DELIVERABLE 2

STATE OF ART IN eBPF APPLIED TO MOBILE SECURITY

Ramya RADJESH

Supervisor: Nicolas DEJON

An eBPF state-of-art report investigating mobile security issues and potentiality of eBPF, researched and submitted to Orange Innovations.

Contents

Date		MODIFICATION-Versions
11.6.2024	Modified by: Ramya Radjesh Supervised: Nicolas Dejon	DRAFT 1 – Presentation Document Structure modification : 1. SECTION 3- Combining Android Security and Kernel Security. 2. SECTION 4- Discussion 3. SECTION 5 – STATE OF ART
18.6.2024	Modified by: Ramya Radjesh Supervised: Nicolas Dejon	DRAFT 1 – Submission Page Count - 38 1. Table 1: Accurate comparison
24.6.2024	Modified by: Ramya Radjesh Supervised: Nicolas Dejon	DRAFT 1 – Correction 1. Addition of new section Problem Statement and Related work. 2. Added a subsection Discussion to section 5. 3. Retransformed the Future directions. 4. Added Current Malware. 5. Pattern format change.
16.7.2024	Modified by: Ramya Radjesh Supervised: Nicolas Dejon	DRAFT 2 submission Page Count - 35
19.7.2024	Modified by: Ramya Radjesh Supervised: Nicolas Dejon	DRAFT 2 – Corrections 1. Adding references and updating Bibliography 2. Changes in the structure of report.
24.7.2024	Modified by: Ramya Radjesh Supervised: Nicolas Dejon	DELIVERABLE 2 - Final Submission

Table 1: Work Modification and Submission details (T1)

Introduction

When a legitimate message appears on your mobile phone asking for permission for a software update, you may grant it without thinking. However, a few days later, you receive an unusual notification indicating unknown login. How can we prevent such scenarios and ensure mobile device security? Features like syscall tracing, runtime detection , kernel level protection, resource access control and event logging and auditing are the operations based on eBPF, a kernel technology that provides all the features, one of the potential approaches to protect a mobile device. It is important to note that it may not inherently protect users from their own mistakes.

In recent time, there are malware, ransomware and adware that are affecting mobile devices through very legitimate ways injecting malicious operations. From the scenario said above, the threat actor can gain root access via permissions that malwares can abuse permissions legitimately granted to an application by the user from the first place.

The document presents a comprehensive analysis of the current landscape of eBPF for mobile security, with a focus on identifying potential research. It explains about mobile related threats and Android related threats and tactics of the threat actors. For this report a research operation was conducted on various research papers and websites that was about eBPF and Mobile security. A discussion on 11 research paper is explained to understand why and how eBPF can help mitigate mobile related vulnerabilities.

The main goal of this report is to understand if eBPF operations that are relevant to mobile security landscape that they help to avoid vulnerabilities, and to give an idea about how eBPF is operated in Linux kernel and its feature that work for the enhancement in mobile security. Here we focus on user space eBPF operations for the development in security aspects in the user side.

1 Mobile Security

Mobile devices contain sensitive passwords, contact information, and banking details. Protecting this information from cyber criminals who may use it for malicious purposes is essential. It has become increasingly important in mobile computing. Currently OWASP is the industry standard for mobile app security. It can be used by mobile software architects and developers seeking to develop secure mobile applications, as well as security testers to ensure completeness and consistency of test results (34). Listed below are the top 10 mobile security issues: (30)

1. Improper Credential Usage
2. Inadequate Supply Chain Security
3. Insecure Authentication/Authorization
4. Insufficient Input/Output Validation
5. Insecure Communication
6. Inadequate Privacy Controls
7. Insufficient Binary Protections
8. Security Misconfiguration
9. Insecure Data Storage
10. Insufficient Cryptography

In this following section, we will explore the nature of mobile malware, including its types, behaviors, and propagation methods, as well as the techniques employed to detect these threats. Additionally, we will examine the threat actors behind these malicious activities and their tactics. Finally, we will discuss future directions in mobile security, highlighting emerging threats and potential solutions.

1.1 Threats in mobile security

Mobile devices are increasingly exposed to malicious activity, particularly malware apps. These attacks can include phishing, supply chain compromise, crypto miner code, and click-fraud advertising. Criminals create apps that act as browser windows to phishing sites, steal user credentials or sensitive data, and include modified versions in factory firmware. Mobile Threats are classified into Application based threats, Web based threats, Network based threats, Physical threats (25).

1.2 Malware

Mobile Malware behaviors of mobile malware

Mobile malware is a type of malicious software designed to target mobile devices like smartphones and tablets, aiming to gain access to private data (6). It can be classified into viruses, worms, Trojans, spyware, backdoors, and droppers. Hackers exploit these vulnerabilities to steal personal and business information, extort money, and steal personal and business data. Mobile malware threats include Android GMBot, AceDeceiver iOS malware, Marcher Android malware, backdoor families, mobile miners, and fake applications. These threats are distributed through repackaging, application updates, and phishing. As smartphones become more advanced, hackers are targeting these devices to spread their malicious capabilities. For example, in Africa according to Statcounter statistics in 2024 (40), more people use mobile devices instead of desktops and tablets that increases the consequence of malware affecting the device.

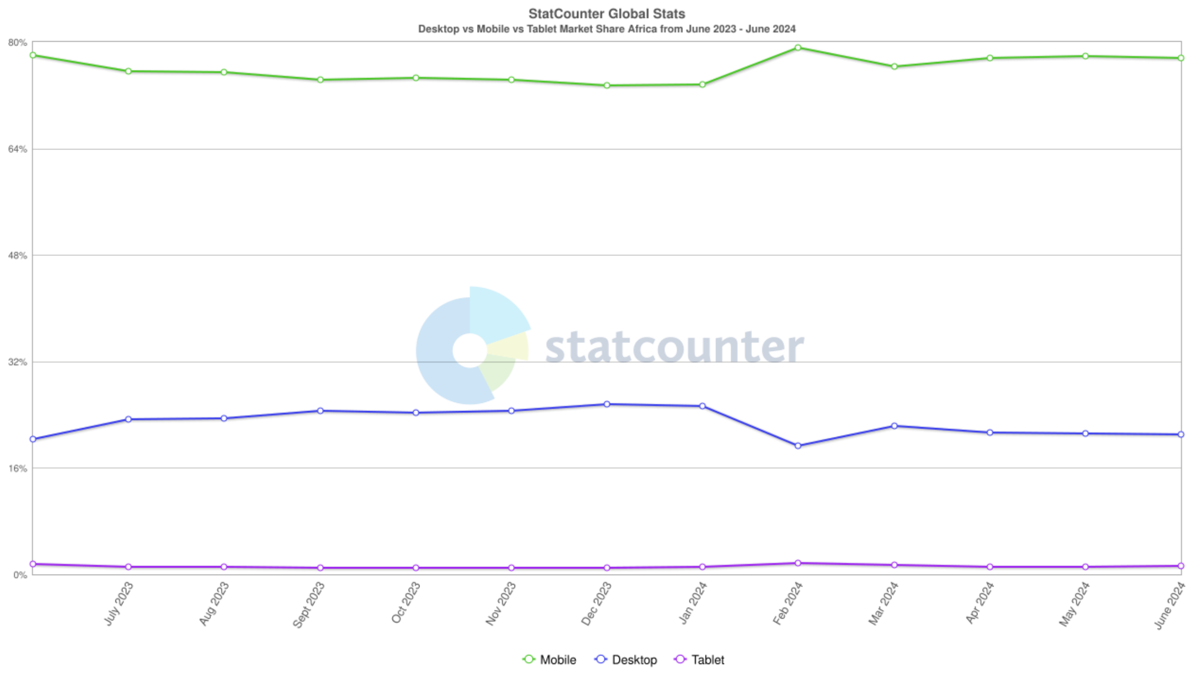

Fig 1: StatCounter statistics for Africa (F1)

Mobile malware can be divided into three groups according to its behavior:

- The **propagation behavior** refers to the access of the malware to the users
  Examples: Malicious apps, Drive by Download and Social Engineering.
- The **remote-control behavior** refers to the use of the remote server,
  Examples: Command Execution, Data Exfiltration and Updates & exploits.
- The **attack behavior** refers to the attacking of the users with different applications after infecting their devices.
  Examples: Data theft, Unauthorized access, financial fraud.

1.3 Malware Injection Techniques

Malware injection techniques are divided into three categories: Process, code and malware propagation that are discussed below.

1.3.1 Process Injection

Malware creates a new process to execute malicious code and compromise a cell phone. This technique is used when user operations are required, like downloading software or opening a message. The new process contains a program descriptor, describing address content, execution state, and security context. This technique is widely adopted due to its simplicity. Malware attacks through legally installed applications, like Symbian and Windows programs.

Examples: Write Process memory, Auto bombing, PE injection, DLL injection etc.

1.3.2 Code Injection

Code injection is the term used to describe attacks that inject code into an application. That injected code is then interpreted by the application, changing the way a program executes. Code injection attacks typically exploit an application vulnerability that allows the processing of invalid data. This type of attack exploits poor handling of untrusted data, and these types of attacks are usually made possible due to a lack of proper input/output data validation. Attackers can introduce (or inject) code into a computer program with this type of vulnerability (9).Major target is Open-Source OS and application frameworks like Android smartphones.

Example : Command Injection, XXE injection, PHP code injection, LDAP injection etc.

1.3.3 Malware Propagation

Malware propagation involves various methods by which malicious software spreads from one system to another, often exploiting vulnerabilities to compromise the target system. Malware propagation model is divided into different types of methods listed below (4).

- **Self-propagation**: Malware like worms can execute and spread without user intervention by exploiting vulnerabilities in network protocols or software.
- **User-interaction**: Malware such as viruses often rely on user actions to initiate execution and spread. They typically attach themselves to legitimate programs or files, which are then executed by the user.
- **Direct pair-wise communication resources**: Malware can propagate through direct communication channels such as Bluetooth, Wi-Fi, the Internet, and Infrared, exploiting these mediums to spread to nearby or connected devices.

1.4 Malware detection techniques

Malware detection techniques are divided into three categories: Static Analysis, Dynamic Analysis and Hybrid Analysis that are discussed below. (36)

1.4.1 Static Analysis

The Static Approach to malware detection is useful when checking for malicious behaviors of a file without executing the code. It does this by disassembling and analyzing the source code. Static analysis will analyze the APK file, User permission and identifying the suspicious code. Static malware analysis has limitations, notably its ineffectiveness against advanced threats that use obfuscation and encryption to hide malicious code. The 2 types of static analysis are Signature based technique and Permission based technique while these 2 techniques are dominant, however there exists many static analysis techniques. Signature-based techniques (38) detect malware patterns, while permission-based techniques (11) check app permissions for genuine needs and malicious purposes. Android security model focuses on denying access to potentially harmful features.

Examples of Static analysis: Androsimilar, Andrubis, APKInspector, DroidMOS, Kirin and Drozer.

1.4.2 Dynamic Analysis

The dynamic approach, also known as behavior-based analysis, uses debuggers, decompilers, and disassemblers to analyze apps during execution. Dynamic analysis (34)is used to overcome the limitations faced by the static analysis where it performs the analysis during runtime. Certain application will reveal their original behavior during it runtime, hence dynamic analysis helps in uncover some features that cannot be determined by static analysis. The common behaviors are system calls, API calls, network traffic or even file access. It can detect obfuscation and encrypted payloads but is resource intensive. Dynamic approach is better equipped to handle obfuscation and encrypted payloads being able to detect polymorphic malware (42) (Polymorphic malware continually changes its features using dynamic encryption keys, making each iteration appear different. Ex: - Strom Worm) and metamorphic malwares (42) (Metamorphic malware evades detection by rewriting its own code with every iteration, making it new and unique from its previous code. Ex: - Virlock is ransomware, file infector, polymorphic algorithm, metamorphic algorithm, and screen locker). Dynamic analysis can be conducted with an emulator or a real device. The dynamic analysis using emulator will be used to run the suspicious APK file to check the application behavior. Emulation, Sandboxing, and honeypots are some of the dynamic analysis techniques for malware detection (18).

Examples of Dynamic analysis: Drozer, TaintDroid, DroidScope, DroidBox, CopperDroid, Crowdroid, Bouncer, Aurasium.

1.4.3 Hybrid Analysis

Combines static and dynamic analysis techniques. It starts by analyzing the signature of any malware code and continue by combining it with other behavioral pattern parameters to enhance malware analysis. Due to this reason, it overcomes both the shortcoming of static and dynamic analysis technique. This increases the ability in detecting malicious software correctly. At the same time, this analysis technique has almost all the strength of static and hybrid technique. In detecting malware in android application, mobile sandbox is an example of hybrid analysis technique. Static analysis will analyze the APK file, user permission and identifying suspicious code. While the dynamic analysis using emulator will be used to run the suspicious APK file to check the application behavior.

STATIC	DYNAMIC	HYBRID
Android application is analyzed by parsing its code without executing it.	Android application is executed in controlled environment	Android application is analyzed using both static and dynamic analysis
Static Feature: Permission, intent, opcode etc.	Dynamic Feature: API Calls, System Calls etc.	Both static and dynamic features
Scan the code using various parsers	Execution environments; debuggers, simulators, emulators, virtual machines	Polyunpack: hidden codes compared with runtime execution + instances with runtime codes
Computationally expensive	Poor performance with trigger based malware	Sometimes computationally very expensive
Better overall code coverage	Poor/limited coverage problem	Good overall code coverage
Inability due to undecidability, lack of support for runtime packages, limitation to obfuscation	Efficient in analyzing packed malware, more time consuming, requires more resources than static	80:40 ration makes it more efficient as compared to both the approaches

Table 2: Comparison among malware detection process (T2)

To understand more about the malware detection techniques, refer (36)

1.5 Threat Actors

Threat actors also known as cyberthreat actors or malicious actors are individuals or groups that intentionally cause harm to digital devices or systems (28). The threat actors are on the top layer of the ecosystem. They can be categorized in multiple ways, depending on their skills, motivation, target, and modus operandi.

Threat actors deploy a mixture of tactics when running a cyberattack, relying more heavily on some than others, depending on their primary motivation, resources and intended target. The tactics of threat actors are:

1. **Malware** - help threat actors steal data, take over computer systems and attack other computers.
2. **Ransomware** - These attacks are double-extortion attacks that also threaten to steal the victim's data and sell it or leak it online.
3. **Phishing & Social Engineering** – These attacks use email, text messages, voice messages or fake websites. The threat actors are often motivated by financial gain.
4. **Denial of Service attack** - DoS attacks are usually carried out by flooding the target with traffic or requests until it can no longer handle the load and becomes unavailable. They can also be used to disable systems or networks by corrupting data, taking advantage of vulnerabilities, or overloading resources and steal data from victims.
5. **Advanced Persistent Threat (APT)** - APT actors often take advantage of zero-day vulnerabilities in software or hardware that have recently been discovered but not yet patched. By exploiting the vulnerabilities before they've been addressed, threat actors can easily gain unauthorized access to target systems. The purpose of APT is to stay under the radar for a long time.
6. **Backdoor attacks** - A backdoor attack occurs when threat actors create or use a backdoor to gain remote access to a system.

1.5.1 Types of threat actors

Threat actors or CTA (Cyber Threat Actors) are often categorized into different types based on their motivation and to a lesser degree, their level of sophistication (28). Refer (Fig 2) that explains threat profiling, capability & commitment of the threat actors. The types of threat actors are listed below:

- Cybercriminals: Financial gain
- Nation-state actors: Espionage & Financial gain (ex. North Korea)
- Hacktivists: Activism
- Thrill seekers: Amusement
- Insider threats: Insider
- Cyberterrorists: Terrorism
- Script Kiddies: Exploration or curiosity

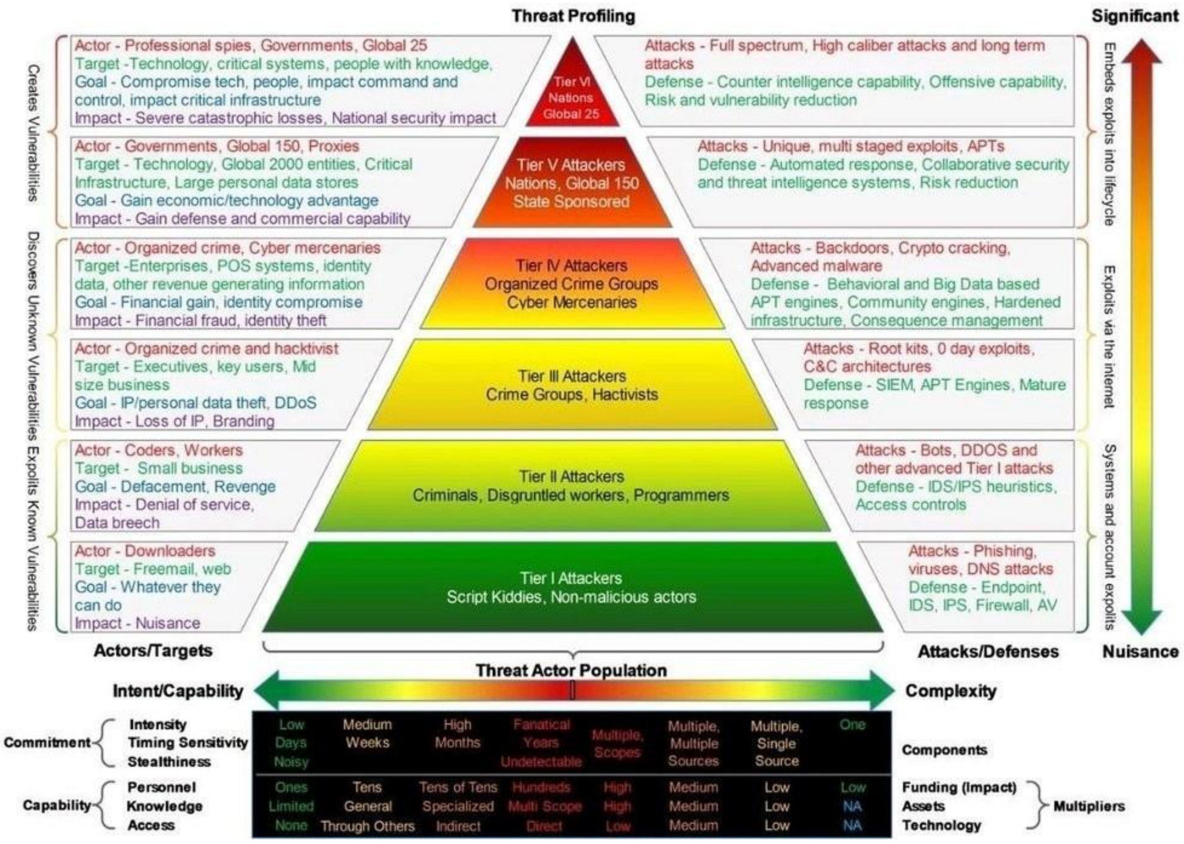
Threat Profiling
Significant
Actor - Professional spies, Governments, Global 25
Target -Technology, critical systems, people with knowledge,
Goal - Compromise tech, people, impact command and control, impact critical infrastructure
Impact - Severe catastrophic losses, National security impact
Tier VI Nations Global 25
Attacks - Full spectrum, High caliber attacks and long term attacks
Defense - Counter intelligence capability, Offensive capability, Risk and vulnerability reduction
Actor - Governments, Global 150, Proxies
Target - Technology, Global 2000 entities, Critical Infrastructure, Large personal data stores
Goal - Gain economic/technology advantage
Impact - Gain defense and commercial capability
Tier V Attackers Nations, Global 150 State Sponsored
Attacks - Unique, multi staged exploits, APTs
Defense - Automated response, Collaborative security and threat intelligence systems, Risk reduction
Actor - Organized crime, Cyber mercenaries
Target -Enterprises, POS systems, identity data, other revenue generating information
Goal - Financial gain, identity compromise
Impact - Financial fraud, identity theft
Tier IV Attackers Organized Crime Groups Cyber Mercenaries
Attacks - Backdoors, Crypto cracking, Advanced malware
Defense - Behavioral and Big Data based APT engines, Community engines, Hardened infrastructure, Consequence management
Actor - Organized crime and hacktivist
Target - Executives, key users, Mid size business
Goal - IP/personal data theft, DDoS
Impact - Loss of IP, Branding
Tier III Attackers Crime Groups, Hactivists
Attacks - Root kits, 0 day exploits, C&C architectures
Defense - SIEM, APT Engines, Mature response
Actor - Coders, Workers
Target - Small business
Goal - Defacement, Revenge
Impact - Denial of service, Data breech
Tier II Attackers Criminals, Disgruntled workers, Programmers
Attacks - Bots, DDOS and other advanced Tier I attacks
Defense - IDS/IPS heuristics, Access controls
Actor - Downloaders
Target - Freemail, web
Goal - Whatever they can do
Impact - Nuisance
Tier I Attackers Script Kiddies, Non-malicious actors
Attacks - Phishing, viruses, DNS attacks
Defense - Endpoint, IDS, IPS, Firewall, AV
Creates Vulnerabilities
Discovers Unknown Vulnerabilities Exploits Known Vulnerabilities
Embeds exploits into lifecycle
Exploits via the internet
Systems and account exploits
Actors/Targets
Attacks/Defenses
Nuisance
Threat Actor Population
Intent/Capability
Complexity
Commitment
Intensity
Timing Sensitivity
Stealthiness
Capability
Personnel
Knowledge
Access
Low Days Noisy
Medium Weeks
High Months
Fanatical Years Undetectable
Multiple, Scopes
Multiple, Multiple Sources
Multiple, Single Source
One
Ones Limited None
Tens General Through Others
Tens of Tens Specialized Indirect
Hundreds Multi Scope Direct
High High Low
Medium Medium Medium
Low Low Low
Low NA NA
Components
Funding (Impact)
Assets
Technology
Multipliers

Fig 2: Cyber Threat Profiling: understanding the different threat actors (F2)

1.6 Future Directions in malware detection: challenges and solution

When it comes to combating advanced strategies like obfuscation, polymorphism, and zero-day attacks, conventional solutions frequently fail. The need to provide reliable tools and techniques that can successfully resist these attacks grows as malware developers continue to hone their tactics. This section highlights several important areas where further study and development will be necessary to improve malware mitigation and detection. The reference for this section is from (5)

1. Outrageous reverse engineering techniques: The need to make reverse engineering and obfuscation technique more difficult for malware authors who use repackaging methods to add malicious code to Android applications is important. One way to achieve this is by using encryption algorithms to make the code of an application opaque and harder to reverse engineer.

2. Extensive hybrid analysis tool: A robust and efficient tool is needed to analyze applications and notify users if they contain obfuscated code that may be malicious. Obfuscation is a technique used by malware to hide itself and evade detection. Additionally, malware authors use polymorphic techniques and insert dead code into the original application code to further avoid detection. Since static analysis can detect obfuscated code but it is more difficult, dynamic analysis becomes essential at this stage to identify malicious code.

3. Solution of Code coverage problems: Dynamic analysis can be effective in detecting dynamically loaded code and native code, which may not be covered by static analysis. However, it is important to note that achieving full code coverage with dynamic analysis is challenging. Malware writers often choose complex execution paths to evade detection mechanism. Therefore, in the future, a robust hybrid analysis approach should be adopted to analyze all versions of the android OS with comprehensive code coverage. This will help in developing more effective solutions for detecting and mitigating malware, especially in the user space where dynamic malware is typically found.

4. Tool to detect zero-day attacks: The requirement is for highly intelligent and lightweight procedures capable of examining the behavior of applications at runtime to identify zero-day attacks. Existing methods are complicated and time-consuming for malware detection. However, developing such detection procedures is challenging and requires optimistic research directions.

5. Identical datasets with updated malware families: As time flies, there is a striking increase in android malwares and its variants. Hence, the requirement is standardization and updated datasets to perform malware detection analysis in an efficient way. Moreover, restructured, and new simplified datasets is helping to detect emerging malwares that are increases with every passing day. Now it's essential to develop multilevel procedures (eg. Network analysis, Heuristic and Machine Learning Analysis, etc.) that deals with advance malware attacks and capable to detect and mitigate malicious mobile applications.

6. Laws and regulations: Governments should make legislations to deal with information security along with cybersecurity related attacks and enforce them at international level. Recently, European law CRA aims to enhance the cybersecurity of digital products and services within the European Union and Federal Bureau investigation (FBI) declared cybersecurity law for internet of things (IoT) devices in which they state that "if you found any toy is being compromised security terms then you have to report it." Such methods need to be implemented in mobile devices security.

7. Cybersecurity insurance: The emerging rate of malware attacks such as distributed denial of services (DDos), phishing, ransomware, or any other ways that causes financial loss to individuals. Consequently, proper insurance structure should be applied on every scale of business that will surely mitigate the risk factor. Moreover, cybersecurity insurance covers cost of accidental attacks and hardware damage.

8. Big data challenge: As big data familiarized with special malware detection challenges and machine learning techniques are unable to deal with big data.

Therefore, required efficient deep learning methods that overcome this issue and reduce the computational overhead with affordable cost.

2 Android Security

2.1 Android and Linux Security

Android

Android is developed by Google at first and then via Open Handset Alliance (OHA) is promoted (29). The Android platform is built on top of the Linux kernel, comprising APIs, libraries, and middleware written in the C programming language. Above this layer lies the application framework, where application software runs, including additional libraries compatible with Java. Security is a top priority in the development of the Android operating system, with the goal of providing a secure environment for users (Fig 4).Android apps are written in Java and kotlin. Multiple layers of security measures are implemented to protect against potential threats. Regular security updates and patches are released to address any identified vulnerabilities promptly. Furthermore, Android devices are equipped with built-in security features like encryption and secure boot, safeguarding user data and preventing unauthorized modifications to the system. These comprehensive security measures work together to create a secure and reliable platform for Android users.

Linux Kernel Security

The Android kernel is based on an upstream Linux Long Term Supported (LTS) kernel (39). A kernel has a protected kernel Region, a trusted memory space that loads the kernel's code, ensuring security and preventing other applications from accessing it. It manages process, file system & memory management, security, and network operations. A customized embedded Linux system interacts with the phone hardware. The middleware and application API runs on top of Linux.

Security Architecture

Characteristics:

- Customized Linux kernel for limited resource embedded environments.
- Android developed on top of Linux kernel for robust driver model, efficient memory and process management, and networking support.
- Supports ARM and x86 instruction architectures.
- Native C/C++ libraries support high-performance third-party libraries.
- Java code translated into Dalvik byte code for efficient resource utilization.
- Before Dalvik is used, now replaced by ART (Android Runtime).
- Application framework layer provides uniform view of Java libraries.
- Runs sensitive functionality as system services, protected with permissions.

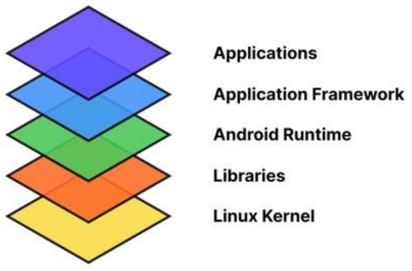

Fig 3: Layers of Android Mobile design (F3)

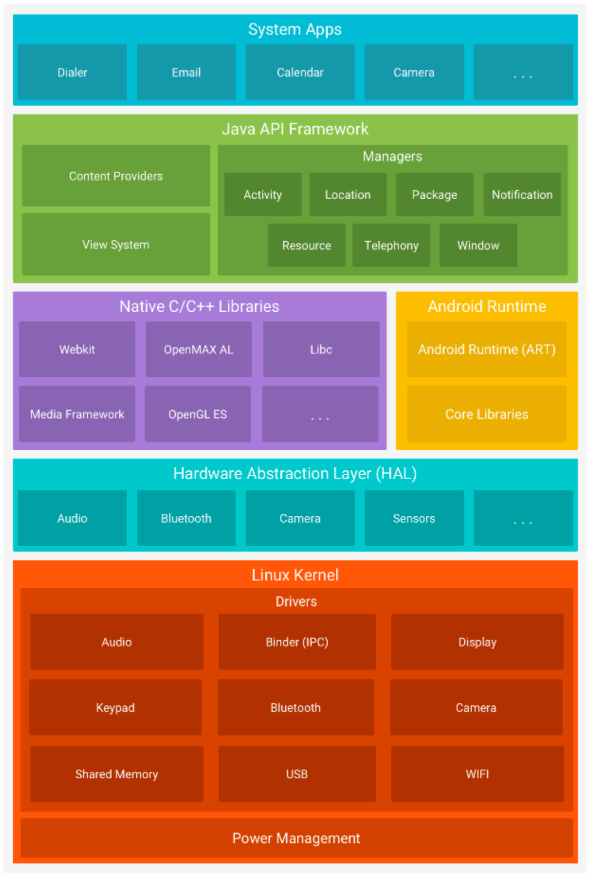

Fig 4: Android Security Architecture (F4)

2.2 App Structure

An Android application is available in (Android Package Kit) APK form. APK file consist of lib, Assets, Resources, Dalvik Bytecode, Files, Manifest and Res. Refer to the Fig.5.

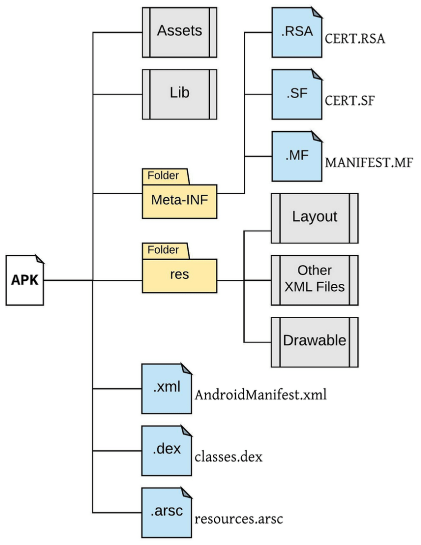

Fig 5: Android App structure (F5)

- APK .apk – a zip archive consisting of several files and folders.
- AndroidManifest.xml – stores the meta data such as package name, required permissions, external libraries, and platform requirements.
- Res – stores icons, images, strings/numeric/color constants, UI layouts, menu, animations compiled into the library.
- Assets – non-compiled resources.
- Classes. Dex – Stores Dalvik bytecode to be executed on the Dalvik VM.
- META-INF – stores the signatures of the app developers' certificate to verify the third-party developer identity.

2.3 Android Security Features

2.3.1 Android application Sandboxing

Application sandboxing is also called as application containerization. It is an approach to Mobile Application development and Management that limits the environments in which certain code can execute (19). Android applications run in an isolated area of the system, known as a sandbox, that does not have access to the rest of the systems resources, unless and until the access permissions are explicitly granted by the user during the application installation. To protect the applications data from unauthorized access, the Android kernel implements the Linux Discretionary Access Control (DAC) to manage and protect the devices resources to be misused. Each application process is protected with an assigned unique ID (UID) within an isolated sandbox (19).

2.3.2 Permissions at framework-level

Android provides a permission-based security model at the application framework level to restrict apps from accessing crucial smartphone functionality. Permissions are coarse-grained, and apps must declare them using tags in the AndroidManifest.xml file. There are different types of permissions (12):

Install-time permission, Normal Permissions, Signature permissions, Runtime Permissions (location and contact information), Special Permissions.

Android permissions are divided into following four protection-levels: normal, dangerous, signature and internal. Normal permissions have minimal risk, while dangerous permissions allow access to private data and device features. Users must grant these permissions before installation. Signature or system permissions are granted automatically. The internet permission does not restrict access to URL domains, while READ_PHONE_STATE allows phone ringing and in-hold checking. Users are often unable to judge permission appropriateness, exposing themselves to risks.

2.3.3 Security Kernel Module

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. A module can be configured as built-in or loadable.

The Linux Security Modules (LSM) project has developed a lightweight, general purpose, access control framework for the mainstream Linux kernel that enables many different accesses control models to be implemented as loadable kernel modules (20). LSM allows modules to mediate access to kernel objects by placing hooks in the kernel code just ahead of the access, as shown in (Fig 6). LSM merely provides the fields and a set of calls to security hooks that can be implemented by the module to manage the security fields as desired. The currently accepted security modules in the mainstream kernel are AppArmor, bpf (for eBPF hooks), integrity, LoadPin, Lockdown, safesetid, SELinux, Smack, TOMOYO Linux, and Yama.

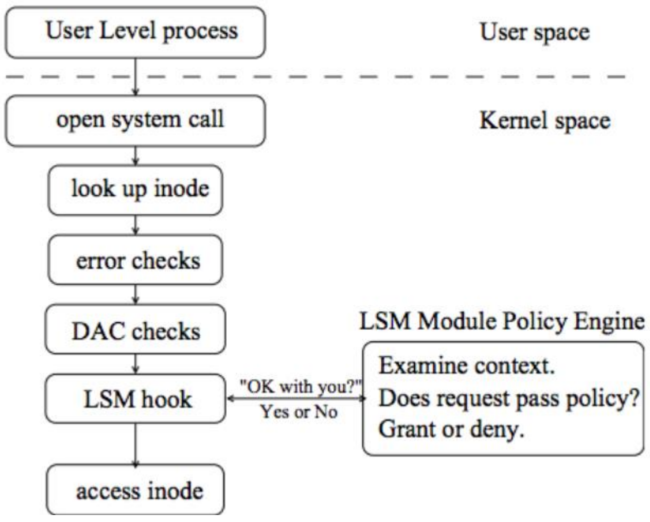

Fig 6: LSM Hook Architecture (F6)

In LKM there are two different ways to load the kernel modules.

- Insmod: To load a module into the kernel.

- modprobe: (To detect any dependent module required) and load it.

eBPF code is organized into compact units called programs. Each program is attached to a specific function named hook point and is executed in a non-preemptable fashion every time the hook is reached. There are several types of hook point both in kernel space and in user space. Valid examples are (41): system calls, kernel tracepoints, network events, function entry/exit points, and LSM hooks. Specifically, LSM hooks correspond to the functions used by LSMs (e.g., SELinux) to perform security decisions and are characterized by operating entirely on arguments in kernel memory.

SELinux

SELinux provides several types of enforcement, such as type enforcement, role-based access control, and multi-level security. Authorization decisions are based on a policy, which is loaded from a file. SELinux can limit a process's access to files even if it runs with the root user. A set of kernel modifications and user-space tools that have been added to various Linux distributions. Its architecture strives to separate enforcement of security decisions from the security policy and streamlines the amount of software involved with security policy enforcement (37). SELinux is not an antivirus solution and replacement of passwords, firewall, and other security solution (35) .

Benefits:

1. SELinux is designed to protect against misuse and unauthorized use such as: Unauthorized reading of data and programs, unauthorized modification of data and programs. Bypassing application security mechanisms.
2. Android uses SELinux to enforce mandatory access control (MAC) over all processes, even processes running with root/superuser privileges (Linux capabilities).
3. SELinux can be used to enforce data confidentiality and integrity, as well as protecting processes from untrusted inputs (35).
4. SELinux policy is administratively defined and enforced system wide.

2.3.3.1 Syscall Security

syscall () is a small library function that invokes the system call whose assembly language interface has the specified number with the specified arguments.

A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel.

Syscall interacts with the kernel modules follow rich set of patterns, encoded in the module itself, often through a set of ioctl syscall. Example syscalls are read, write, open, execve, chown, clone. The security of system calls, which are the interface between user-space applications and the kernel. The Syscall security aspects are.

- Access control (Permissions, Privileges),
- Input Validation (Sanitization, Bound checking),
- Syscall filtering (Limiting, Reducing attack surface),

- System Call Auditing (Monitoring, Logging),
- Kernel Hardening (Security patches, Configuration).

2.4 Android Malware threats and Weaknesses

In this section, listed below are the threats to android devices that discusses about the various malicious activity that are carried out on the android and other mobile device environment and the examples under each threat.

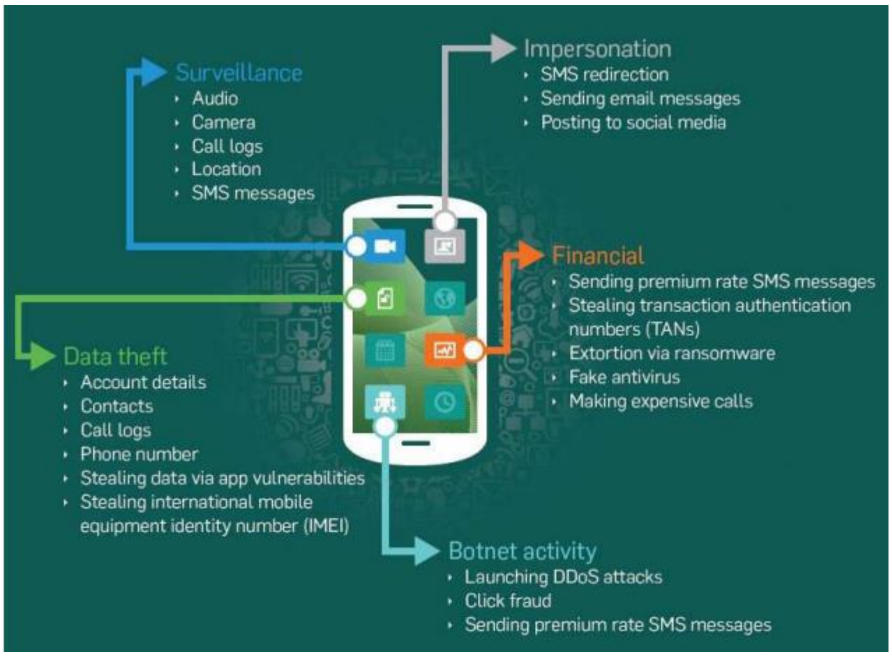

Fig 7: Mobile Malware Evolution (F7)

Threats to Android Devices

Threat 1: Premium Rate Calls and SMS

Charges victims' phone bills by sending premium calls and SMS. The hacker doesn't get any money themselves.

Example: Fake player charges victims by sending the message “798657” to multiple premium numbers.

Threat 2: Botnets

Controls a group of Android devices under one botmaster. Gives bots commands to perform various tasks. Botnets can intercept information during transactions.

Examples: Brute Force attack, Click fraud, Remote Desktop Protocol (RDP) attack, Zitmo.

Threat 3: Ransomware

Ransomware can cause loss of device access until the ransom amount is paid. No guarantee that the device will return to normal after payment.

Examples: Locky, GandCrab, Cryptowall, Crypto ransomeware, Petya, Dearcry.

Threat 4: Trojan

Trojans usually marries as benign apps but performs harmful activities without user consent. They leak confidential user information, phishes, and steals sensitive information. Majority of Android variants belong to various SMS trojan families. Malware authors target two-factor mobile banking authentication.

Examples: FakeNetflix, Fakeplayer, Zsone, and Android.Foney, BianLian, MoneytiseSDK, FreeCoin, GodFather, Droiddream, Android.Bmaster, AnserverBot, TigerBot, Master Key, Fontal, Liberty crack DownAPK.

Threat 5: Backdoor

Backdoor attacks often involve the use of spyware or malware. Allows other malware to bypass security procedures using root exploits.

Examples Basebridge, KMin, Obad.

Threat 6: Worm

Worm apps can create identical copies and spread them through network or removable media. Covert method of bypassing security restrictions to gain unauthorized access to a computer system. Bluetooth worms can exploit Bluetooth functionality and send copies to paired devices.

Example: Mydoom, Blaster, Conficker, code red, email worms.

Threat 7: Spyware

Surreptitiously monitors contacts, messages, location, and bank mTANs. Spyware can send collected information to a remote server.

Examples: Idshark, Gptrap, SamsungBrowserSpy, SoftPhoneMonitor, BouldSpy, LaSurv, InfamousChiselSurv, StatisticalSales, RedDrop.

Threat 8: Riskware

Riskware is any potentially unwanted application that is not classified as malware but may utilize system resources in an undesirable or annoying manner, and/or may pose a security risk.

Example: RiskySigner, SourMint, Virtualization, Unsafe Market.

Threat 8: Aggressive Adware

Misuse of Android's location services by affiliate networks for personalized advertisements. They can create shortcuts, steal bookmarks, change default search engine settings, and push unnecessary notifications.

Examples: HummingWhale, CopyCat, Shedun, Kemoge, GhostPush.

3 eBPF

eBPF (extended Berkeley Packet Filter) is a technology with its origins in Linux kernel () that allows users to run code in the Linux kernel without changing kernel source code or loading kernel modules. Initially BPF programs were tailored for networking tasks, its extended version eBPF has now ventured into various kernel subsystems enabling applications in performance monitoring, security enforcement, and surpassed classic BPF. eBPF operates by allowing pre-compiled programs to be executed within an event-driven framework in response to kernel and userspace events, all under the vigilant scrutiny of a verifier ensuring the kernel's stability and security. eBPF functions like a sandbox virtual machine within the kernel, executing code securely and efficiently, thus extending the kernel's capabilities without direct modification. Modern eBPF development is facilitated by the presence of frontends. These frameworks permit to write eBPF programs in a C dialect and assist the developer in automatically performing the steps needed to load and attach the programs to the intended hooks (26). one interesting feature of eBPF is that it is also present and protects peripheral devices such as network cards and USB drive (10).

eBPF is used extensively to drive a wide variety of use cases:

- Providing high-performance networking
- load-balancing in modern data centers
- cloud native environments
- extracting fine-grained security observability data at low overhead
- helping application developers trace applications
- providing insights for performance troubleshooting
- preventive application
- container runtime security enforcement, and much more.

Architecture of eBPF:

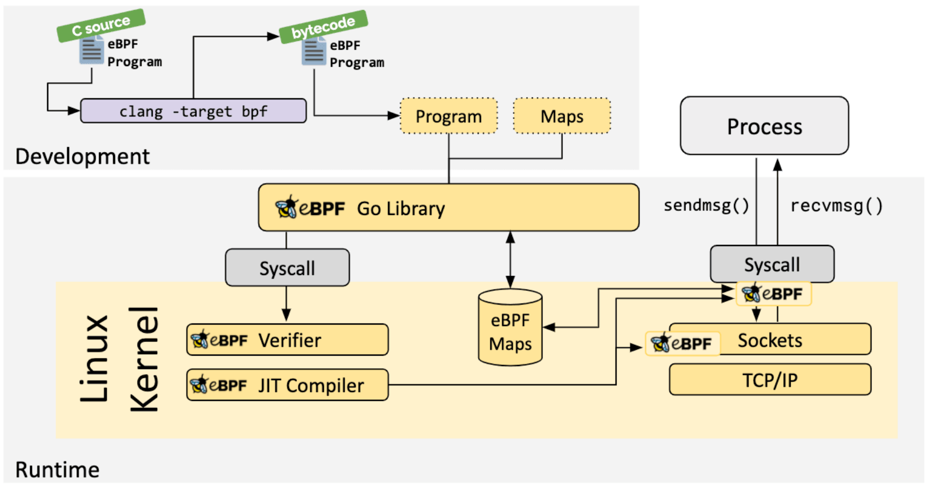

Fig 8: eBPF components and architecture (F8)

3.1 BPF to eBPF in Security

BPF

BPF programs are primarily used for packet inspection, offering significant performance enhancements. Those programs are used for packet inspection, offering significant performance enhancements. It manages packet copying from kernel to user space, using an interpreter and JIT compiler.

Security Perspective:

From a security point of view, BPF(43) provides basic safety mechanisms, mainly focused on preventing packet filter crashes. Offers a basic level of safety due to its simplicity and the restricted nature of the virtual machine where BPF programs run.

eBPF

eBPF allows for the creation of secure programs and filtering applications, allowing them to be appended directly to kernel code without redesigning or rebooting the system (). It is not limited to kernel applications and can be executed during system running time. Unlike classic BPF, eBPF can be injected between hardware, kernel, user space, and kernel space boundaries.

Security Perspective:

Comparing with BPF, eBPF (43) offers Advanced safety features, including program verification to prevent harmful actions in the kernel. In Application, they are ideally used for comprehensive system monitoring, security enforcement, and performance profiling.

Fig 9 shows the architecture of eBPF and BPF and their differences. BPF consists of two registers: accumulator and index register, an implied program counter, and temporary auxiliary memory. On the other hand, eBPF was expanded from 2 to 11 registers, the size of registers increased from 32 bits to 64 bits, and a stack of size 512 bytes was introduced in the eBPF engine. There are no more limitations to the data size or structure due to a global data map added to the architecture of eBPF.

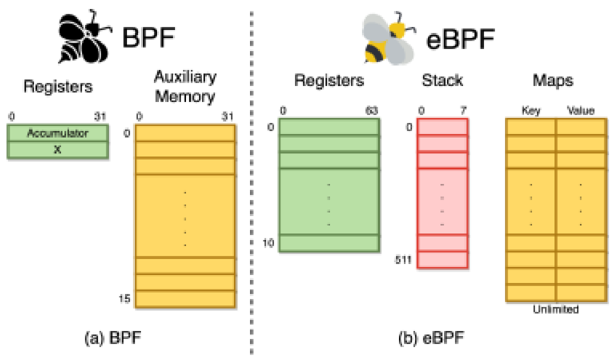

Fig 9: Evolution of Register and Memory Architecture in BPF and eBPF (F9)

3.2 How eBPF works?

To understand how eBPF works, knowing about the components and their functionalities in the main architecture (Fig 1) is important. To know about the tools and libraries of eBPF refer (14).

eBPF Components	Functionalities
eBPF Maps	eBPF programs can leverage the concept of eBPF maps to store and retrieve data in a wide set of data structures. eBPF maps can be accessed from eBPF programs as well as from applications in user space via a system call.
eBPF Verifier	eBPF verifier ensures memory safety, scrutinizing programs from issues like accessing beyond bounds or reading uninitialized memory. This verification process mitigates security risk, promoting overall kernel safety. (32)
JIT Compiler	Compilation & convert generic bytecode into machine specific instruction. Helps eBPF Program to run efficiently as natively compiled kernel code or loaded as kernel modules.
eBPF Helper Functions	eBPF programs cannot call into arbitrary kernel functions. Allowing this would bind eBPF programs to kernel versions and would complicate compatibility of programs. Instead, eBPF programs can make function calls into helper functions, a well-known and stable API offered by the kernel.
eBPF Kprobes & Uprobes	Kprobes, short for Kernel Probes, enables developers to insert probes into the kernel code at runtime. These probes can be attached to specific kernel functions or even to individual instructions (15). Unlike kprobes or tracepoints, uprobes focus on user-space instrumentation. They allow for the insertion of probes into user-space applications, rather than kernel code. Uprobes can be attached to user-space functions; when these functions are executed, the associated probe handlers are triggered (15).

Table 3: eBPF components and Functionalities (T3)

eBPF programs are initially written in a restricted C subset and then compiled into eBPF bytecode using tools like LLVM (Low Level Virtual Machine), which serves as eBPF's back-end architecture for front-end programming languages like Clang (23). eBPF program execution is event driven. An event can be a system call, function entry, exit, perf event, Kprobes, and Uprobes. The input context to the eBPF program varies based on the hook/event and tracepoints that are attached. eBPF programs are loaded into the kernel using the bpf syscall (5). After being loaded, each program undergoes a two-phase process comprising program verification and JIT compilation. The eBPF Verifier is required to guarantee that the program is safe to execute by the kernel. The second phase instead ensures the bytecode is optimized; hence it can be run as efficiently as compiled kernel code on the underlying architecture. In case no errors are raised, the eBPF program is attached to the proper hook and it is ready to be executed.

3.3 Why eBPF for Mobile Security? Why eBPF for Mitigating mobile malware vulnerabilities?

In recent times, the vulnerability count has been increased along with the development in the different versions of the Android devices. eBPF is a powerful concept operating inside Linux kernel and android uses Linux as it core kernel. The main capability of eBPF is syscall tracing and network traffic analysis and monitoring, with this capabilities spyware like Pegasus (1), ransomwares like Phobos (31) and Rafel RAT (3) and SMS malware like Flubot (20) be mitigated. The main eBPF components that are useful to mitigate is the eBPF programs and the eBPF maps. For example, In BPFroid detects malware in android devices, detection of convert channel works on the detection of malware in network. Also, eBPF programs can be used to monitor syscall and enforce security policies to detect and prevent many malicious activities. Additionally, eBPF can be employed to implement network filters and perform deep packet inspection to identify and block malicious traffic associated with any malicious activity.

3.4 eBPF in Userspace:

The eBPF program (47) source is compiled using toolchains like clang and loaded from a user space eBPF application to the kernel via the BPF syscall, aided by libraries such as libbpf. Post verification, the BPF Just-In-Time (JIT) compiler transmutes the BPF bytecode to machine code for execution. The user space eBPF application also orchestrates eBPF maps to steer the kernel eBPF behavior, aggregate or encapsulate the kernel BPF data, and relay information from the kernel to the user space. Moreover, it can gather BPF programs to events like Kprobes, socket, syscall trace points, Uprobe, and more, furnishing a robust mechanism for real-time interaction and data retrieval.

4 Problem Statement and Literature Review

4.1 Methodology

The goal of this report is to assess the potential of eBPF based implemented solutions on mobile device to mitigate the possible vulnerabilities. The research was conducted on research papers to understand the present and gaps in the future developments of eBPF in mobile security. The keywords were “mobile security” “eBPF”, “Android Security”, “malware mitigation”, “mobile device”. These keywords were selected to capture a wide range of relevant studies. After finding the total research paper count was 90. Out of which the most focused area of eBPF were “networking security” & “cloud security”. To align with the objective, after filtering the count of research papers the finalized count was 11. Where they focused on eBPF in mobile security landscape and on Linux based storage, patch-based software updates and detection of malicious activities.

Listed below are the research paper that were selected based on research operation that came under “mobile security” “eBPF”.

Problem Statement:

This report is to address various vulnerabilities regarding mobile device. The goal of the report is to evaluate the potentials of eBPF for mobile security by researching in content that are available on the research papers which are security framework and features suitable for mobile device. eBPF is already used by android as important kernel feature. Here discussion about various features of eBPF is explained to understand how eBPF works in the mobile security field and how far it can prevent malware getting injected in device.

4.2 Analysis on Research papers

LBM: A security framework for peripherals within the linux kernel (2019)

LBM (10) discusses the use of eBPF in mobile security, focusing on security and packet filtering. The LBM tool is a security framework built on eBPF, providing efficient and extensible protection against malicious peripherals. It uses a high-level language for filtering functionality. LBM addresses USB attacks and offers host protection. LBMTOOL compiles rules into eBPF instructions. Evaluation shows minimal overhead and outperformance of other solutions. LBM is a practical and comprehensive framework for the Linux kernel peripheral subsystem.

BPFroid: Robust Real Time Android Malware Detection Framework (2021)

BPFroid (45) is an Android malware detection framework using eBPF technology. It traces system calls, kernel functions, and Android framework API calls in real-time. It enhances device security by detecting suspicious behaviors and capturing forensic artifacts. BPFroid provides context, filters events, and allows data manipulation. It supports ARM64 devices and offers strong security and application-to-OS interaction tracing. Limitations include elevated privileges, untraceable API functions, and limited tracing of API function arguments. Future work includes policy enforcement, system image integration, and detection of hardware-based attacks.

Detecting convert channel through code augmentation (2021)

The paper investigates the use of code augmentation with eBPF to detect covert communications (27). It focuses on detecting covert channels targeting IPv6 traffic. The approach involves monitoring kernel functions and evaluating changes in specific fields of incoming packets. The evaluation shows that changes in certain fields can indicate the presence of a hidden channel. The approach is lightweight and introduces minimal overhead.

Limitations include its applicability to other types of covert channels and communication methods. Future work includes refining the approach and evaluating its performance, as well as developing threat-independent metrics and automatic alarm mechanisms. Overall, the approach offers a foundation for detecting and mitigating covert communications.

Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction (2022)

Sifter (22) is a solution for protecting security-critical kernel modules in Android using eBPF. It reduces the attack surface through fine-grained filters, making vulnerabilities unreachable for untrusted programs. Sifter generates and deploys filters using eBPF with tracepoint and syscall templates. Its dynamic approach yields stricter filters with minimal performance impact. The goal is to prevent malware from exploiting kernel module vulnerabilities through syscalls. The approach is not limited to Android and can be applied to other devices with the Linux kernel. Limitations include manual interaction and access constraints in Android. Future work includes automating trace collection, integrating with other security techniques, and exploring deployment on other systems. Sifter's filters mitigate about half of syscall-triggered vulnerabilities with no false positives and minimal performance impact.

NCScope: Hardware-Assisted Analyzer for Native Code in Android Apps (2022)

NCScope (21) is a hardware-assisted analyzer for native code in Android apps that utilizes eBPF and the ETM hardware feature of the ARM platform. It retrieves and stores memory data, marks data sources, and adds timestamps. NCScope collects real execution traces and memory data to analyze self-protection, anti-analysis mechanisms, memory corruption, and performance differences in native code. The architecture includes an ARM module for data collection and an analysis module to minimize overhead. Limitations include potential missed instructions and the possibility of app detection. Future work aims to address these limitations and expand capabilities. NCScope has proven effective in analyzing self-protection, anti-analysis methods, memory corruption, and performance in financial and malicious apps.

RapidPatch: RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices (2022)

RapidPatch (46) is a hotpatching framework for real-time embedded devices that addresses challenges in timely patching and vulnerability management. It utilizes eBPF virtual machines and achieves over 90% vulnerability mitigation. The system works on devices with limited resources and demonstrates low latency overhead. RapidPatch allows generic patches without disrupting tasks and supports different microcontroller architectures. It uses eBPF file patching, eBPF helper functions, and Clang for patch generation and verification. The framework aims to provide a common hotpatching runtime for embedded devices, allowing patches to be written and deployed separately without modifying the firmware.

Bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions (2023)

Bpftime (47) is a userspace eBPF runtime that enhances the performance of Uprobe and Syscall hooks. It provides programmatic Syscall hooking, shared memory maps, and a smooth interface with popular tools. Bpftime can be seamlessly attached to any running process without the need for restart or recompilation. It utilizes eBPF functionality to enable efficient interactions with the host environment. The runtime ensures stability and reliability by isolating libraries and namespaces. Bpftime improves performance, compatibility, and runtime efficiency for userspace eBPF applications. Mitigation strategies are in place to address potential security risks. Future work includes expanding platform support and refining performance.

eXpress Data Path extension for high capacity 5G user plane functions (2023)

The eXpress Data Path (XDP) (8) extension for high capacity 5G user plane functions aim to enhance packet forwarding and classification in the GPRS Tunneling Protocol (GTP-U). It includes improvements such as XDP enhancements, Receive Side Scaling mechanism, XDP program for GTP-U traffic forwarding, and Packet XDP implementation for fast packet forwarding. The UPF with XDP enhancements reduces overhead and improves throughput. Future work includes exploring XDP-hardware offloading for even higher bandwidths in 5G core networks.

Programmable System Call Security with eBPF (2023)

The Programmable System Call Security with eBPF (24) extension aims to enhance system call filtering in Linux's Seccomp module. It introduces a new Seccomp-eBPF program type that leverages the extended BPF language (eBPF) to express advanced security policies. The extension includes helper functions, filter operations, syscall filtering, and maps. It enables more expressive and stateful security policies while maintaining the existing usage model and interfaces. The evaluation shows improved security, reduced attack surface, and accelerated filters. Future work could focus on automatic generation of Seccomp-eBPF filters and integration with other security mechanisms.

NatiSand- Native Code Sandboxing for Javascript Runtimes (2023)

NatiSand (26) is a tool that provides native code sandboxing for JavaScript runtimes, using Landlock, eBPF, and Seccomp to control filesystem, IPC, and network resources. It mitigates security risks associated with potentially malicious or vulnerable native code components in web applications. NatiSand's architecture includes a sandboxer component, eBPF programs and maps for security context information, and a context pool for thread pre-allocation. It enables fine-grained access control to system resources, allowing transparent execution of code in isolated compartments with policy-based ambient rights. Developers define permissions in a JSON policy file for different security contexts associated with the application, including file access, process communication, and network connections. NatiSand provides a CLI utility in Go for policy creation and automates network policy generation through eBPF program observation. Future work includes enhancing network and IPC permission control and expanding support to other operating systems.

Mitigating IoT Botnet DDoS Attacks through MUD and eBPF based Traffic Filtering (2024)

The paper (2) proposes a novel system that uses MUD and eBPF-based traffic filtering to mitigate IoT botnet DDoS attacks. The system includes an osMUD manager, eBPF-IoT-MUD adapter, and iptables firewall adapter. The evaluation results show that these techniques are effective in protecting against attacks with minimal impact on legitimate traffic. The paper also discusses future work, including deployment on actual systems and identifying optimal rate limit values.

5 Classification and Commonalities

In this section, discussion about various eBPF based operations that are performed in the above discussed research paper takes place. This comparison below involves the research papers that are discussed above with the commonalities that they have in their functions along with the common patterns followed in all these research papers.

eBPF MAPS

eBPF maps is used to do many operations like security enhancement, efficient lookup (monitoring), sharing data and storage. The below listed research papers deals with eBPF maps for security enhancement and storage.

In **Mitigation of IoT Botnet DDoS Attacks through MUD and eBPF based Traffic Filtering** (2), eBPF maps play a crucial role in enhancing the security of IoT networks by enabling efficient traffic filtering and rate-limiting operations. These maps act as key-value stores where the keys represent specific communication flows, defined by attributes such as source and destination IP addresses, ports, and protocol types. The values associated with these keys store flow statistics, including the number of packets and bytes transmitted, as well as predefined rate limits for these metrics. When a packet arrives, the eBPF program parses its headers to construct a key and then looks up this key in the allowlist maps. If the key is found, the program updates the flow statistics and checks them against the rate limits. If the traffic exceeds these limits, the packet is dropped, thereby preventing potential security threats such as Distributed Denial-of-Service (DDoS) attacks. This real-time monitoring and enforcement ensure that IoT devices adhere to their specified communication patterns, mitigating the risk of unauthorized access and excessive traffic generation. By leveraging eBPF maps, the system achieves high-speed, low-overhead security operations, making it effective for protecting IoT networks.

In **Sifter** (22), eBPF maps are used for several specific operations to enhance security. These operations include logging syscall numbers, arguments, and timestamps into eBPF maps for detailed inspection. They also involve performing deep copies of syscall arguments from user space memory into eBPF maps. Additionally, eBPF maps cache syscall arguments to prevent Time-of-Check to Time-of-Use (TOCTOU) attacks. Finally, eBPF maps implement a locking mechanism to serialize operations and prevent race conditions, using custom eBPF helper functions such as `bpf_lock_fd`, `bpf_unlock_fd`, and `bpf_wait_if_fd_locked`. These specialized uses of eBPF maps go beyond their native operations, which typically involve basic key-value storage and retrieval.

In **Bpftime** (47), this usersapce eBPF runtime enables secure, unprivileged access to kernel eBPF maps by using the BPF filesystem (BPFFS). A privileged process first creates and pins eBPF maps to BPFFS. The bpftime-daemon then manages these maps and exposes them to unprivileged processes. Unprivileged processes can use bpf_obj_get() to retrieve file descriptors for these maps and perform operations like update, lookup, and delete without needing elevated privileges. This reduces security risks associated with broad administrative access. The eBPF runtime in userspace can only access necessary maps and cannot create new maps, load new programs, or attach to kernel events, ensuring secure map accessibility.

Pattern	Mitigation of IoT Botnets DDoS Attacks	Sifter	Bpftime
Key-Value Storage	For monitoring traffic flows and statistics.	For monitoring syscall details and arguments.	For managing unprivileged access via BFFS.
Attack mitigation	Drops packets exceeding limits (i.e) DDoS Attack	Prevents TOCTOU attacks with the helper functions	Restrict maps access, isolates memory.

Table 4: Comparison for eBPF maps security and storage operation

PROGRAM VERIFICATION

eBPF verifier is for verifying the programs or patches. Here in the below listed research papers eBPF verifier is used in to check the eBPF program from harmful behavior such as unbounded loops which could compromise the target system

In **RapidPatch** (46), the developers have customized the use of eBPF verifier as Patch safety verifier to ensure patch safety. In the architecture of RapidPatch, there is an element called the patch generator compiles the eBPF source code into eBPF bytecode and analyzes the firmware symbol files to decide the correct patch installation address and obtain the global addresses (e.g., global variables, functions, or basic block addresses) that are required by the eBPF code. Given the eBPF bytecode patches, the device maintainers can use a tool named patch verifier to verify the safety of the patch by identifying harmful behaviors such as illegal access of kernel memory or excessive loop iterations. Patch verification happens offline. Linux eBPF verifier does verification in runtime. If patch is safe then it is deployed in the device directly using SFI (Software-based isolation) approach for both eBPF interpreter and eBPF JIT compiler. Note that Linux's original eBPF verifier cannot be directly applied here, since it performs very strict checks on memory accessing scope and loops. Even the filter patch may have to break these rules, the eBPF verifier is customized to verify the patch safety.

In **bpftime** (47), the userspace verifier plays a crucial role in ensuring the safety and integrity of eBPF programs. eBPF is not designed as a sandbox; instead, it minimizes runtime checks by verifying programs before execution to ensure high performance. This verification can be performed using either the kernel eBPF verifier or a userspace verifier. These verifiers conduct comprehensive checks, including the validation of BPF bytecode, to guarantee that eBPF programs do not compromise the target process. The verifier checks the eBPF bytecode for adherence to the instruction set, performs control flow analysis to ensure termination, validates memory and type safety, enforces resource limits, and ensures correct usage of helper functions.

Pattern	RapidPatch	Bpftime
Verifier type	Customized eBPF verifier for patch safety	Kernel eBPF verifier or user pace equivalent
Additional security	Focuses on the correct patch installation address and global address	Multi-layer security architecture to prevent unauthorized access.

Table 5: Comparison in eBPF verifier processes

System Calls:

When eBPF is used in a mobile device, it can trace all layers of the android stack (4 layer can be traced – Internal kernel function, system call, native library function and java API framework. All syscall operations are attached with eBPF programs. In syscall operation, on the perspective of security, there are two main functions that system call process helps in protection of the mobile device. One is any changes of values or adding any code that happens in the file in the android system, that is being traced and alert given to user. Second is the protection of kernel operations and security critical operations by limiting the number of system call invocation and checking for the privilege escalation.

BPFroid (45) **and Bpftime** (47) – uses syscall to protect the files and other kernel modules. In both the research papers, the tracepoints, hooks and probes are used with syscall operations. Syscall tracepoints can be set to react or change system call parameters, return values, allows detailed and controlled system interactions. In BPFroid, to detect droppers, vfs_write kernel function which alert the user with the device ID, pathname of the written file, and inode number if any write operations are made in the file. Also, Android has unique UID that remains constant during the runtime to write simple rules and to check if the application has privilege escalated. After a process of android is forked, developers use PID as a key to an eBPF maps and save its UID as the value. Between 2 subsequent system call the value is check if the value is changed, alert is given to the user with the pathname and file that is modified. In Bpftime, syscall compatible library is there in the user space to monitor daemons and tracers typically written in libbpf. eBPF program checks for the syscall parameter to make decision according to the rule and even changes the values. In these two research papers, the operation of eBPF program differs but both uses syscalls to mitigate from malware or other malware related vulnerabilities.

Difference:

Research paper	Pattern – syscall operation for monitoring
BPFroid	eBPF programs to monitor and detect malware
Bpftime	eBPF programs are used to trace and modify syscall parameter if required to maintain detailed controlled system interactions.

Table 6: BPFroid and Bpftime comparison for syscall monitoring

Programmable System Call Security with eBPF (24) **and Natisand** (26) – uses additional Linux security modules to enhance security like Landlock, Seccomp, Seccomp Notifier. In both research paper they use seccomp commonly. In Programmable System Call Security with eBPF, all the syscall related operations are associated with the user space (trusted) agent. Seccomp Notifier is here to capture syscall before getting executed, to block and redirects syscalls context such as process ID, call ID and argument values to user space agent to take security decision. In Natisand, this paper deals with the protection of native code with seccomp and landlock. Seccomp is used here to provide fine-grained control over the native code by minimizing attack surface and enhancing the security. Seccomp is used to filter syscalls to restrict specific operations that are based on configuration flags which is crucial for enforcing granular security policies. Landlock is used to provide fine-grained control over the filesystem access. It allows the authors to define specific ruleset that restricts the actions a program can perform on the filesystem. It operates in a stackable manner, allowing it to be combined with other Linux Security Modules (LSMs) like Seccomp and eBPF for comprehensive security. The combination of seccomp and landlock ensures the native code executed within the JavaScript runtime is confined to a secure environment with limited privileges.

Difference:

Research paper	Pattern – Seccomp operations
Programmable System Call Security with eBPF	Seccomp operations are associated with userspace operation for security enhancement.
NatiSand	Seccomp & Landlock operates in stack to protect and manage filesystem and its permissions.

Table 7: Research paper comparisons for seccomp operations

<u>Nativecode execution and protect:</u>

In the two research papers, native code is the common pattern. Integrating native code into high-level environments boosts performance but raises security concerns. NatiSand and NCScope provide tailored solutions for different platforms using distinct security methods.

NCScope (21) **and NatiSand** (26)– in both the research papers they use native code for the reason of security enhancing with eBPF. In NatiSand, Integrates native code execution within JavaScript runtimes (e.g., Node.js, Deno). This allows JavaScript applications to leverage high-performance native libraries and binary programs. In NCScope, integrates native code execution within Android apps using the Java Native Interface (JNI). This allows Java applications to call and be called by native libraries written in C or C++. Both papers employ policy-based mechanisms to control and secure the execution of native code. In NatiSand, they use JSON-formatted policies to specify ambient rights for each native component, ensuring fine-grained control over system resources. In NCScope, uses offline symbolic execution and behavior identification to enforce security policies and detect malicious behaviors in native code. Based on security, this chose different environment to give distinct security approaches. In NatiSand, Addresses the security risks associated with native code execution by sandboxing it. This includes controlling access to the filesystem, IPC, and network resources using Landlock, eBPF, and Seccomp. In NCScope, it focuses on analyzing and detecting security threats in native code, such as memory corruption vulnerabilities and anti-analysis techniques. It uses hardware-assisted features like ETM and eBPF for detailed execution tracing and memory data collection.

Pattern	NatiSand	NCScope
Targeted environment	Javascript runtime (Deno, Node.js)	Android app
Security mechanism	Landlock, eBPF and Seccomp	ETM and eBPF
Policy Enforcement	JSON formatted policies	Offline symbolic execution and behavioral identification to enforce security policies.

Table 8: Comparison of security features in NCScope and NatiSand

<u>Userspace operations</u>

In all four research papers—LBM, Detection of Covert Channels, NatiSand, and Programmable System Call Security with eBPF—a common pattern that emerges from a security perspective is the management of policies in user-space.

In **LBM** (10)(Linux (e)BPF Modules), the user-space tool LBMTOOL performs several critical operations. It parses LBM rules written in a high-level Domain-Specific Language (DSL) and compiles them into eBPF programs. These compiled programs are then loaded into the kernel. LBMTOOL also manages the lifecycle of these eBPF programs, including loading, attaching, and detaching them from the kernel. Additionally, it configures LBM filters and modules, ensuring they are correctly deployed to protect against malicious peripherals. The user-space component monitors the performance impact of LBM filters to maintain system efficiency.

In the **Detection of Covert Channels** (27) paper, the user-space utility is responsible for configuring eBPF programs and managing their lifecycle. This includes loading, attaching, and detaching eBPF programs from specific kernel hooks. The utility collects data from these eBPF programs, including metrics and statistics about network packets and system calls. This collected data is then analyzed in user-space to detect anomalies that may indicate the presence of covert channels. The utility also monitors the performance impact of the eBPF programs to ensure that the overhead introduced is minimal.

In **NatiSand** (26), the user-space component is responsible for parsing the JSON-formatted policy file provided by the developer. Based on this policy, it initializes sandboxing and tracing programs and sets up a pool of isolated contexts (threads) with restricted permissions. During runtime, NatiSand intercepts calls to native code and assigns them to the appropriate pre-allocated isolated context. It ensures that the correct security context is applied to each native code execution, enforcing the restrictions specified in the policy. This dynamic management allows for fine-grained control over native code execution.

In **Programmable System Call Security with eBPF** (24), the user-space operations involve loading eBPF programs into the kernel using the `bpf ()` system call and managing their lifecycle. This includes configuring system call filters and deploying them to enforce security policies. The user-space component also monitors the performance impact of these eBPF-based system call filters. By ensuring that the system call filters are correctly applied, the user-space operations dynamically restrict or monitor system call usage, reducing the attack surface and preventing exploitation of vulnerabilities.

By managing these policies in user-space, each solution provides a flexible and powerful mechanism for dynamically enforcing security measures.

Research papers	Userspace operation
LBM	Parsing and compiling policies, loading eBPF programs, managing filter lifecycle, monitoring performance
Detection of convert channel via code augmentation	Configures eBPF programs, initializing data collection, analyzing data for anomalies, managing eBPF lifecycle, monitoring performance
NatiSand	Parsing policy files, initializing sandboxing and tracing programs, intercepting native code calls, managinf context lifecycle
Programmable syscall security with eBPF	Loading eBPF programs, configuring syscall filters, enforcing system call filters, manging filter lifecycle and monitoring performance.

Table 9: Comparison of User space operations in security

Name	BPFroid	NCscope	Bpftime	Sifter	LBM	Express Datapath	Detection of convert channel	Mitigation of IoT Botnets	Natisand	Rapidpatch	Programmable Syscall Security
Syscall	●		●						●		●
eBPF Maps			●	●				●			
User-space operations					●		●		●		●
Program verification			●							●	
Native Code Protection		●							●		

Table 10: eBPF operational capabilities being common in the research papers.

Conclusion

In conclusion, this document presents a thorough investigation into the current landscape of eBPF for mobile security. By analyzing of 11 research papers and identifying common patterns in the eBPF functionalities that was implemented in the research papers. The results indicate that eBPF offers advanced capabilities for monitoring, analyzing, and enforcing security policies directly within the kernel. This report helps us understand how eBPF is being used in innovative ways to address security challenges, like malware attacks, alerting users to unauthorized access, setting security policies, and user space operations that reduce potential vulnerabilities towards mobile devices. These patterns are what highlight the new ways that eBPF is being used. By continuing to explore and expand the applications of eBPF, we can further enhance the security and resilience of mobile devices against evolving threats.

Bibliography

1. Andrew Zola, L. R. (2024, May). *Pegasus malware*. Retrieved from TechTarget: https://www.techtarget.com/searchsecurity/definition/Pegasus-malware

2. Angelo Feraudo, D. A. (2024). *Mitigating IoT Botnet DDoS Attacks through MUD and eBPF based Traffic Filtering.* ACM Digital Library.

3. Antonis Terefos, B. M. (2024, June 20). *RAFEL RAT, ANDROID MALWARE FROM ESPIONAGE TO RANSOMWARE OPERATIONS*. Retrieved from cpr by checkpoint: https://research.checkpoint.com/2024/rafel-rat-android-malware-from-espionage-to-ransomware-operations/

4. Ashwini Gour, J. P. (2014). *Modelling & Detaining Mobile Virus Proliferation over Smart phones.* India: International Journal of Computer Science and Information Technologies.

5. Attia Qamar, A. K. (n.d.). *MOBILE MALWARE ATTACKS: REVIEW, TAXONOMY & FUTURE DIRECTIONS.* China.

6. Baker, K. (2023, November 3). *WHAT IS MOBILE MALWARE?* Retrieved from CROWDSTRIKE: https://www.crowdstrike.com/cybersecurity-101/malware/mobile-malware/

7. Chris Wright and Crispin Cowan, S. S.-H. (2002). *Linux Security Modules: General Security Support for the Linux Kernel.* San Francisco, California, USA: USENIX Association.

8. Christian Scheich, M. C. (2023). *eXpress data path extensions fpr high capacity 5G user plane function.* ACM Digital Library.

9. Contrast security. (n.d.). *Code Injection* . Retrieved from Contrast Security: https://www.contrastsecurity.com/glossary/code-injection#:~:text=Code%20injection%20prevention-,What%20is%20code%20injection%3F,the%20way%20a%20program%20executes.

10. Dave Jing Tian, G. H. (19-23 May 2019). *LBM: A Security Framework for Peripherals within the Linux Kernel.* USA: IEEE Xplore .

11. David Barrera, H. G. (2010). *A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android* . Canada: ACM.

12. Developers, A. (2024, June 18). *Permissions on Android* . Retrieved from Android Developer: https://developer.android.com/guide/topics/permissions/overview

13. Dimitriou, H. S. (January 2022). *Extended Berkeley Packet Filter: An Application Perspective.* ResearchGate.

14. eBPF. (2024). *eBPF Submit 2024*. Retrieved from eBPF: https://ebpf.io/applications/

15. eBPF. (n.d.). *eBPF*. Retrieved from eBPF: https://ebpf.io/what-is-ebpf/

16. eBPF. (n.d.). *eBPF*. Retrieved from eBPF Submit 2024: https://ebpf.io/infrastructure/

17. *eBPF Overview - Learn how eBPF helps simplify access to kernel space operations without sacrificing security.* (n.d.). Retrieved from DATA DOG Knowledge Center: https://www.datadoghq.com/knowledge-

center/ebpf/#:~:text=Kernel%20space%20protects%20memory%20and,before%20allowing%20it%20to%20execute.

18. Faria Nawshin a, b. R.-A. (2024). *Malware detection for mobile computing using secure and privacy-preserving machine learning approaches: A comprehensive survey.* Qatar: ELSEVIER.
19. Fung, B. R. (n.d.). *A Survey of Android Security Threats and Defenses.* Richmond, Virginia, USA.
20. Grant, C. (2022, July 7). *What Is FluBot SMS Malware? Is It Really Gone? How to Get Rid of It?* Retrieved from ENEA: https://www.enea.com/insights/flubot-malware-gone-but-for-how-long/
21. Hao Zhou, S. W. (2022). *NCScope: hardware-assisted analyzer for native code in Android apps.* Hong Kong: The Hong Kong Polytechnic University.
22. Hsin-Wei Hung, Y. L. (2022). *Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction.* Sydney, NSW, Australia: ACM Digitam Library.
23. IBM. (n.d.). *What is eBPF?* Retrieved from IBM: https://www.ibm.com/topics/ebpf#:~:text=eBPF%20programs%20are%20initially%20written,end%20programming%20languages%20like%20Clang.
24. Jinghao Jia, Y. Z. (2023). *Programmable System Call Security with eBPF.* arXiv.
25. LOOKOUT. (n.d.). *what is mobile threat?* Retrieved from Lookout: https://www.lookout.com/glossary/what-is-a-mobile-threat
26. Marco Abbadini, M. R. (2023). *NatiSand: Native Code Sandboxing for JavaScript Runtimes.* China: ACM Digital Library.
27. Marco Zuppelli, L. C. (2021). *Detecting Covert Channels Through Code Augumentation.* CEUR-WS.
28. Mirko Sailio, O.-M. L. (2020). Cyber Threat Actors for the Factory of the Future. *MDPI*, 25.
29. Omar M. Ahmed, A. B. (2017-08-30). *Android Security: A Review.* Iraq.
30. OWASP. (n.d.). *Mobile Top 10 2024: Final Release Updates*. Retrieved from OWASP: https://owasp.org/www-project-mobile-top-10/
31. Özeren, S. (2024, FEB 29). *February 2024: Latest Malware, Vulnerabilities and Exploits*. Retrieved from PICUS: https://www.picussecurity.com/resource/blog/february-2024-latest-malware-vulnerabilities-and-exploits
32. Podobnik, T. (2023, NOV 13). *Security Evaluation: eBPF vs. WebAssembly*. Retrieved from Medium : https://cloudchirp.medium.com/security-evaluation-ebpf-vs-webassembly-738df2566b18#:~:text=eBPF%20programs%20operate%20within%20a,bounds%20or%20reading%20uninitialized%20memory.
33. PROMON. (n.d.). *What is the OWASP MASVS?* Retrieved from PROMON: https://promon.co/owasp-masvs-resilience#:~:text=The%20OWASP%20MASVS%20(Mobile%20Application,and%20consistency%20of%20test%20results
34. Rajan Thangaveloo, W. W. (2020). *DATDroid: Dynamic Analysis Technique in Android Malware Detection.* International Journal in Advanced Science Engineering Information Technology.

35. RED HAT . (n.d.). *SELinux User's and Administrator's Guide*. Retrieved from Red Hat Documentation: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/chap-security-enhanced_linux-introduction#chap-Security-Enhanced_Linux-Introduction

36. Saba Arshad, M. A. (2016). *Android Malware Detection & Protection: A Survey.* Pakistan .

37. (2010). *Securing Android-Powered Mobile Devices Using SELinux.* IEEE.

38. Shubair Abdulla, A. A. (2015). *Intelligent Approach for Android Malware Detection.*

39. Source, A. (2024, April 29). *Kernel Overview*. Retrieved from Android Source: https://source.android.com/docs/core/architecture/kernel

40. Statcounter. (2024). Retrieved from Statecounter: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

41. The kernel development community. (n.d.). *LSM BPF Programs*. Retrieved from The Linux Kernel: https://docs.kernel.org/bpf/prog_lsm.html

42. *Understanding how Polymorphic and Metamorphic malware evades detection to infect systems*. (2023, May 24). Retrieved from FORTA: https://www.tripwire.com/state-of-security/understanding-how-polymorphic-and-metamorphic-malware-evades-detection-infect

43. *What Are The Differences Between BPF and eBPF: An Overview*. (n.d.). Retrieved from NETDATA: https://www.netdata.cloud/academy/what-are-the-differences-between-bpf-and-ebpf-an-overview/#:~:text=BPF%3A%20Primarily%20focused%20on%20network,other%20system%2Dlevel%20monitoring%20tasks.

44. Wiper Soft. (n.d.). *Mobile Malware Evolution 2019 [Part 1]*. Retrieved from Wiper Soft: https://www.wipersoft.com/mobile-malware-evolution-2019-part-1/

45. Yaniv Agman, D. H. (MAy 2021). *BPFroid: Robust Real Time Android Malware Detection Framework.* ResearchGAte.

46. Yi He and Zhenhua Zou, T. U., Kun Sun, G. M., Zhuotao Liu and Ke Xu, T. U., Qian Wang, W. U., Chao Shen, X. J., Zhi Wang, F. S., & Qi Li, T. (2022). *RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices.* USENIX.

47. Yusheng Zheng, T. Y. (2023). *bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions.* arXiv.

List of Tables and Figures