# Detecting Malicious Behavior in Android OS through Syscall Hooking and YARA Integration

1st Lucia Lup
*Technical University of Cluj-Napoca*
*author*

2nd S.L. Dr. Eng. Ciprian Pavel Oprisa
*Technical University of Cluj-Napoca*
*supervisor*

***Abstract*—As the dominant operating system for mobile devices, the Android OS is a prime target for malware attacks. This paper explores the technique of syscall hooking to monitor system calls within the Android kernel, enabling the detection of irregular activities that may indicate malicious behavior. Our approach involves a custom Android Goldfish kernel deployed in an emulator environment, as well as the integration with YARA, a powerful tool for malware researchers. We present an efficient syscall logging mechanism that can record a wide range of system calls while addressing concerns such as deadlocks, data loss, and verbosity. We demonstrate its effectiveness in detecting ransomware by monitoring rapid encryption activities occurring in device photos. Our syscall hooking method is hooking directly into the kernel, ensuring that bypassing is not possible and providing substantial system insights to identify malicious behavior.**

***Index Terms*—Android OS, syscall hooking, malware detection, YARA integration, ransomware detection**

## I. Introduction

Android is an open-source mobile operating system developed by Google for smartphones and tablets. Primarily based on the Linux kernel, it has since become the leading operating system for mobile devices, with over 2.5 billion devices [1]. The surge in mobile devices, especially Android-powered ones, has opened up a whole new realm of cybersecurity challenges [2]. Consequently, there is an essential need for techniques and mechanisms to protect against malicious attacks. This paper aims to explain a method of malware analysis based on syscall hooking and pattern matching.

To understand the methodology proposed in this paper, it is essential to first comprehend the function of system calls (syscalls). A "syscall" or system call is a mechanism provided by operating systems that allows processes running in User Mode to interact with hardware devices such as CPUs, disks, and printers [3]. Every operation that requires the kernel's involvement, such as file operations or network access, is realized through a syscall.

Several methods can be implemented to hook into the syscalls for system monitoring and intrusion detection, with varying degrees of effectiveness. Popular approaches include function pointer modification inside the syscall table, kernel object hooking, and hooking via the virtual file system. However, these common methods often present limitations in scope and effectiveness and can be circumvented. This paper proposes a comprehensive, yet invasive approach by directly hooking into the Android Linux Kernel to capture all syscalls. Although this method has a high degree of intrusion, it provides the highest level of accessibility, allowing developers to track malicious behavior directly from the kernel, which is nearly impossible to bypass.

Over time, the complexity and amount of Android malware have increased, as malicious individuals continuously find new ways to exploit vulnerabilities. Traditional methods of malware analysis often fall short in the face of these evolving threats. This paper proposes an integrated approach combining syscall hooking with pattern matching for efficient Android malware detection and analysis. Using pattern matching techniques effectively detects malware by applying predefined rule sets, such as those found in Yara [4], a powerful tool for malware researchers. In our approach, these rules can be dynamically created and loaded in real-time to detect suspicious activities, such as rapid file encryption which may indicate ransomware.

This paper is organized as follows: In the 'Introduction,' we provide an overview of the problem and our approach. The 'Methodology' section details our hooking mechanism and pattern-matching technique used for system call monitoring and malware detection. The 'Experiments and Results' section presents our testing scenarios, demonstrating the effectiveness of our approach. In the 'Related Work' section, we discuss existing methods and how our approach builds upon and differs from them. The 'Discussion' section elaborates on the implications of our work, the potential for further enhancements, and its broader applications. Finally, the 'Conclusion' provides a summary of our work, its contributions, and future prospects.

## II. Methodology

In this chapter, we turn our focus towards the application of our real-time syscall hooking and pattern-matching methodology in the context of Android malware analysis. Furthermore, we present an overview of Android malware and its typical behaviors, as well as how our approach can help detect them.

### A. System call hooking

In the Linux kernel (and, by extension, the Android kernel), syscalls are one of the primary mechanisms by which user-space programs interact with the system kernel.

The syscall flow starts when an application issues a syscall, which is a request to the kernel to perform some operation on its behalf. The kernel has a syscall table, resembling a lookup table, where each syscall is associated with a function that implements it. When a syscall is issued, the kernel identifies the syscall number from the syscall table and then jumps to the corresponding function to execute it. After its execution, the result is passed back to the user-space application.

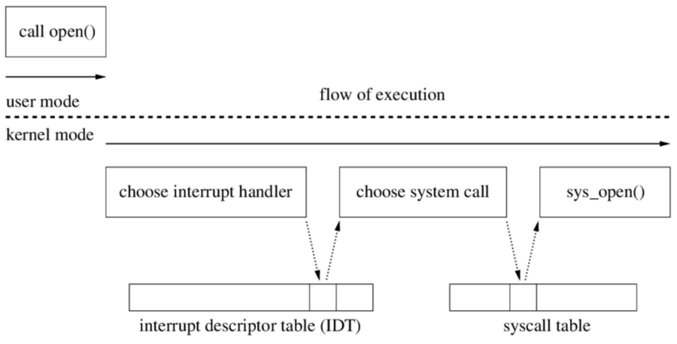


Fig. 1. Syscall execution flow. Adapted from Andreas Bunden et al. (2004) [5].

### *1) Hook function overview*

We implemented a standalone hook function designed to log system calls in a Linux environment. The function is invoked at the start of each system call that we want to trace (e.g. 'write'), intercepts its arguments (file descriptor, path, filename, buffer, and count), logs them, and then allows the system call to proceed.

The hook() function serves as a centralized utility that records details about each system call made in the system. It logs the syscall name, argument types, and specific details based on the arguments. A brief outline of the workflow is presented in Fig. 2.

```
0: function HOOK(syscall_name, arg_types, ...)
0:   Initialize hook by getting current CPU and log buffer
0:   Allocate memory for buffer
0:   if memory allocation is successful then
0:     Begin Variable Argument Processing
0:     Create log buffer by processing variable arguments
0:     End Variable Argument Processing
0:     if creation of log buffer is successful then
0:       Add the log message to the per-CPU log buffer.
0:     end if
0:   end if
0:   Free the temporary buffer and release the CPU.
0:   Release current CPU
0: end function=0
```

Fig. 2. Hook Function Workflow

### *2) Function implementation*

**Initial setup** We used the Android Goldfish kernel for this research, thanks to its ability to be built and run on an Android emulator, which provides an accessible environment across a range of Android devices. We successfully managed to operate an Android Virtual Device (AVD) with Android Version 7, using the Linux Kernel branch android-goldfish-3.10-n-dev [6]. Although we restricted our operations to these specifications, other combinations may give successful results.

**Deadlock handling** To avoid deadlocks, the function uses the logging_allowed flag on a per-CPU basis to prevent recursive logging, especially for the read and write syscalls. Recursive logging occurs when the process of logging an event itself triggers another loggable event. This could happen if, for instance, the process of logging a syscall involved writing to a file. Since the write syscall is also being logged, this could lead to an infinite loop, causing a deadlock. The hook function avoids this by using the logging_allowed flag to temporarily disable logging during the process of writing log messages.

**Minimize overhead** Logging in real time all of the syscalls might cause a high overhead in the system, which would later result in a kernel panic [3] or loss of log data (some logs will be skipped). In order to minimize overhead, we have implemented two major features:

- *Ring buffer*. The print operation is time-costly, and due to a large number of logs that are individually printed, this would lead to high overhead. Consequently. in order to manage log volume, improve performance, and prevent memory overflow the function uses a ring buffer to store log messages. A ring buffer is an array of fixed size, and when it is full, the next item to be added replaces the oldest item, and the buffer gets flushed to printk [8]. This way, the buffer always contains the most recent messages, and its size in memory remains constant, preventing memory overflow. The function also uses a timer to regularly flush the log buffers to printk, ensuring that log messages are not lost if the buffer fills up.
- *Exclusion List*. To reduce log volume and filter out irrelevant messages, the function uses an exclusion list. This is a list of strings that, if they appear in a system call's path or filename, will cause the syscall to be skipped over.

**Function signature** The hook function has a highly flexible signature that can adapt to any system call. It takes the name of the system call as a string, a format string that specifies the types of the system call's arguments (described in Table I), and then a arbitrary number of arguments that match the format string. This allows the function to be used with any system call, regardless of the number or types of its arguments.

The argument types are passed as a format string, with each character representing a type. Table I explains the possible parameter types.

### *3) Hook Usage*

In order to trace a syscall's behavior, we can simply "hook" into its implementation, using the previously described hook

TABLE I
ARGUMENT POSSIBLE TYPES

| Argument Value | Argument Type | Argument Description |
|---|---|---|
| d | int | File Descriptor |
| p | char* | File Path |
| n | char* | File Name |
| f | int | Flags |
| c | int | Count |
| b | char __user* | Buffer |

function. We can call it inside the system call definition we wish to monitor. This is done by passing the system call's name, argument types, and arguments to the hook function.

For instance, the 'write' system call is hooked as shown in 3. Thanks to its volatile nature, we can choose to monitor any syscall, as long as we only pass supported parameters (Table I). Furthermore, the list can be expanded and modified in future development.

```
SYSCALL_DEFINE3(write, unsigned int, fd,
   const char __user *, buf, size_t, count)
{
    ... //original code
    hook("write", "dcb", fd, count, buf);
}
```

Fig. 3. Example of a 'write' System Call Hook

### B. Pattern-matching and malware analysis

The use of pattern matching as a method for detecting malware has been extensively discussed in several studies. For instance, [9], [10] shows how YARA [4], an open-source tool, uses pattern matching to identify malware samples based on pre-defined rules. However, their work primarily focuses on static pattern matching on pre-analyzed samples, which is not directly applicable to real-time Android malware detection and does not address the issue of dynamic rule creation based on runtime behavior.

Following syscall hooking, we integrate the pattern-matching engine YARA to identify potentially malicious behavior. The logged syscalls from the previous step form a sequence that we can analyze to find recurring patterns. Our approach utilizes real-time pattern matching, which means we continuously match the patterns as new syscalls are logged.

We define patterns for malicious behavior based on previous literature and malware analysis. These patterns can range from simple sequences (such as a series of file access syscalls followed by network access syscalls) to complex behaviors that may involve intricate combinations of various syscalls. When a match is found, an alert is triggered, and the system administrator can take appropriate action, such as terminating the application.

Because we use the YARA engine, we can write our own custom YARA rules [9] used for pattern-matching. A guide on how to write a custom YARA rule can be found here [9]. The rules must be stored in a .yara file, and it can be run against a log file with the following command:

```
    yara custom_rules.yara logs.txt
```

In order for the command to be executed, YARA must be installed in the system. If there is a match between the patterns specified in the YARA rules and the content of the "logs.txt" file, the command would display information about the matched rule(s) or any identified malware or suspicious patterns. However, if there are no matches, the command does not produce any output or it simply indicates that no matches have been found based on the specified YARA rules.

## III. EXPERIMENTS AND RESULTS

The first experiment to be implemented was the implementation of syscall hooking on the Android goldfish kernel as discussed in the methodology. A custom version of this kernel was built, deployed on a compatible emulator, and used to trace syscalls directly from the kernel itself. A first observation was that we needed to start the emulator in root mode. The command used to start the emulator with a custom kernel is shown in Figure 4.

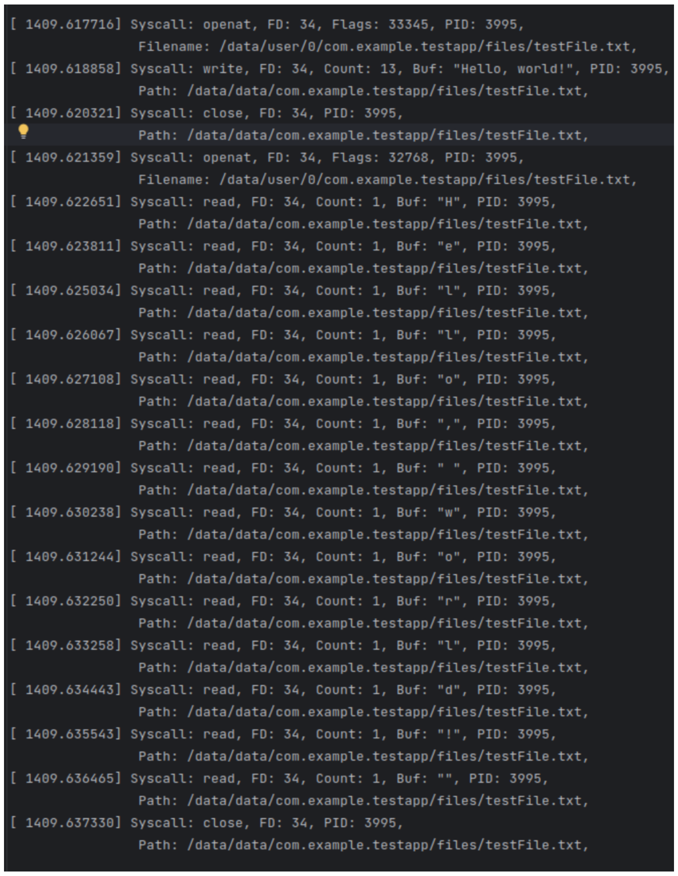
```
cd @emulator_folder && ./emulator -verbose
  @avd_name @kernel_image_path -show-kernel
  -qemu -enable-kvm
```

Fig. 4. Emulator Setup Commands

In the above command, emulator_folder represents the folder where the emulator is installed, avd_name the name of the existing AVD (can be created with the AVD manager), and kernel_image_path is the path to the pre-built kernel image.

Firstly, the correctness of the hook was tested by developing additional simple Android applications designed to perform various file operations (Fig. 5). The hook logged each syscall performed, including openat, read, write, and close operations, along with the associated parameters like the file descriptor (fd), process ID (pid), path, filename, flags, buffer, and count. The hook functionality was successful in logging all sycall operations missing or bypassing any, demonstrating its accuracy and reliability. Furthermore, the custom kernel proved stable and reliable, maintaining efficient system performance despite the intensive logging processes. The output of this logging experiment is found in Fig. 5.

A critical aspect of the test was verifying the verbose hook function and its associated synchronization and locking mechanisms. The detailed logging generated by hooking every syscall presented several challenges, including the risk of deadlocks and lag due to the high volume of data, which may result in data loss. A series of safeguards, including a check variable to prevent recursive logging of read/write operations, a ring buffer to efficiently manage log storage, race conditions to prevent log corruption, and an exclusion list to filter unnecessary log entries, mitigated these challenges.



```
[ 1409.617716] Syscall: openat, FD: 34, Flags: 33345, PID: 3995,
                Filename: /data/user/0/com.example.testapp/files/testFile.txt,
[ 1409.618858] Syscall: write, FD: 34, Count: 13, Buf: "Hello, world!", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.620321] Syscall: close, FD: 34, PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.621359] Syscall: openat, FD: 34, Flags: 32768, PID: 3995,
                Filename: /data/user/0/com.example.testapp/files/testFile.txt,
[ 1409.622651] Syscall: read, FD: 34, Count: 1, Buf: "H", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.623811] Syscall: read, FD: 34, Count: 1, Buf: "e", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.625034] Syscall: read, FD: 34, Count: 1, Buf: "l", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.626067] Syscall: read, FD: 34, Count: 1, Buf: "l", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.627108] Syscall: read, FD: 34, Count: 1, Buf: "o", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.628118] Syscall: read, FD: 34, Count: 1, Buf: ",", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.629190] Syscall: read, FD: 34, Count: 1, Buf: " ", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.630238] Syscall: read, FD: 34, Count: 1, Buf: "w", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.631244] Syscall: read, FD: 34, Count: 1, Buf: "o", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.632250] Syscall: read, FD: 34, Count: 1, Buf: "r", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.633258] Syscall: read, FD: 34, Count: 1, Buf: "l", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.634443] Syscall: read, FD: 34, Count: 1, Buf: "d", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.635543] Syscall: read, FD: 34, Count: 1, Buf: "!", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.636465] Syscall: read, FD: 34, Count: 1, Buf: "", PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
[ 1409.637330] Syscall: close, FD: 34, PID: 3995,
                Path: /data/data/com.example.testapp/files/testFile.txt,
```

Fig. 5. Syscall logs for file operations.

For the second part, we worked on the detection of ransomware. Ransomware, as its name suggests, is a form of malware that holds a victim's data ransom. This is typically achieved by encrypting the victim's files, making them inaccessible, and then demanding a ransom to decrypt them and restore access [16]. This malicious activity often manifests in a characteristic pattern that can be detected – sequences of open, write, and close operations are performed on files in quick succession, indicating the encryption activity that is indicative of ransomware behavior.

Our second significant evaluation involved integrating the YARA engine to detect ransomware. A YARA rule designed to detect the presence of ransomware-like behavior in device photos was developed and tested. This rule checked for sequences of open, write, and close operations performed in quick succession, indicative of encryption activity. An example of how to write YARA rules for detecting malware is found here [11]. The structure of the YARA rule designed to detect ransomware behavior in device photos can be found in 6.

To test the efficacy of this rule, we introduced a third application that was designed to simulate ransomware behavior. Once this app was run on our emulator environment, we applied the YARA rule to the syscall logs that were generated during its operation.

```
rule RansomwareImageAccess
{
 meta:
    description = "Detect␣potential␣ransomware"
    reference = "Internal␣research"
    date = "2023-06-18"
    version = "1.1"

    strings:
     $open = /Syscall: openat, .{1,100}Filename:
       \/data\/media\/0\/Pictures\/.{1,100}\.png/
     $write = /Syscall: write, .{1,100}Path:
       \/data\/media\/0\/Pictures\/.{1,100}\.png/
     $close = /Syscall: close, .{1,100}Path:
       \/data\/media\/0\/Pictures\/.{1,100}\.png/
     $filetype = /\.png$/

    condition:
       all of them for 30 or more times

    private function all of them for(n: int) {
     for all i in (1..#open)
       : (i + n - 1 <= #open) {
      for j in (0..n - 1) {
       if (@open[i + j] != @write[i + j]
          or @open[i + j] != @close[i + j]) {
        break
       }
       if (j == n - 1) {
        return true
       }
      }
    }
    false
    }
}
```

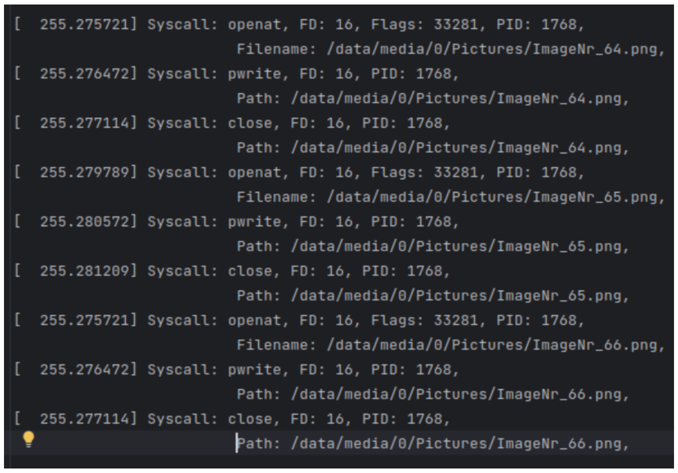
Fig. 6. YARA rule designed to detect ransomware in device photos.

A screenshot of a log sequence (Fig. 7), clearly indicates the sequence of open, write, and close operations being performed rapidly on a series of image files, a pattern that matches our YARA rule. The rapid succession of these operations is unlikely to be caused by a user's activity, providing further validation that our YARA rule has correctly identified potential ransomware behavior.

The result of applying the YARA rule was a successful match, demonstrating the rule's effectiveness in identifying the simulated ransomware activity. The YARA engine output the following when the rule was matched:

```
Possible malware behavior detected:
RansomwareImageAccess logs.txt
```

This successful match confirmed the correctness and accuracy of our YARA rule in identifying ransomware-like behavior based on syscall patterns. It demonstrates the capability of our system to utilize pattern matching to detect malware activity in the Android operating system. This further exemplifies the potential of our approach, offering a kernel-level safeguard impacting malware detection in Android systems.



```
[  255.275721] Syscall: openat, FD: 16, Flags: 33281, PID: 1768,
                        Filename: /data/media/0/Pictures/ImageNr_64.png,
[  255.276472] Syscall: pwrite, FD: 16, PID: 1768,
                        Path: /data/media/0/Pictures/ImageNr_64.png,
[  255.277114] Syscall: close, FD: 16, PID: 1768,
                        Path: /data/media/0/Pictures/ImageNr_64.png,
[  255.279789] Syscall: openat, FD: 16, Flags: 33281, PID: 1768,
                        Filename: /data/media/0/Pictures/ImageNr_65.png,
[  255.280572] Syscall: pwrite, FD: 16, PID: 1768,
                        Path: /data/media/0/Pictures/ImageNr_65.png,
[  255.281209] Syscall: close, FD: 16, PID: 1768,
                        Path: /data/media/0/Pictures/ImageNr_65.png,
[  255.275721] Syscall: openat, FD: 16, Flags: 33281, PID: 1768,
                        Filename: /data/media/0/Pictures/ImageNr_66.png,
[  255.276472] Syscall: pwrite, FD: 16, PID: 1768,
                        Path: /data/media/0/Pictures/ImageNr_66.png,
[  255.277114] Syscall: close, FD: 16, PID: 1768,
                        Path: /data/media/0/Pictures/ImageNr_66.png,
```

Fig. 7. Syscall log sequence exhibiting ransomware behavior.

In conclusion, the project succeeded in achieving its objectives, demonstrating the efficacy of the custom kernel with the verbose hook function, managing detailed logging and mitigating overhead, and applying YARA rule-based malware detection on syscall logs. The results indicate that this methodology is a feasible and effective method for analyzing system behavior and identifying malicious actions at the kernel level. This lays a strong foundation for future research in enhancing the security of Android systems.

## IV. Related Work

Starting with kernel-level monitoring, several researchers have studied syscall hooking as a security measure. For instance, 'DroidScope' by Yan and Yin (2012) [13] is an Android-focused dynamic binary instrumentation tool that provides low-level interaction capabilities, including system calls. Our work builds on such research, pushing further by developing a custom verbose hook function in the Android Goldfish kernel for real-time syscall monitoring, resulting in a more detailed system inspection.

System call hooking is a technique frequently employed in computer security and malware analysis. It involves redirecting system call requests, allowing a monitoring process to examine or modify the behavior of an operating system. Various methods exist for syscall hooking, each with its own merits and limitations.

A common method of syscall hooking involves manipulating the system call table directly. The system call table is an array of pointers to system call handler functions 1. By modifying these pointers, one can intercept and monitor system calls. This method, while simple and effective, is invasive and can be easily detected, limiting its practical applications [18].

A more subtle approach is the use of kernel modules for hooking. This method involves loading a kernel module that overwrites the system call table entries, thereby intercepting system calls. While this method is less detectable than direct system call table manipulation, it can still be discovered through in-depth system analyses [19].

Loadable Kernel Module (LKM) rootkits utilize a technique that involves hiding the hook itself. This method works by directly altering the start of a system call's function in memory, effectively redirecting it to a malicious function. This approach is stealthy but complex to implement and can be detected by recent technologies [20].

Our method, while more invasive than these traditional syscall hooking techniques, provides a deeper level of visibility into the Android kernel's operation. By creating a custom hook function within the kernel, we can monitor a wider range of system calls without the possibility of bypass.

The utilization of syscall traces for security analysis has also been investigated previously. Bathia and Khausal (2017) [14] proposed a Machine Learning (ML) based approach to detect malicious activities using syscall trace analysis on the Android platform. Our work complements this approach but opts for a more deterministic pattern-matching technique using YARA rules, providing the flexibility for users to define custom patterns to identify malware.

Casolare et al. (2021) [17] proposed a novel approach of using system call-based image representation for dynamic mobile malware detection. In their work, they transformed system call traces into an image representation which they used as input to various machine and deep learning algorithms. Their research demonstrated effective malware detection, achieving an accuracy of up to 0.89. However, while their work examined the dynamic analysis of Android applications, it was conducted at the user-space level. Our project expands on the concept of system call monitoring, by implementing it directly within the kernel, ensuring a comprehensive level of analysis.

In the field of ransomware detection, Andronio et al. (2015) [15] developed a system, HelDroid, that identifies potential ransomware behavior by looking for certain tell-tale signs, such as screen locking and rapid file encryption. Our project, too, uses a similar behavioral pattern for ransomware detection but implements it at the kernel level for deeper visibility and more robust results.

The integration of YARA in Android system security is less explored. YARA, originally a tool for malware researchers, has not seen extensive use in runtime Android security. We bridge this gap, offering an integration of YARA engine for pattern detection, significantly expanding our system's malware detection capabilities.

Therefore, while our work shares common traits with existing research, it distinguishes itself through the combination of kernel-level syscall hooking and YARA integration. This unique approach makes our project a useful contribution to the field of Android system security.

## V. Discussion

There are many opportunities for the future development and improvement of this project. Thanks to the integration with the YARA engine, the user can define custom YARA rules to detect specific patterns in syscall logs. Adding support for network-based syscalls could extend the detection capabilities to network-based attacks. Also, Machine Learning techniques for pattern recognition can be integrated [12] in order to significantly improve the system's ability to flag malicious behavior in real time.

Furthermore, the independent hook function allows for the tracing of various system calls and may be easily expanded to accommodate a variety of arguments. Additionally, the logging system can be further optimized to reduce system overhead. While the circular buffer (ring buffer) and exclusion list handled the volume of logs effectively in the test scenarios, large-scale real-world deployments might demand more efficient data processing strategies. More complex data structures or compression methods could be researched for the capacity to handle larger volumes of log data.

This project offers a wide range of practical applications. It may, for example, be used as a security tool for Android app developers, allowing them to monitor syscall behavior during testing and identify any potentially harmful activities. This program can also be used in malware research, letting researchers see how malicious software interacts with the system at the kernel level in real-time. This visibility can support the creation of more effective measures against malware.

## VI. Conclusion

Due to its open-source nature and popularity, Android Operating System has increasingly become a primary focus for malicious attacks. In this study, we presented the use of syscall hooking in the Android Goldfish kernel as a method for enhancing the security of Android systems. Deployed in an emulator environment, our custom kernel facilitates real-time monitoring of system calls.

The primary contribution of this project is an effective syscall logging mechanism, performed by a custom standalone hook function developed directly in the kernel. Addressing several issues such as deadlocks, managing verbosity, and potential data loss, this function successfully recorded extensive system call operations in various experiments.

Another significant aspect of our work was the successful integration with the YARA engine. The engine's ability to define custom rules introduces increased flexibility, enabling the flagging of various patterns in the syscall logs that could indicate malicious behavior. We created and used a custom YARA rule to detect rapid encryption activities that typically indicate the presence of ransomware.

Expanding our project to include network-focused syscalls, integrating complex machine-learning methods for detecting patterns, and improving the efficiency of our logging system would increase its usefulness in real-life scenarios. This would benefit not only malware researchers but also Android App developers by allowing them to monitor system calls activities at the kernel level in real-time.

## References

[1] Android. ”What is Android?” Available online: https://www.android.com/intl/en_uk/what-is-android/.

[2] ”Mobile malware evolution 2021” Available online: https://securelist.com/mobile-malware-evolution-2021/105876/.

[3] D. P. Bovet și M. Cesati, Understanding the Linux Kernel, O'Reilly, 2002.

[4] YARA. ”The pattern matching swiss knife for malware researchers” Available online: https://virustotal.github.io/yara/

[5] Bunten, Andreas. ”Unix and linux based rootkits techniques and countermeasures.” 16th Annual First Conference on Computer Security Incident Handling, Budapest. 2004.

[6] ”Run Android emulator with a custom kernel” Available online: https://gabrio-tognozzi.medium.com/run-android-emulator-with-a-custom-kernel-547287ef708c.

[7] ”What is kernel panic?” Available online: https://www.techtarget.com/searchdatacenter/definition/kernel-panic.

[8] ”Message logging with printk” Available online: https://www.kernel.org/doc/html/next/core-api/printk-basics.html.

[9] P. Bharti, S. S. Roy and A. Suresh, ”Implementation of Yara Rules in Android,” 2023 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 2023, pp. 1-5, doi: 10.1109/ICCCI56745.2023.10128288.

[10] J. Park, H. Chun and S. Jung, ”API and permission-based classification system for Android malware analysis,” 2018 International Conference on Information Networking (ICOIN), Chiang Mai, Thailand, 2018, pp. 930-935, doi: 10.1109/ICOIN.2018.8343260.

[11] N. Naik, P. Jenkins, N. Savage, L. Yang, K. Naik and J. Song, ”Augmented YARA Rules Fused With Fuzzy Hashing in Ransomware Triaging,” 2019 IEEE Symposium Series on Computational Intelligence (SSCI), Xiamen, China, 2019, pp. 625-632, doi: 10.1109/SSCI44817.2019.9002773.

[12] Jannat, Umme Sumaya, et al. ”Analysis and detection of malware in Android applications using machine learning.” 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE). IEEE, 2019.

[13] Yan, Lok K. and Heng Yin. “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.” USENIX Security Symposium (2012).

[14] T. Bhatia and R. Kaushal, ”Malware detection in android based on dynamic analysis,” 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security), London, UK, 2017, pp. 1-6, doi: 10.1109/CyberSecPODS.2017.8074847.

[15] Andronio, Nicolo (2015). Heldroid: Fast and Efficient Linguistic-Based Ransomware Detection. University of Illinois at Chicago. Thesis. https://hdl.handle.net/10027/19676.

[16] Richardson, Ronny, and Max M. North. ”Ransomware: Evolution, mitigation and prevention.” International Management Review 13.1 (2017): 10.

[17] Casolare, Rosangela, et al. ”Dynamic Mobile Malware Detection through System Call-based Image representation.” J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl. 12.1 (2021): 44-63.

[18] ”Linux Kernel Module Rootkit — Syscall Table Hijacking” Available online: https://infosecwriteups.com/linux-kernel-module-rootkit-syscall-table-hijacking-8f1bc0bd099c

[19] Lopez, Juan, et al. ”A survey on function and system call hooking approaches.” Journal of Hardware and Systems Security 1 (2017): 114-136.

[20] Wampler, Doug, and James Graham. ”A method for detecting linux kernel module rootkits.” Advances in Digital Forensics III: IFIP International Conference on Digital Forensics, National Centre for Forensic Science, Orlando, Florida, January 28-January 31, 2007 3. Springer New York, 2007.