while leafnode.caller != Ã¸ do
11:        caller.set.add(leafnode.set)
12:        leafnode = leafnode.caller
13:    end while
14: end for
```

This method improves upon current techniques by offering a more precise, function-level control of system calls, leading to enhanced security with reduced false positives and negatives. It is an adaptive approach that aligns closely with the actual behavior of the software.

Our framework is built from the application function layer, which is different from the TEMP framework. Based on this example, we will now explain how these two frameworks affect the final output.

The result is divided into two stages within the framework of the TEMP. The legitimate set for the initial stage is {writev, read, malloc, bind, listen} while the legitimate set for the serving stage is {writev, read, malloc}. Under FFRA, not only should the result of the TEMP be included, but functions such as apr_palloc should also be subject to more granular restrictions as shown in Fig. 7.

```
file_writev: {writev}
sock_ recv: {read}
apr_ palloc: {malloc}
sock_ bind: {bind}
p_listn: {listen}
child_ main: {writev ,read ,malloc}
mk_ child : {writev ,read ,malloc}
main: {writev ,read ,malloc ,bind ,listen}
```

Figure 7: The result of simple example

Supplement for each Sys-call Set Despite the comprehensiveness of static analysis, some system calls might still be overlooked. Consider an application that utilizes dlopen() and dlsym() functions, which are provided by the dynamic linking loader. These functions allow an application to load a shared library at runtime and retrieve the addresses of functions within that library, respectively. In such scenarios, system calls are challenging to capture through static analysis alone.

If these system calls are ignored, when the program is running normally, there will be many false positives. Therefore, we employ dynamic analysis to collect "special" system calls as a supplement to the legitimate set.

4.3 Abnormal Behavior Detection

Custom system call "ins" The method in this framework is considered from the application function layer, but the strace tool has no stage distinction in dynamically capturing the actual system call sequence. However, arbitrarily inserting functions can disrupt the original execution sequence of the program. For example, if the print function is inserted when the write system call appears later in the program since the print function will eventually call write and the cache policy of the operating system, only one write system call will be called. This does not conform to the logic of our analysis.

Therefore, we have customized a system called "ins" to mark the program running. Specifically, we add an individualized system call number and function name to the file: /usr/src/linux-X.XX/arch/x86/syscalls/syscall 32.tbl.

Instrument the source code We recompiled the LLVM compiler and made corresponding changes to the strace tool to use ins ("XXX", enter) to mark when entering a function, and use ins ("XXX", exit) to mark when exiting a function.

Track and compare In our FFRA framework, the validation process involves real-time monitoring and analysis of software execution sequences, tracking system calls made by each function during runtime. This helps in understanding the normal behavior of the software. We compare these observed sequences with pre-established legitimate system call sets derived from comprehensive static and dynamic analyses, using these sets as a baseline for normal behavior.

We use the Algorithm 2 to compare it with the legitimate sys-call sets for abnormal detection. Any deviation from these sets flags potential anomalies, such as unexpected system calls or unusual call sequences, indicating possible security breaches. This process is crucial for enhancing application security by ensuring only safe system calls are made, significantly reducing the attack surface.

Algorithm 2: Anomaly detection

```
Require: legal set of syscalls for each function syscalls sequence of target app
Ensure: Anomaly or not callStack = []
1: for event in seq do
2:     if event.sysName == "ins" then
3:         if event.funcAction == "enter" then
4:             callStack.add(event.funcName)
5:         else
6:             exit
7:         end if
8:     else
9:         if callee in lib.keys() then
10:            return true
11:        else
12:            return false
13:        end if
14:    end if
15: end for
```

5 Experimental Evaluation

The focus of our experimental evaluation lies in assessing the additional attack surface reduction achieved by FFRA to the existing framework and evaluating its security benefits. Our experiments are conducted on a set of six popular server applications: Nginx, Apache Httpd, Lighttpd, Bind, Memcached, and Redis.

To evaluate how well the attack surface is reduced, we analyze the number of system calls that are restricted under the above frameworks. Specifically, we calculated the retention rate of high-risk system calls.

In addition, we also assess how well the FFRA framework defends against ROP attacks. We test it against a large collection of shell codes to see how well it blocks exploit code.

5.1 Filtered System Calls

The number of accessible system calls serves as a measure of the potential attack surface. Fewer accessible system calls make it more challenging for attackers to succeed. By considering the function layer, we substantially reduce the number of accessible system calls, offering a more refined approach than previous methods that consider the entire program lifecycle.

We compare our approach with the TEMP framework to show the benefit of applying FFRA. As shown in Table 1, FFRA retains fewer system calls on average than TEMP. In most cases, the reduction is significant (in the best case for Nginx, the number of system calls drops from 97 to 51, while in the worst case for Bind, an additional 7 system calls are removed).

Table 1: Comparison between FFRA and TEMP

Applications	FFRA		Temp	
	Fun-num	Ave-syscall	Initial	Serving
Redis	4349	59	90	82
Memcached	1471	55	99	84
Bind	6801	78	99	85
Lighttpd	802	40	95	76
Apache httpd	2491	52	94	79
Nginx	1626	51	104	97

5.2 *Security-Critical System Calls*

Compared to the reduction in the number of system calls, a more crucial question is whether the removed system calls are 'critical' or not, i.e., whether they will hinder the execution of exploit code that relies on them.

We chose a set of 10 security-critical system calls that are used as part of shellcode and ROP payloads: Chmod, fchmod, chown, fchown, lchown, execve, mount, rename, open, link. As shown in Table 2, we observed a universal decrease in the retention of these security-critical system calls, with a notable reduction reaching up to 0.14%. A diminished retention rate of security-critical system calls proportionally reduces the potential attack surface.

Table 2: The retention rate of high-risk sys-calls

Syscalls	Redis	Memcached	Nginx	Lighttpd	Apache httpd	Bind
Chmod	67.03% (2915/4349)	76.95% (1132/1471)	71.16% (1157/1626)	66.96% (537/802)	62.30% (1552/2491)	88.53% (6021/6801)
Fchmod	32.08% (1395/4349)	–	–	–	0.04% (1/2491)	0.65% (44/6801)
Chown	70.87% (3082/4349)	81.58% (1200/1471)	76.08% (1237/1626)	73.57% (590/802)	74.79% (1863/2491)	91.47% (6221/6801)
Fchown	32.05% (1394/4349)	–	–	–	0.12% (3/2491)	0.74% (50/6801)
Lchown		–	–	–	29.02% (723/2491)	0.63% (43/6801)
Execve	67.03% (2915/4349)	76.95% (1132/1471)	71.09% (1156/1626)	66.96% (537/802)	61.98% (1544/2491)	88.35% (6009/6801)
Mount	20.39%	21.16%	–	–	–	0.65%

(Continued)

Table 2 (continued)

Syscalls	Redis	Memcached	Nginx	Lighttpd	Apache httpd	Bind
		(300/1471)	(344/1626)			(44/6801)
Rename	32.05% (1394/4349)	0.14% (2/1471)	20.97% (341/1626)	–	29.02% (723/2491)	82.75% (5628/6801)
Open	32.47% (1412/4349)	25.29% (372/1471)	24.29% (395/1626)	9.60% (77/802)	29.51% (735/2491)	83.37% (5670/6801)
Link	32.61% (1418/4349)	21.41% (315/1471)	23.73% (386/1626)	10.22% (82/802)	32.28% (804/2491)	82.84% (5634/6801)

5.3 Shellcode Analysis

To evaluate the security benefits of FFRA, we collected a large and diverse set of exploit payloads. This set consists of 53 shellcodes from Metasploit and 514 shellcodes from shell-storm.

We specifically evaluated the number of shell codes mentioned above that can be protected by applying the FFRA framework. We compared the results with the implementation of the TEMP framework. We can see that under the FFRA framework, more shellcode attacks can be defended because FFRA is more fine-grained. The detailed results are presented in Fig. 8.

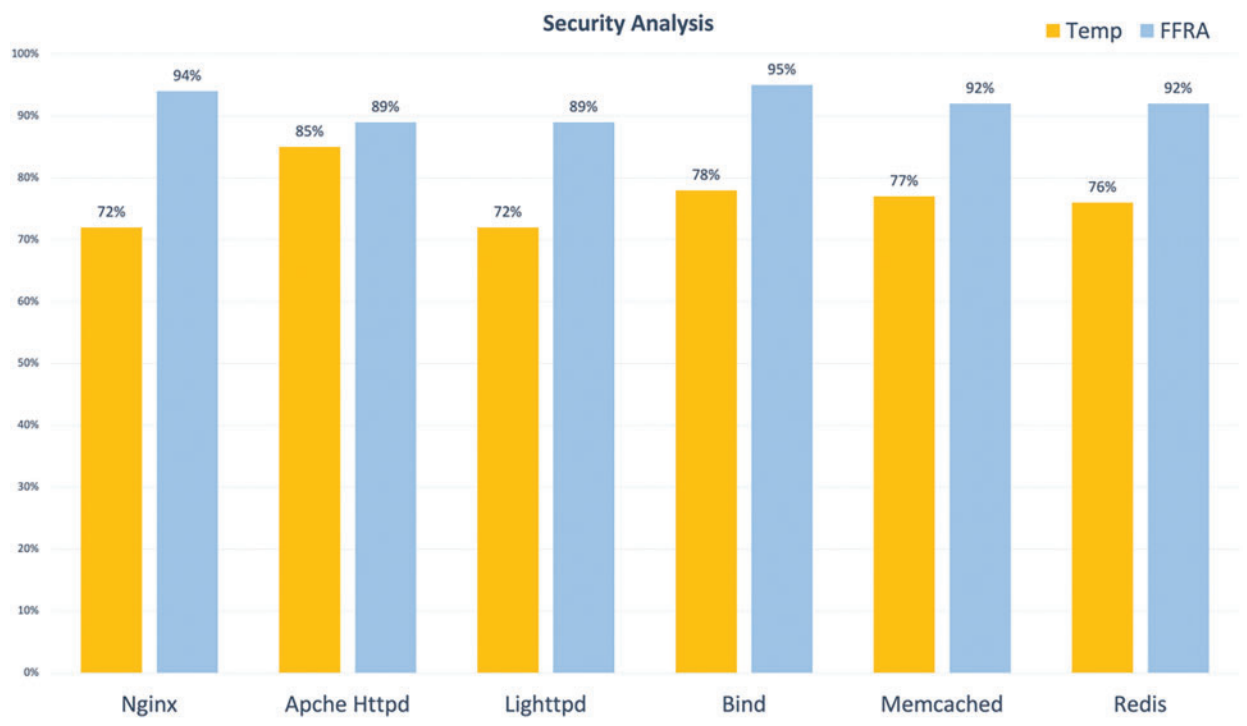


Figure 8: Shellcode analysis

5.4 Summary

FFRA demonstrates superior performance in both disabling system calls and detecting shellcode vulnerabilities compared to existing methods like TEMP. Specifically, FFRA achieves a more significant reduction in the number of accessible system calls, exemplified by the drop from 97 to 51 in the case of Nginx.

Furthermore, there is a notable reduction in the retention of security-critical system calls, which is essential for hindering exploit code execution. This reduction is quantified with a decrease reaching up to 0.14%. In shellcode analysis, FFRA outperforms TEMP by effectively defending against a greater number of shellcode attacks, showcasing its finer granularity and enhanced security capabilities.

6 Discussion and Limitations

Our approach does not remove any code from the protected program, and consequently cannot mitigate any vulnerabilities in the application itself or reduce the code that could be reused by an attacker.

As our results show, FFRA may not prevent all possible ways an attacker can perform harmful interactions with the OS. Our equivalent system calls analysis attempts to quantify the evasion potential by replacing system calls with others, but depending on the attacker's specific goals, there may be more creative ways to accomplish them using the remaining system calls.

For example, without our technique, an attacker could read the contents of a file simply by executing the cat program. Once the exec_family of system calls is blocked, the attacker would have to implement a more complex shellcode to open and read the file and write it to an already open socket. As part of our future work, we plan to extend our analysis by extracting the arguments passed to system calls and constraining them as well.

7 Conclusion

We propose a fine-grained function-level framework in this paper to remove unneeded system calls. It can make stricter restrictions on system calls from a more precise perspective than the existing TEMP framework.

FFRA includes two phases: Legitimate sys-call set establishment and abnormal detection. The initial phase employs static analysis to generate a legitimate sys-call set for each application function, and dynamic analysis is further used to compensate for the limitations of static analysis. The combination of dynamic and static analysis increases the precision of the legitimate sys-call sets. In the second stage, based on the legitimate sys-call sets generated in the first phase, it is judged whether an abnormality occurs during the actual running of the software.

Several experiments were designed to confirm our framework's security. In comparison to TEMP, FFRA filters more system calls and maintains low retention of these security-critical system calls. Also, FFRA successfully breaks more shellcode variants than TEMP does for each of the six tested applications. In addition, the deployment of FFRA essentially has no impact on the software's regular operation. In summary, it is proved that the FFRA framework is effective and efficient.

Acknowledgement: The authors express their gratitude for the support and resources provided by the School of Modern Posts, Nanjing University of Posts and Telecommunications. The authors also acknowledge the contributions of their peers and collaborators in the study's development. All authors have read and agreed to the published version of the manuscript.

Funding Statement: This work is supported by the National Natural Science Foundation of China under No. U20B2050 and Public Service Platform for Basic Software and Hardware Supply Chain Guarantee under No. TC210804A.

Author Contributions: The authors confirm their contribution to the paper as follows: Study conception and design: X. Zhang, L. Liu; data collection: X. Zhang; analysis and interpretation of results: X. Zhang, Y. Fan; draft manuscript preparation: X. Zhang; revise and polish: X. Zhang, L. Liu, Y. Fan; supervise: Q. Zhou. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the corresponding author, L. Liu, upon reasonable request.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] G. J. Holzmann, "Code inflation," *IEEE Softw.*, vol. 32, no. 2, pp. 10–13, 2015. doi: 10.1109/MS.2015.40.
[2] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments," in *Proc. 2017 Workshop Form. Ecosyst. around Softw. Trans.*, 2017, pp. 65–70.
[3] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *2018 IEEE Symp. Secur. Priv. (SP)*, 2018, pp. 161–175.
[4] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *27th USENIX Secur. Symp. (USENIX Security 18)*, 2018, pp. 869–886.
[5] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, 2019, pp. 70–83.
[6] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *28th USENIX Secur. Symp. (USENIX Security 19)*, 2019, pp. 1697–1714.
[7] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *28th USENIX Secur. Symp. (USENIX Security 19)*, 2019, pp. 1733–1750.
[8] F. Ullah, G. Srivastava, and S. Ullah, "A malware detection system using a hybrid approach of multi-heads attention-based control flow traces and image visualization," *J. Cloud Comput.*, vol. 11, no. 1, pp. 1–21, 2022.
[9] N. Mani, M. Moh, and T. S. Moh, "Defending deep learning models against adversarial attacks," *Int. J. Softw. Sci. Comput. Intell.*, vol. 13, no. 1, pp. 72–89, 2021.
[10] A. Mishra, N. Gupta, and B. B. Gupta, "Defense mechanisms against DDoS attack based on entropy in SDN-cloud using POX controller," *Telecommun. Syst.*, vol. 77, pp. 47–62, 2021. doi: 10.1007/s11235-020-00747-w.
[11] A. Singh and B. B. Gupta, "Distributed denial-of-service (DDoS) attacks and defense mechanisms in various web-enabled computing platforms: Issues, challenges, and future research directions," *Int. J. Semant. Web Inf. Syst. (IJSWIS)*, vol. 18, no. 1, pp. 1–43, 2022.
[12] R. Sharma and N. Sharma, "Attacks on resource-constrained IoT devices and security solutions," *Int. J. Softw. Sci. Comput. Intell.*, vol. 14, no. 1, pp. 1–21, 2023.
[13] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *Proc. 2017 IEEE Int. Conf. Softw. Testing, Verification and Validation (ICST)*, 2017, pp. 92–102.
[14] J. Thalheim, P. Bhatotia, and P. Fonseca, "CNTR: Lightweight {OS} containers," in *Proc. 2018 USENIX Annu. Tech. Conf. (USENIX ATC 18)*, 2018, pp. 199–212.
[15] L. Lei *et al.*, "Speaker: Split-phase execution of application containers," in *Detection Intrusions and Malware, and Vulnerability Assess.: 14th Int. Conf. (DIMVA 2017)*, 2017, pp. 230–251.

[16] J. A. Priest and W. Powrie, "Determination of dynamic track modulus from measurement of track velocity during train passage," *J. Geotech. Geoenviron. Eng.*, vol. 135, no. 11, pp. 1732–1740, 2009. doi: 10.1061/(ASCE)GT.1943-5606.0000130.

[17] V. Rastogi, D. Davidson, L. de Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *Proc. 2017 11th Joint Meet. Found. Softw. Eng.*, 2017, pp. 476–486.

[18] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha, "New directions for container debloating," in *Proc. 2017 Workshop Form. Ecosyst. Around Softw. Transf.*, 2017, pp. 51–56.

[19] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd Int. Symp. Res. Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 443–458.

[20] M. D. Brown and S. Pande, "CARVE: Practical security-focused software debloating using simple feature set mappings," in *Proc. 3rd ACM Workshop Form. Ecosyst. around Softw. Trans.*, 2018, pp. 1–7.

[21] X. Wang, T. Hui, L. Zhao, and Y. Cheng, "Input-driven dynamic program debloating for code-reuse attack mitigation," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 934–946.

[22] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Secur. Symp. (USENIX Security 20)*, 2020, pp. 1749–1766.

[23] K. Asmitha and P. Vinod, "A machine learning approach for linux malware detection," in *2014 Int. Conf. on Issues Chall. Intell. Comput. Tech. (ICICT)*, 2014, pp. 825–830.

[24] A. S. Morris, "Measurement and instrumentation principles," *Meas. Sci. Technol.*, vol. 12, pp. 1743–1744, 2001. doi: 10.1088/0957-0233/12/10/702.