MiLK: Minimizing the Linux Kernel for Android Malware

Uriel Antonio Buitrago
uriel.buitrago@utexas.edu

ABSTRACT

MiLK introduces a novel approach to Android security, integrating dynamic runtime protection with static analysis, and harnessing machine learning to construct Secure Computing (seccomp) profiles from provenance analysis. Inspired by advancements in intrusion detection systems [9] and security policy generation, MiLK adapts these methodologies to the Android environment. Utilizing machine learning, it analyzes provenance graphs generated by CamFLow [3], identifying malicious patterns to inform Secure Computing profile (seccomp) construction. This approach employs dynamic analysis to offer a comprehensive security solution tailored to Android's unique application ecosystem and runtime behavior. By embedding machine learning in seccomp profile generation, MiLK enhances the identification and mitigation of kernel exploits, narrowing the Android attack surface. This strategy draws from the foundational work of recent research, including Phoenix [13] discussion on dynamic runtime protection, and SASAK's Android attack surface reduction through seccomp policies [16]. MiLK aims to significantly enhance Android security by proposing deployment as a configurable mobile application. The mobile application is desgined to run a GNN on-device to offer a convenient solution in minimizing the Linux-based Android Kernel.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Android Malware detection, Android Security, Seccomp Profiles, Kernel Surface Reduction, System call analysis

ACM Reference Format:
Uriel Antonio Buitrago. 2024. MiLK: Minimizing the Linux Kernel for Android Malware. In *Proceedings of Mobile Computing (Spring 2024).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobile phones are ubiquitous. From making banking transactions, interacting on social networks, and much more, mobile phones have undeniably contributed to our personal enrichment. Android, being the dominant mobile operating system with over 3 billion users worldwide, has attracted many bad actors who seek to abuse mobile marketplaces such as Google Play. Studies have shown these marketplaces host presumably legitimate apps which actually contain embedded malware [22]. Additionally, Android allows for side-loading code. This means a user can install an application APK from a source that is not a centralized marketplace like Google Play. This greatly exacerbates the threat landscape an attack vectors. In combination with the increasing vulnerabilities in the Android Kernel [16], kernel-targeted attacks initiated through system calls can be perceived as inconspicuous application activity.

System calls (syscalls) are the mechanism through which user-space applications request services from the operating system's kernel, such as file operations, process management, or network communication. When a syscall is made, the application provides a specific request number and parameters; the system then switches to kernel mode to safely execute the request, returning to user mode upon completion. This interface provides a useful attack vector for compromising system security. Because of this, Google introduced a Linux kernel security mechanism named "seccomp" (Secure computing mode) since Android Oreo (v8.0) to constrain the system calls accessible to Android apps. Seccomp (Secure Computing mode) works by enabling a process to define a strict whitelist of allowed system calls. When seccomp is activated for a process, it uses a filter mechanism, typically defined in Berkeley Packet Filter (BPF) syntax, to specify which system calls the process is permitted to execute. Any system calls that are not explicitly allowed by the filter will cause the process to be terminated or the calls to fail, depending on the policy defined in the seccomp filter. This significantly reduces the attack surface by limiting the actions an attacker can perform through exploiting a vulnerable application, thereby enhancing the overall security posture of the system. Seccomp operates in two main modes: strict mode, which is very restrictive and allows only a handful of system calls, and filter mode, which allows more flexibility by letting developers specify exactly which system calls are permitted.

A seccomp profile that restricts an application to only the syscalls necessary for functionality is an effective method of reducing the general attack surface. However, this does not address a more than possible scenario where an exploit is able to leverage syscalls which are also legitimately used by the application. Unpatched vulnerabilities and zero-day attacks are the two primary cases where this could occur. Current solutions include Host Intrusion Detection Systems (HIDS). By analyzing system logs, file integrity, network traffic, and executing process behaviors, HIDS can identify patterns or actions that might signify an attack using an unpatched vulnerability or a zero-day exploit, even before specific signatures for these threats have been developed. Recent contributions in the security field [23] have identified Provenance Tracking as an enhancement to Intrusion Detection Systems. Provenance tracking is the process of recording the history and origin of data as it moves and changes within a system. Provenance graphs represent the flow of information between different subjects (e.g., processes) and objects (e.g., files, sockets, and pipes). It allows for comprehensive auditing and

Spring 2024, ECE382V, Mobile Computing, Austin, TX

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

analysis of data provenance, enabling the identification of how data is accessed, used, and modified.

In the context of Android apps, provenance tracking can be used to monitor and log system call (syscall) traces made by applications, capturing the sequence, parameters, and relations of these calls. This detailed recording facilitates the analysis of app behavior, helping to identify potentially malicious activities or vulnerabilities by examining how apps interact with the system at the syscall level. MiLK seeks to improve upon current advancements in kernel hardening by employing an automated solution to malicious sub-graph classification. Our main contributions are as follows:

(1) Applying seccomp filters to Android applications with an on-device client-server model.
(2) Automating classification of malicious syscall sequences from provenance graphs with Machine Learning.
(3) Dynamic runtime protection for Android.

2 RELATED WORK

In this section, we review literature contributions for system call malware detection and Linux kernel hardening techniques. Since the Android kernel is based on an upstream Linux Long Term Supported (LTS) kernel, we are able to leverage these contributions for the benefit of Android OS.

2.1 Reducing Linux attack surface

Linux containers share a similar isolation and deployment model with Android applications (i.e. DockerHub vs Google Play). Several works have shown that reducing the Linux attack surface makes a big impact when defending against vulnerabilities. Based on the Principle of Least Privilege (PoLP), seccomp policies block a set of system calls that are not normally needed for an application/container to function.

Seyedhamed et al. leverage static code analysis to create custom seccomp profiles for Linux containers, while maintaining their functionality[10]. These profiles blocked 145 or more system calls (out of 326) for more than half of the containers analyzed. And, they blocked 51 previously disclosed kernel vulnerabilities, some of which were Denial of Service and Privilege escalation attacks. A benefit of static analysis is that it is not limited by the code coverage from test cases, a common challenge, and so higher accuracy can be expected. However, it has a time complexity dependent on the code structure of an application, which could make it impractical for mobile devices. Kermabon et. al have introduced a solution, called Phoenix, which seeks to understand system exploitation from summarized provenance graphs, and leverage ptrace to dynamically update a container seccomp profile during runtime [13]. The limitation of Phoenix is that it relies on crowd-sourcing to identify the malicious sub-graph of syscalls and their parameters. MiLK seeks to improve upon this by leveraging machine learning to identify malicious sub-graphs.

2.2 Reducing Android attack surface

Hung et. al propose Sifter, a prototype for protecting security-critical kernel modules in Android [12]. With fine-grained, highly-selective seccomp filters, these kernel modules are made unreachable to untrusted programs. They leverage eBPF programming to deploy stateful syscall filters, and remove domain knowledge overhead of a kernel module syscall interface. Sifter resulted in minimal performance overhead, with only 1.7% increased execution time on macro-benchmarks, and 0.73mAh increase during a six-minute gameplay of a 3D game. Niu et al. developed a seccomp filter solution for app-dedicated control [16]. They dynamically trace the system call invocation of an app with the strace tool, and enforce an architecture-specific seccomp filter by modifying the ForkAndSpecializeCommon function in Android kernel. This function is called by Zygote/Zygote64 process to fork a child process when an Android app is launched. The seccomp filters take effect once the Android apps are started, and enforced with a kernel module which modifies the apps pointer to the secomp structure. For 64-bit apps, they observed a 66.9% reduction ratio in available system calls when compared to the native policy, while introducing negligible impact on the app performance. For 32-bit apps, 22 vulnerable system calls were removed, of which 21 had a CVE security level of MEDIUM or above, and 8 of those were privilege escalation exploits.

2.3 Malware analysis via system call inspection on Android

Previous works [20] [23] have established that syscall execution traces are valuable context for modeling system behaviour to detect malware early. Zhang et al. presents a novel approach to identifying Android malware by analyzing system call traces using a Multi-layer Perceptron (MLP) model [21]. The paper highlights the development of an Android malware detection system (AMDS) that demonstrates high accuracy in early malware detection using traces of the first 3000 system calls, achieving an average accuracy of 99.34%. They also implemented a Client-Server based app that used the MLP model to classify unseen Android apps as benign or malware. Hou et al. also deployed a Machine Learning grounded application, Deep4MalDroid, for early detection of malware [11]. It is a Deep Learning framework for Android Malware Detection Based on Linux Kernel system call graphs. With promising experimental results on data sampled from Comodo Cloud Security Center, they integrated Deep4MalDroid into a commercial Android anti-malware software. However, their framework only consider the names of Linux kernel system calls, while ignoring their parameters. For the case where malware exploits syscalls legitimately needed by other processes in the user space, Deep4MalDroid would fail to classify malicious syscall invocations using outlier parameters. This is why MiLK applies provenance graphs, to account for syscall state transformations with parameters.

3 ANDROID OS MECHANICS

In Android, Zygote is the parent process for all Android application processes. It acts as a part of the Android runtime and is responsible for launching apps. The Zygote process starts at system boot and loads common framework code and resources, (classes and assets) into its memory. When an application is started, Zygote forks itself (using a system call to create a new process), and this child process becomes the process for the new Android application [16]. This results in two types of processes. The apps running on top of the Android runtime environment and the native processes running directly over the kernel. Depending on Android version, the runtime

will be ART or Dalvik VM [1]. Every process is associated with a `task_struct` structure, within which there is a seccomp field detailing the process seccomp status. When seccomp protection is active for a process, its seccomp structure filter attribute links to the initial `seccomp_filter` structure. This structure logs an enumeration of permissible system calls. A process can have several `seccomp_filters` attached, organized in a singly linked list. The enforcement of system call filtering within the Android kernel is enforced by the `BPF_interpreter` [16]. The seccomp facility is a security feature that restricts the set of system calls a process is allowed to make, reducing its attack surface. Seccomp policies, once defined, act as a checkpoint within the kernel, intercepting and validating system calls against these predetermined rules. Transitioning to eBPF programming, this same principle of monitoring and controlling system interactions is expanded, allowing for even more sophisticated and programmable approaches to kernel-level security.

Extended Berkeley Packet Filter (eBPF) is an in-kernel virtual machine that runs user-supplied eBPF programs to extend kernel functionality. These programs can be hooked to probes or events in the kernel and used to collect useful kernel statistics, monitor, and debug. The Android build system has support for compiling C programs to eBPF using simple build file syntax [5]. The eBPF applicability in Android offers a safer and more flexible way to extend kernel functionalities without the risks associated with traditional kernel modules. As eBPF programs are verified for safety and must terminate, they cannot crash the system or run indefinitely, mitigating potential security and stability issues. Their ability to be dynamically loaded and managed from user space simplifies the deployment process, making eBPF a practical choice for Android environments where security and ease of updates are paramount.

4 SECURITY MODEL

I conceptualize the problem of detecting malware during Android app execution as a graph-based outlier detection challenge. The Android application process looks like when the user first opens the app, and interacts with it for a defined period of time. I represent the behaviors observed during an app installation as a series of system events leading to the modification of the system, such as the creation of new files or registry changes. This is encapsulated in an application graph $G = (V, E)$, an attributed directed acyclic graph (DAG), where nodes V represent system entities like processes, files, and sockets, and edges E depict the interactions between these entities. By analyzing a set of known benign application graphs $L = \{G(s_1), G(s_2), \ldots, G(s_j)\}$ from various APKs, the aim is to train a model M that accurately classifies new APK graphs as benign or malicious and identifies any outlier processes (sub-graphs) that exhibit anomalous behavior.

In the context of Android malware, particularly those that use a Command and Control (C2) server to exfiltrate data, the threat model becomes significantly complex. Such malware typically masquerades as a legitimate application, gaining user trust and system permissions upon installation. Once active, the malware covertly accesses and harvests sensitive data, such as contacts, emails, and SMS messages. This harvested data is silently transmitted to a C2 server, often using encrypted channels to bypass standard network security measures. The C2 server serves as the operational hub for the attackers, facilitating not just data aggregation but also the transmission of further malicious commands and updates to the malware [18]. This scenario highlights the acute security risks associated with Android malware, underlining the necessity for advanced detection mechanisms capable of protecting from known and unknown kernel vulnerabilities.

5 MILK ARCHITECTURE

MiLK contains components which run in both the kernel space and user space. The heavy lifting of building syscall traces, constructing provenance graphs, and enforcing seccomp policies all reside within the kernel space as kernel modules. The user space components are the policies and procedures performed on provenance data, packaged in a User Interface.

Figure 1 summarizes the high level architecture and process flow of MiLK. When a new Android application is downloaded, MiLK will monitor the system call invocations that it makes using a custom kernel hook. This kernel hook would be designed to monitor the newly downloaded application at runtime. Similar to strace, it would leverage the ptrace system call to hook into and monitor the system calls that a process makes to the kernel [13]. Because this kernel hook would leverage eBPF programming to identify new processes created by the parent Zygote/Zygote64 process, it would circumvent strace's failure to capture syscalls invoked during startup of an Android app[16]. The kernel hook will constantly monitor in a background thread, and its main purpose is to provide the Finite State Machine (FSM) with a constant stream of syscall activity. This is essential for dynamic runtime protection. A user could configure MiLK's kernel service to start/stop monitoring a certain application.

In a parallel thread, CamFlow would be configured to collect the provenance data of a target app syscall invocations, parameters, data sources, and data sinks (refer to Section 4.2). The provenance data captured in the kernel by CamFlow, would be shared with the camflow daemon (camflowd) in the user space. `camflowd` retrieves records from relayfs pseudo files whcih are also available in Android [4]. The retrieved provenance data would be serialised in a JSON format and sent to a Mosquitto (MQTT) message broker. MQTT is a lightweight messaging protocol designed for small sensors and mobile devices, making it an ideal choice for sharing provenance data with MiLK to generate seccomps (refer to Section 4.3). The seccomp profiles built by the generator employ the Principle of Least Privilege, which means only syscalls necessary for operation are allowed, and the rest are blocked. MiLK creates an entry of these profiles to its database table with an accompanying MD5 hash. These hashes serve as references to deciding if the latest seccomp needs to be updated.

For classifying malicous sub-sequences of sysalls, with their parameters, MiLK sends formatted provenance (JSON graph) to the classification model which employs a Graph Neural Network (GNN) to classify malicious syscall sequences and their parameters. These malicious sequences are passed to the FSM to inform sequence state monitoring, which is updated by the MiLK monitoring service. If a malicious sequence occurs on the device, the user is alerted with a push notification. To account for situations where the malicious

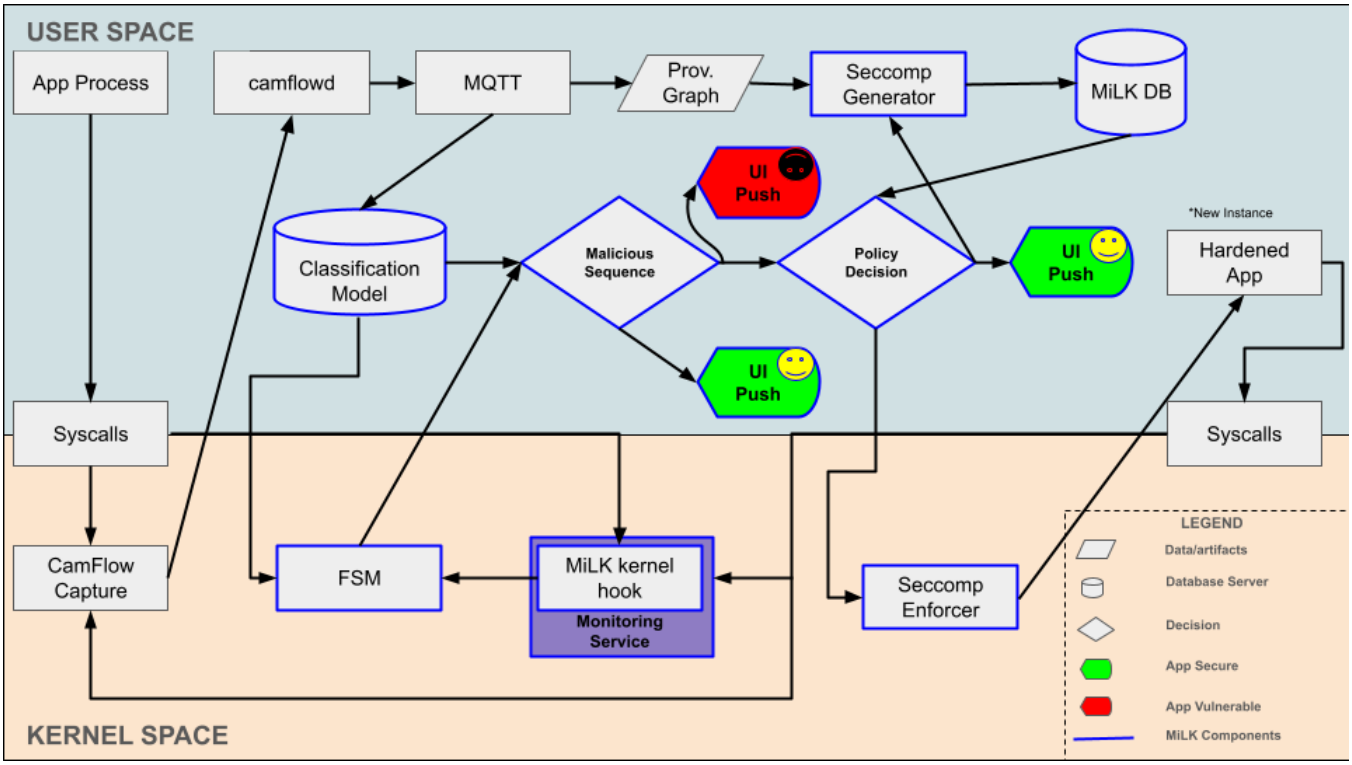

Figure 1: MiLK Architecture and Process Flow.

sequence is interfering with app functionality, users can make policy decisions to safely bypass MiLK's protection. In the scenario that they decide to bypass MiLK, the MD5 hash for the generated seccomp would be updated to include the newly accepted syscalls (at user's risk).

In the scenario that they decided to accept protection, the Seccomp Enforcer would modify the `ForkAndSpecializeCommon` function in the Android kernel. This functions is called by the parent process Zygote to fork a child process when apps are launched. Similar to the approach employed in SASAK, MiLK would generate BPF instructions based on the seccomp filter, and invoke the prctl (or seccomp) system call to enforce the filter for the newly created process (Hardened App in Figure 1). This newly created process would remain under protection by the monitoring kernel service.

5.1 Provenance graphs

In the context of the Android kernel and malware execution, a provenance graph would visually represent the intricate interactions and data flows initiated by malware within the system. This graph would typically include nodes representing various system entities such as processes, files, and sockets, interconnected by edges that delineate system calls and data exchanges. For instance, nodes could illustrate malware processes, hijacked user processes, or sensitive system files, while edges might show actions like file creations, modifications, network connections, or permission changes initiated by the malware. Specifically, such a graph would track the sequence and effect of system calls made by the malware to manipulate the kernel or other processes, potentially highlighting suspicious activities like unauthorized access to system resources or data exfiltration paths to a command and control (C2) server.

Figure 2 is a simple example of a visual mapping which would typically be employed by security analysts to identify malicious patterns of behavior. From the graph, we can see a file reading from a process belonging to password manager. This process also has a connection socket open to an unkown ip, on a insecure port (port 80 which commonly used for HTTP). Another file is reading from the password manager, and loading data onto an existing shell. A single chain-of-events is not particularly suspicious. However, when you combine these change of events, it starts to craft the story that a bad-actor may be stealing credentials, and possibly remotely executing code. This contrived example emphasises the relation that data has to the overall system, and singularly observed events are not sufficient context for identifying attacks. For practicality, MiLK would integrate an Open-Source provenance system, CamFlow [3].

CamFLow is a Linux security module designed for fine-grained, whole-system provenance capture. The module works by using hooks to monitor security access in the kernel space as well as NetFilter hooks to capture network provenance. It can be configured to monitoring target processes and output in JSON format. Although only officially supported on Fedora, it is feasible to extend CamFlow to support mobile environments [17]. Android supports the full SELinux enforcement, and an Android-LSM integration has been demonstrated [19].

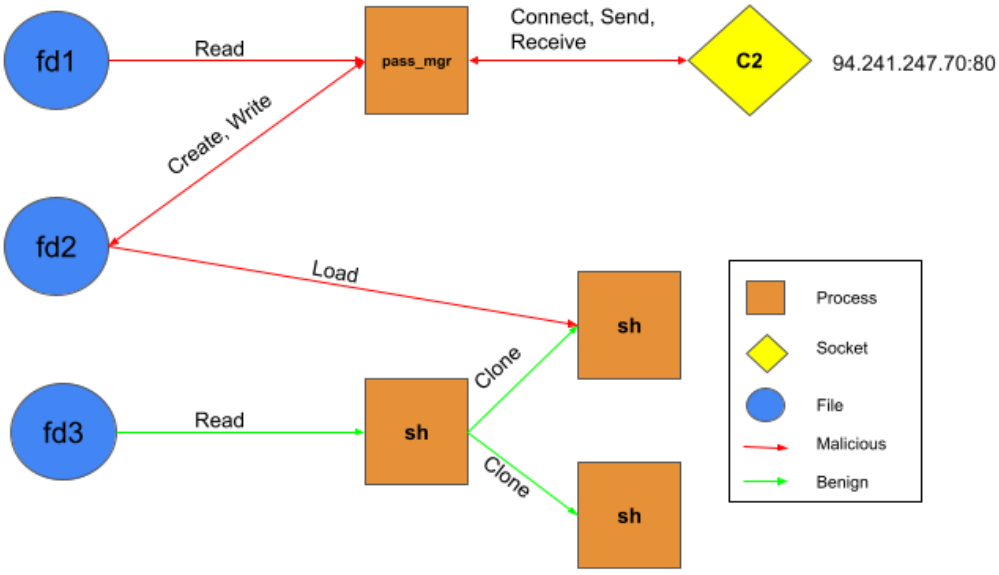

Figure 2: Provenance graph example.

6 CLASSIFICATION MODEL

In this section, I outline how I would go about building a Machine Learning classification model using input provenance graphs.

6.1 Data Processing

If I were to develop a Machine Learning model, I would leverage the CICMalDroid2020 data set [14] [15]. This dataset collected over 17,341 Android samples from December 2017 to December 2018 from various sources such as VirusTotal, Contagio security blog, AMD, MalDozer, and other research datasets. Each APK is categorized into five distinct groups: Adware, Banking malware, SMS malware, Riskware, and Benign. For effective learning, I would split this dataset into training and testing data. I would use 75% of the dataset for training, and the remaining 25% for testing.

Running these APKs for analysis would require an emulator or virutal machine. Given the multitude of emulation solutions, Genymotion is a great option since it is relatively fast, robust, and adopted by millions of users [8]. I would then instrument the emulator with the CamFlow provenance system, and a script to copy out JSON artifacts. Android Debug Bridge (ADB) is a command line tool for communicating with the emulator. I would use ADB to load the APK under-analysis into the emulator. Since we employ dynamic analysis, we are limited by code coverage given there are no manual testers available to interact with all app functionality. Luckily, there exists a powerful development tool, Monkey. The Monkey is a program that runs on the emulator and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events [7]. This is how I would automate the apps execution, and run three times on the same APK to increase coverage. Lastly, I would execute the pre-loaded script to copy out provenance artifacts.

6.2 Model Development

Zhang et al. focused on using system call traces for feature engineering [21]. They use techniques like N-gram models and TF-IDF to capture patterns of benign and malicious behavior. This approach demonstrates that sequential or structured data can effectively train models to classify behavior. A variety of machine learning algorithms, including decision trees, random forest, K-nearest neighbors, naive Bayes, SVM, and MLP, have been used with considerable success. These models could serve as a basis or comparative benchmarks for a GNN-based approach. Provenance graphs naturally lend themselves to graph-based learning.

Graph Neural Networks (GNNs) for classifying malicious subgraphs in provenance graphs is strongly supported by their intrinsic ability to handle graph-structured data, as highlighted by Hou et al. [11]. GNNs excel in capturing local connectivity patterns by aggregating features from a node's neighbors, a crucial aspect for understanding interactions within provenance graphs. Unlike traditional deep learning models, GNNs maintain the natural graph structure during processing, allowing for dynamic learning of node representations. This capability is particularly advantageous for anomaly detection, where understanding the unique and varying structures of each graph is essential for identifying deviations from normal behaviors. Furthermore, GNNs' flexibility in accommodating graphs of varying sizes and structures without the need for fixed input dimensions makes them exceptionally suitable for analyzing the diverse configurations found in provenance graphs. This is not the case for Convolutional Neural Networks (CNNs) or Multi Layer Perceptrons (MLPs). Overall, GNNs provide a robust framework for effectively detecting malicious activities by leveraging both the structural and feature-based information embedded within the graph data.

The following figure represents the process to follow if I were training this GNN.

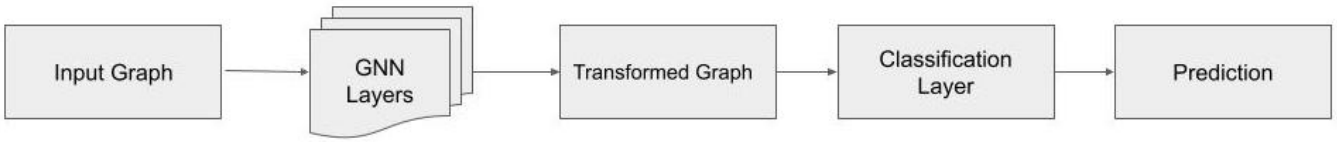

Figure 3: GNN training process.

The initial stage involves constructing the input graph in a manner that nodes embody system entities such as processes, files, and sockets. The edges in this graph will depict the interactions between these entities, which could include system calls with their parameters and API calls. This graph structure allows for the representation of the full context in which system entities operate and interact, thereby capturing the complex behaviors and potential sequences that could indicate malicious activity. The structured input graph is fed into the GNN layers, which process the nodes by aggregating information from their edges (interactions and API calls), thereby embedding the nodes with contextual information about their behavior and the nature of their interactions. After several GNN layers, the graph is transformed into a new representation that embeds the intricacies and nuances of the entities' interactions, highlighting patterns that could differentiate between normal and anomalous behaviors. The classification layer, usually a dense layer with a non-linear activation function, interprets the graph embedding to classify the entire graph or its sub-components as benign or malicious, based on the learned patterns in the transformed graph. A softmax function at the end converts the scores from the classification layer into a probability distribution over possible classes. The final prediction is made by choosing the class with the highest probability, signifying whether the observed interactions in the graph are indicative of malicious behavior.

The model would be optimized using a cross-entropy loss function, which is well-suited for classification problems where the outputs are probabilities. This loss function effectively measures how well the probability distributions predicted by the model match the actual distribution represented by the labels in the training data. Minimizing this loss through an optimization algorithm, such as stochastic gradient descent, will train the model to discern between benign and malicious interactions within the graph accurately.

6.3 Model Deployment

The deployment strategies applied by Hou et al. [11] involve a client-server architectures for real-time classification, which could translate into deploying or integrating the GNN model in a production environment. However, the drawbacks of this approach is that personal data would be broadcast to servers that MiLK users do not have access to. Data leaks from enterprise servers are damaging to brand and reputation. Most importantly, they result in a user's valuable data being compromised.

MiLK, on smartphones, would involve several tailored steps to ensure seamless integration and real-time classification of system activities. MiLK's GNN model would be thoroughly optimized for mobile deployment, using techniques like quantization to reduce the precision of the numbers used in the model, pruning to remove redundant information, and potentially knowledge distillation to create a smaller, more efficient model that retains the knowledge of the original. These techniques help in creating a lightweight version of MiLK suitable for on-device execution.

Once optimized, MiLK would be packaged within a mobile application using frameworks such as TensorFlow Lite or PyTorch Mobile, allowing the GNN to operate directly on users' devices. The mobile application would function as an intelligent system monitor, constructing a real-time graph of system entities and interactions from system calls and API usage and applying MiLK to detect any malicious patterns. This event-driven monitoring ensures that MiLK is activated only when configured, or triggered by the Finite State Machine, to conserve device resources.

For the model to stay current with the evolving landscape of security threats, the application would be designed to periodically update MiLK's parameters through a secure connection (https), ensuring that users always have the latest version capable of detecting new and emerging threats. On-device deployment of MiLK offers privacy advantages by keeping data local, making security a convenience rather than a hindrance. This strategy of deploying machine learning models on devices has seen success in other applications as well. For example, Google's on-device machine learning powers features like Smart Reply in Gmail and predictive text in Gboard, operating effectively in real-time without the need for a server connection [6].

6.4 Evaluations

To evaluate the effectiveness of the GNN in classifying malicious sub-graph sequences, I would use my test set (section 6.1) to evalute the models key performance metrics. Accuracy, precision, recall, and the F1 score are fundamental metrics that would provide an initial understanding of the model's classification ability. For a more nuanced evaluation, particularly important in the context of imbalanced datasets typical of security-related tasks, I would use the Area Under the Receiver Operating Characteristic Curve (AUC-ROC), which measures the model's ability to differentiate between benign and malicious classes. Additionally, a confusion matrix would be vital to break down the true positives, false positives, true negatives, and false negatives, offering detailed insights into the model's predictive behavior. This comprehensive set of metrics would provide a robust framework for assessing the model's statistical performance.

Computational performance and resource efficiency are critical for on-device deployment. I would measure latency and throughput

to ensure the model's predictions are timely and efficient. Memory usage and battery consumption are also critical, as the model must be lightweight enough to not disrupt the phone's normal operations or drain its battery. CPU utilization would be monitored to prevent device slowdowns. Robustness against adversarial attacks and stress testing under heavy load would gauge the model's reliability. Finally, real-world testing, potentially via a beta program, would offer invaluable feedback on the model's performance in operational scenarios, and any impact on user experience would be assessed to ensure that the model's deployment enhances security without diminishing the usability of the device. Through this multifaceted evaluation, MiLK's readiness for deployment and its potential impact on end-users can be comprehensively understood.

7 FUTURE WORKS

To expand on MiLK's effectiveness in classifying malware syscall sequences, I would evaluate different techniques to combat data imbalance. The MalDroid2020 dataset, and most other Android Malware datasets, will contain much more benign APKs than malicious. There is a field of reserach dedicated to balancing datasets, and one promising technique is to leverage a multi-class AdaBoost for GNN. Chen et al had experimental results show that Boosting-GNN achieves better performance than state-of-the-art on synthetic imbalanced datasets, with an average performance improvement of 4.5% [2].

Central to the MiLK framework is the utilization of the CamFlow provenance system. A valuable investigation would be to determine the completeness of the provenance graphs CamFlow produces. Moreover, comparative analyses between CamFlow and other provenance-capable tools such as strace and sysdig would give valuable insight on the overhead provenance tracking results in. This comparison could also help in determining if prevalent, popular linux monitoring technologies offer smoother integration with the Android OS.

8 CONCLUSION

MiLK represents a significant advancement in the field of Android security, offering a novel integration of machine learning and system call analysis to dynamically protect against kernel-level exploits. Its deployment as a mobile application ensures that it can be widely utilized, providing robust security enhancements that are both practical and effective.

REFERENCES

[1] [n. d.]. Android runtime and Dalvik Android Open Source Project — source.android.com. https://source.android.com/docs/core/runtime. [Accessed 09-04-2024].

[2] [n. d.]. Boosting-GNN: Boosting Algorithm for Graph Networks on Imbalanced Node Classification — frontiersin.org. https://www.frontiersin.org/articles/10.3389/fnbot.2021.775688/full. [Accessed 12-04-2024].

[3] [n. d.]. CamFlow: practical whole-system provenance for Linux — camflow.org. https://camflow.org/. [Accessed 01-04-2024].

[4] [n. d.]. Documentation/filesystems/relay.txt - kernel/common - Git at Google — android.googlesource.com. https://android.googlesource.com/kernel/common/+/bcmdhd-3.10/Documentation/filesystems/relay.txt. [Accessed 27-03-2024].

[5] [n. d.]. Extending the Kernel with eBPF | Android Open Source Project — source.android.com. https://source.android.com/docs/core/architecture/kernel/bpf. [Accessed 13-04-2024].

[6] [n. d.]. On-Device Machine Intelligence — research.google. https://research.google/blog/on-device-machine-intelligence/. [Accessed 12-04-2024].

[7] [n. d.]. UI/Application Exerciser Monkey Android Studio Android Developers. https://developer.android.com/studio/test/other-testing-tools/monkey. [Accessed 13-04-2024].

[8] 2024. https://www.genymotion.com/

[9] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z. Berkay Celik, Xiangyu Zhang, and Dongyan Xu. 2021. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3005–3022. https://www.usenix.org/conference/usenixsecurity21/presentation/alsaheel

[10] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458. https://www.usenix.org/conference/raid2020/presentation/ghavanmnia

[11] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. 104–111. https://doi.org/10.1109/WIW.2016.040

[12] Hsin-Wei Hung, Yingtong Liu, and Ardalan Amiri Sani. 2022. Sifter: protecting security-critical kernel modules in Android through attack surface reduction. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking* (Sydney, NSW, Australia) *(MobiCom '22)*. Association for Computing Machinery, New York, NY, USA, 623–635. https://doi.org/10.1145/3495243.3560548

[13] Hugo Kermabon-Bobinnec, Yosr Jarraya, Lingyu Wang, Suryadipta Majumdar, and Makan Pourzandi. 2020. Phoenix: Surviving Unpatched Vulnerabilities via Accurate and Efficient Filtering of Syscall Sequences. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS)*. NDSS, San Diego, CA, USA. Supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair in SDN/NFV Security and the Canada Foundation for Innovation under JELF Project 38599.

[14] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, and Ali A. Ghorbani. 2020. Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. 515–522. https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00094

[15] Samaneh Mahdavifar, Dima Alhadidi, and Ali. A. Ghorbani. 2022. Effective and Efficient Hybrid Android Malware Classification Using Pseudo-Label Stacked Auto-Encoder. *J. Netw. Syst. Manage.* 30, 1 (jan 2022), 34 pages. https://doi.org/10.1007/s10922-021-09634-4

[16] Yingjiao Niu, Lingguang Lei, Yuewu Wang, Jiang Chang, Shijie Jia, and Chunjing Kou. 2020. SASAK: Shrinking the Attack Surface for Android Kernel with Stricter "seccomp" Restrictions. In *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*. 387–394. https://doi.org/10.1109/MSN50589.2020.00070

[17] Thomas F J-M Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. 2017. Camflow: Managed data-sharing for cloud services. *IEEE Trans. Cloud Comput.* 5, 3 (July 2017), 472–484.

[18] Vladimir Radunovic and Mladen Veinović. 2020. Malware command and control over social media: Towards the server-less infrastructure. *Serbian Journal of Electrical Engineering* 17 (01 2020), 357–375. https://doi.org/10.2298/SJEE2003357R

[19] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android.

[20] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro. 2015. *CopperDroid: Automatic Reconstruction of Android Malware Behaviors*. 1–15. https://doi.org/10.14722/ndss.2015.23145

[21] Xinrun Zhang, Akshay Mathur, Lei Zhao, Safia Rahmat, Quamar Niyaz, Ahmad Javaid, and Xiaoli Yang. 2022. An Early Detection of Android Malware Using System Calls based Machine Learning Model. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 9 pages. https://doi.org/10.1145/3538969.3544413

[22] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) *(CODASPY '12)*. Association for Computing Machinery, New York, NY, USA, 317–326. https://doi.org/10.1145/2133601.2133640

[23] Michael Zipperle, Florian Gottwalt, Elizabeth Chang, and Tharam Dillon. 2022. Provenance-based Intrusion Detection Systems: A Survey. *ACM Comput. Surv.* 55, 7, Article 135 (dec 2022), 36 pages. https://doi.org/10.1145/3539605

Received 13 April 2024