

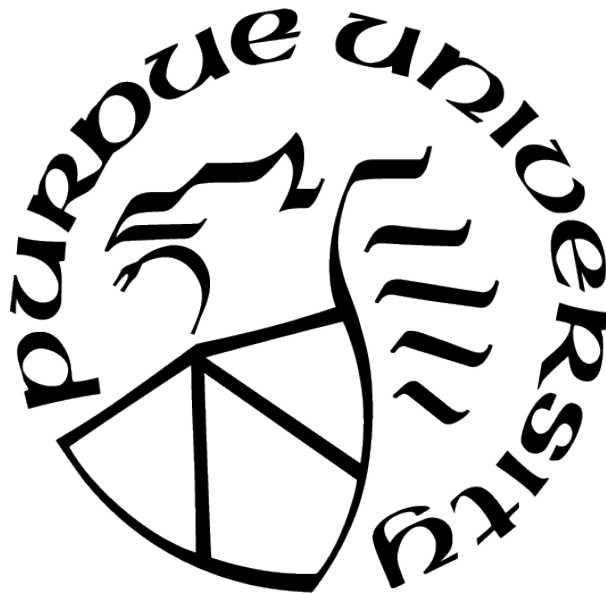
A FRAMEWORK FOR THE SOFTWARE SECURITY ANALYSIS OF MOBILEPOWER SYSTEMS

by
Yung Han Yoon

A Thesis

*Submitted to the Faculty of Purdue University
In Partial Fulfillment of the Requirements for the degree of*

Master of Science



Department of Computer and Information Technology

West Lafayette, Indiana

May 2021

**THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL**

Dr. Umit Karabiyik, Chair

Purdue Polytechnic, Computer and Information Technology

Dr. Marcus Rogers

Purdue Polytechnic, Computer and Information Technology

Dr. Tahir Khan

Purdue Polytechnic, Computer and Information Technology

Approved by:

Dr. John Springer

TABLE OF CONTENTS

LIST OF TABLES	6
LIST OF FIGURES	7
ABBREVIATIONS	8
ABSTRACT	9
1 INTRODUCTION	10
1.1 Significance and Motivation	10
1.2 Research Question	12
1.3 Assumptions	12
1.4 Delimitations	13
1.5 Limitations	14
1.6 Hypothesis	14
1.7 Contributions	14
2 LITERATURE REVIEW	16
2.1 Security Auditing	16
2.1.1 Information Systems Auditing	17
2.1.2 Software Security Auditing	19
Static Code Auditing	21
Dynamic Code Auditing	22
Reverse Engineering	22
2.2 Power Management Systems and Battery Management Systems	23
2.2.1 Security Research	25
2.3 Android Architecture and Security	25
2.3.1 Applications and High Level Layers	26
Android Power Saving Mode	28
Doze	28

	Application Standby	28
	Wake Lock	28
	Adaptive Battery and Brightness	28
	Security Research	29
2.3.2	Java Framework and API's	30
2.3.3	Native Software Libraries and Android Run-time	30
2.3.4	Hardware Abstraction Layer	30
2.3.5	Android Kernel	30
2.4	Previous Work and Related Frameworks	31
2.5	Summary	33
3	FRAMEWORK AND METHODOLOGY	34
3.1	Framework	34
3.2	Methodology	35
3.2.1	Data Acquisition	35
3.2.2	Analysis Procedure	37
4	FRAMEWORK EVALUATION AND EXPERIMENTAL RESULTS	38
4.1	Framework Evaluation and Analysis	38
4.1.1	High Level Investigation	38
4.1.2	Low level Investigation	40
	Partition Discovery	40
	File Discovery	40
	Dependency Mapping	47
	File Analysis	51
	Drivers and Kernel Analysis	53
	Scanning and Auditing	53
4.2	High Level Vulnerability Results	55
4.3	Low Level Vulnerability Results	57
5	DISCUSSION	58

5.1	Research Question	58
5.2	Validity	58
5.3	Viability and Usefulness	59
5.4	Challenges	59
6	CONCLUSION	61
	REFERENCES	63
A	APPENDIX	69
A.1	Device Info	69
A.2	Tools used	69

LIST OF TABLES

2.1	MBS relevant driver directories in Android kernel	31
2.2	Exploits in Android by attack vector as a percentage of 63 total samples. Data sourced from [47].	32
4.1	Extensions of files found within the partitions	42
4.2	List of keywords used in first round of file discovery.	42
4.3	Example of similarly named dependency with different contents.	47
4.4	Security scan results from Oversecured for <i>SamsungDeviceHealthService.apk</i> file	55
4.5	Security scan results from Oversecured for malicious file	55
4.6	Security scan results from Oversecured for Signal	56
6.1	Contributions of this work and the discoveries that have been made toward each	61
A.1	Device information for Samsung A50 used in case study	69

LIST OF FIGURES

2.1	Abstracted BMS architecture.	23
2.2	General architecture of Android.	26
2.3	Proposed architecture of MBS system for Android representing all possible attack vectors, relevant components of data at each level, and analysis methods.	27
3.1	Proposed general framework to conduct power system audit	34
3.2	Detailed framework to conduct audit developed from General framework.	35
3.3	Further refined set of procedures to follow taking the acquired data into account	37
4.1	Curated excerpt of <i>tree</i> tool output showing some of the files discovered in the exported <i>system</i> partition. Many more files with varying filenames and extensions were found but are not shown in the image.	41
4.2	The full file discovery process used to find files which need to be audited using automated Android security scanners	43
4.3	Curated examples of obvious power system related files found in the tree output of the <i>system</i> partition	44
4.4	Example of discovered power system related critical files	44
4.5	Within each shareable object library file, dependencies are recorded as a string	45
4.6	Dependencies of standard library files are largely reliant on <i>.cpp</i> files	46
4.7	Excel worksheet to process dependency data.	47
4.8	Script used to pull dependencies from all critical files and sub-dependencies	48
4.9	Boxed in red, two examples of erroneous dependencies manually removed from final diagram. Erroneous entries were introduced by the bash script's extraction of dependency information from <i>.so</i> files	49
4.10	Final diagram of interdependencies used to understand the BMS system being audited	50
4.11	Some of the HIDL references present in the file <i>6023-android.hardware.power@1.0.so</i>	52
4.12	Method names from source code match strings discovered in the <i>.so</i> file	52
4.13	Carved data from <i>boot</i> partition contains a file with the ELF binary header and references to a "kernel"	54
4.14	Detailed vulnerability report for a hard coded token vulnerability	56
4.15	Detailed vulnerability report for a dynamic broadcast receiver registration	57

ABBREVIATIONS

BMS	Battery Management System
MBS	Mobile Battery System, which includes the BMS for the mobile device
WBMS	Wireless Battery Management System
OS	Operating System
EV	Electric Vehicles
HAL	Hardware Abstraction Layer
ART	Android Run-Time
TTP	Tools, Tactics, and Procedures
AOSP	Android Open Source Project
SoC	System-on-a-chip
OEM	Original Equipment Manufacturer
ODM	Original Design Manufacturer
OTA	Over-the-Air
BSP	Board Support Package
SKU	Stock Keeping Unit
GKI	Generic Kernel Image
LKM	Loadable Kernel Module
ABI	Application Binary Interface
POC	Proof of Concept

ABSTRACT

Mobile devices have become increasingly ubiquitous as they serve many important functions in our daily lives. However, there is not much research on remote threats to the battery and power systems of these mobile devices. The consequences of a successful attack on the power system of a mobile device can range from being a general nuisance, financial harm, to loss of life if emergency communications were interrupted. Despite the relative abundance of work on implementing chemical and physical safety systems for battery cells and power systems, remote cyber threats against a mobile battery system have not been as well studied. This work created a framework aimed at auditing the power systems of mobile devices and validated the framework by implementing it in a case study on an Android device. The framework applied software auditing techniques to both the power system and operating system of a mobile device in a case study to discover possible vulnerabilities which could be used to exploit the power system. Lessons learned from the case study are then used to improve, revise, and discuss the limitations of the framework when put in practice. The effectiveness of the proposed framework was discovered to be limited by the availability of appropriate tools to conduct vulnerability assessments.

1. INTRODUCTION

As the world we live in becomes increasingly dependent on technology, so has our dependency on reliable sources of power to our devices. Of these technologies, smartphones and cell phones in general have become perhaps the most important of them all. Smartphones have become integral to modern day life as they enable people to communicate, socialize, access financial information, take pictures of their surroundings, do business, make emergency calls, and many other tasks that improve our quality of life, safety, and productivity. Security of operating systems, servers, applications, and websites have all improved over-time as companies harden their networks to prevent costly data breaches and loss of public trust. Given the benefits to productivity and quality of life that mobile devices represent, a successful attack that disrupts the accessibility and operation of a mobile device could have far-reaching consequences.

1.1 Significance and Motivation

There are numerous reasons why power systems are significant and deserving of further study. Past product defects which caused smartphone batteries to catch fire have posed serious health risks to the user [49]. These events were not caused by a remote threat, but it poses an important question in what damage may be caused by a remote attacker and how can they be prevented. Recent research has also proposed emergency response systems for automobile drivers who enter a medical emergency while driving [62]. This system relies on mobile devices as a critical component which could potentially save a life [62]. However, security vulnerabilities in power systems could also allow an attacker to cause a Denial-of-Service style attack, endangering human life. Normally, power system compromises tend to only result in the loss of availability of resources. However, the potential impact of a compromised power system goes beyond the realm of accessibility and could also impact confidentiality. Deliberate under-voltage of a CPU can be used to retrieve encryption keys [48], posing an additional threat to the confidentiality, and potentially even the integrity of data if keys related to RSA certificates are stolen.

There is comparatively little research into the BMS of mobile devices from a security perspective. However there has been research which looks at BMS systems in other domains, such as electric vehicles. These systems aim to protect a battery from physical or chemical danger like overheating, voltage, stress, and [39]. Past research also considered operational issues related to battery management systems that preserve battery life to ensure a battery has the longest life possible [37].

Other research into batteries look for better management and monitoring capabilities [18, 57]. Unfortunately, the safety of these systems is usually only considered after the product has been developed [23]. Of course, there is also constant ongoing research into the battery cells themselves to boost performance [20]. Research for BMS has also progressed into the development of systems that are reliant on wireless communications. These wireless BMS (WBMS) rely on wireless communication protocols such as Bluetooth for data communication and management operations [42, 66]. WBMS do not only exist in theory, researchers have implemented WBMS systems and found their performance to be comparable to wired BMS systems, while also improving safety [66]. However, the vulnerability of battery management systems and batteries themselves to deliberate attacks from a remote adversary is not as well researched. While the physical safety of battery cells and their management systems have been explored, there is not much literature on how modern smartphone operating systems (OS) use the stored power or how they are managed. There is also a noticeable lack of frameworks designed for auditing and assessing the security of this crucial cyber-physical system, or other related use cases. It is generally taken for granted that no harm could be done to a mobile battery system (MBS).

There is some research into the security aspects of BMS. BMS has been divided into abstract layers, and each layer was then examined for possible threats [45]. Of particular note is that claims have been made where BMS software can be replaced remotely via wireless communications technology using cross-layer attacks from higher layers [45]. Blockchain has also been a widely proposed solution to secure BMS systems for a variety of applications in IoT and EV [4, 14, 25, 28, 52]. A very recent article from 2020 explored the cybersecurity aspect of BMS and identified blockchain technology as a promising solution to defend against cyber-physical attacks [40]. The attack surfaces of BMS have been identified as “1) network

vulnerability; 2) software/firmware vulnerability; 3) data storage vulnerability; 4) on-board interface vulnerability; and 5) hardware component security vulnerability” [40]. Solutions that address these issues have also been identified, such as the removal of unnecessary interfaces during the latter development stages, use of secure compilers, and encryption of data-at-rest [40]. However, an in-depth evaluation of the current security of BMS, and MBS, in particular, from a cyber-physical and cybersecurity perspective, has not been conducted yet to the best of our knowledge.

To address this gap in research, I created an auditing framework which would improve MBS security by finding vulnerabilities.

1.2 Research Question

The research question I have is **“How effective of a framework can I currently create to discovery and identify real vulnerabilities in the power systems of mobile devices?”**

Effectiveness is operationally defined as “the number of true vulnerabilities that are found as a percentage of all known vulnerabilities”. The true goal of the framework is to discover new vulnerabilities that are currently unknown. However, this is a difficult measurement to use as it is impossible to know how many unknown vulnerabilities exist within a system. A weighting method of granting a numerical value to each unknown vulnerability discovered during our case study does not take into account whether they are truly exposed to remote actors, and thus represent real threats to system security, or whether they are “secured” thanks to other layers of defense. As such, I only consider known vulnerabilities as our way of measuring the success of this framework.

1.3 Assumptions

Some assumptions that were made for this research are:

- Different Android versions rely on the same set of core libraries

- Not using a factory reset device will pose no difference to the recovered firmware and library files.
- A bitstream forensic copy of the device is sufficient in collecting the relevant data
- Dependencies of libraries with similar names are used in the same way by the same other libraries, regardless of the architecture they were compiled for
- Dependencies of a library exist as headers in the binary file
- Users of this framework have a similar level of knowledge, skills, and abilities as the researcher; relatively inexperienced at software auditing but possesses general and basic IT knowledge including basic scripting, Excel proficiency, and digital forensics.

1.4 Delimitations

Given the in-depth and exploratory nature of this work, it may not be feasible in terms of time to conduct this research on multiple devices. As such, only one example device was examined. Only software libraries and firmware which are directly involved in the power system or management of the power system were included for vulnerability scanning. Apps with a power related function, which include libraries for user experience, internet connectivity, or functions unrelated to the power system, were only being examined to the extent of the power related functions present. Additional features and their related libraries used by the app were not be examined; however, they do present a feasible attack surface and possible entry points from a security perspective, which this work does not address due to limitations in scope and time.

- The lower levels of the Android power system are within scope.
- The higher application layers are within scope.
- The Automotive Power Management system is not within scope.
- Files and libraries which are not directly related to the power system are not in the scope for vulnerability assessment.

- Non-executable files are not within scope. For example, data made available by the sysfs file system, logs, and databases.
- Dynamic analysis methods will not be used due to limitations in time.

1.5 Limitations

The limitations of this work represent some of the issues regarding this framework when implemented in practice, as well as its limitations as a guide for its target audience. The limitations of this work are as follows:

- File discovery may not be comprehensive
- Effectiveness of the framework dependent on available tools
- False positives are likely to be generated by any procedure being followed, posing a cost to users of the framework
- False negatives are unavoidable
- Lack of domain knowledge in Android OS and software auditing from the researcher could lead to a less effective framework which misses important components

1.6 Hypothesis

The hypotheses for this research are:

H_0 : The proposed framework is no better than the random chance at identifying vulnerabilities at 50%

H_1 : The proposed framework is able to detect vulnerabilities at 50% or greater

1.7 Contributions

The contributions of this work are:

1. Proposal of a novel auditing framework designed to ensure the security of mobile battery systems.

2. Creation of a practical set of guidelines that can be followed by academic and independent security researchers interested in examining mobile subsystems for possible vulnerabilities
3. Identify components related to MBS for Android devices
4. A process to create diagrams of the intercomponent communication to understand BMS and highlight possible high-risk areas of vulnerability.

2. LITERATURE REVIEW

To develop this framework for assessing possible vulnerabilities in MBS, numerous components that are integral for investigation have been identified. As this research crosses a few different domains, namely, software security, power management systems and batteries, and Android operating system (OS) security, a relatively large corpus of background information is required to understand the different components that were involved in the auditing process. As such, relevant works in the following domains have been consulted:

- *Security auditing* - To understand current practices and limitations related to security auditing. Understand the software auditing techniques that were used.
- *Power and battery management systems (and security)* - Understand how BMS systems function in general.
- *Android operating system security* - The focus of this research lies in mobile devices. The selected test case is Android due to its open source nature providing research opportunities which can be feasibly conducted as compared to a proprietary closed source mobile platform like Apple.

2.1 Security Auditing

As the proposed framework is focused on assessing the security of a subsystem, the software used to run the said system is of paramount importance. As such, security auditing frameworks and research were reviewed for potential ideas and solutions.

Security auditing can be broken down into two categories. The first category is information systems auditing. Information systems(IS) auditing will be defined as primarily concerned with auditing general IT systems and infrastructure of a corporation. Common issues found using this type of audit are noncompliance with company security policies, default passwords being unchanged, incorrect firewall rules, and unnecessary ports and services being made available on servers. Information systems auditing is mainly concerned with the configuration of various IT infrastructure components to ensure that unauthorized access is not granted to a threat actor due to oversight or misconfiguration. The second category is

software security auditing which is concerned with vulnerabilities which exist in the code of certain applications and systems. Software audits are primarily concerned with company coding preferences and vulnerabilities which cause the program to behave unexpectedly.

2.1.1 Information Systems Auditing

There have been numerous works in IS auditing that created practical frameworks to guide people in conducting a full security audit. These frameworks aid system administrators and IT staff in conducting audits to ensure regulatory compliance, corporate policy compliance, as well as adequate security. The goals of information systems auditing include regulatory compliance, critical asset protection, security processes are in place and continuously improved, and controls issued by the company are regularly enforced[53, 55]. Information security auditing is challenging to conduct because of the nonexistent of a generally accepted methodology or approach to use when conducting information security audits[53].

High level conceptual frameworks for conducting information systems audits have been proposed with 5 core components which constitute the major objectives of any IS audit [53]. The 5 core components are the goals, that IS audits aim to satisfy. Together, they cover a companies[53]:

- regulatory compliance
- incident response plan
- email and hiring policy
- artifacts, software and devices which must be considered during an IS audit
- business and security processes
- policies and protocols used to administer technical standards

Designing an audit procedure that addresses these core components will benefit the company by ensuring protection of a business's IT infrastructure, continued business operations, and preparedness for a Computer Emergency Response Team(CERT). Other frameworks

proposed for IS audits focus on the need for auditors to identify assets, potential threats to those assets, and then evaluate risk and impact factors to develop appropriate policies [55].

Frameworks have also been proposed that specifically look at the networking infrastructure also have proposed frameworks [9]. Frameworks that target the networking infrastructure focus on identifying services and systems exposed to the internet, evaluating the risks posed by these exposed systems, testing them using a simulated attack, checking networking logs for evidence of a breach, and continuous research procedures for the organization to follow.

More detailed or practical auditing frameworks include a mix of both auditing networking infrastructure, hosts, and company policies and have been validated using real life case studies [44]. A framework by Lo and Marchand proposes using a top-down, external to internal approach [44]. First, remote threats are considered, and various tools such as IP port scans and vulnerability scanners are launched. Then the system and company policies are checked for compliance and reasonableness. Individual systems and services are then examined, and finally people are surveyed and interviewed to ensure proper training has been done and password security is maintained.

These forms of IS security auditing is conducted to improve the security of a system. One possible way to define “security” is the preservation of the three core principles of confidentiality, integrity, and availability. The CIA triad, as it is more commonly known by “is a widely used benchmark for evaluating information system security effectiveness” [26]. Each component of the triad refers to a distinct, but important aspect of what it means to keep a system secure. Confidentiality means that data should not be seen and interpreted by users who are not authorized. Integrity means that data cannot be tampered with without authorized users of a system being made aware. Availability means that the system being evaluated is operational when needed and can continue to serve the needs of its users. Usually, security threats will attack one of these three components of the CIA triad. When vulnerabilities or exploits are discovered, they usually compromise one or more of the triad. The vulnerabilities that this framework seeks to identify in MBS also fall into this classification. Therefore, this framework seeks to find vulnerabilities in MBS that could lead to the compromise of any of the principles of the CIA triad.

IS audits are generally concerned with meeting high level objectives that are set by the company conducting or preparing themselves for auditing. They focus on ensuring that configurations for the various components in the system are properly set, and policies are adhered to. General tools that check for vulnerabilities and other potential points of entry are run to give auditors an idea of what attack vectors remain open. The high level considerations and perspective on what makes an audit successful and the top-down approach are used in the development of my framework.

2.1.2 Software Security Auditing

Software security auditing is primarily focused on the identification of vulnerabilities. As this framework being developed is aimed at assessing the security of MBS, the tools and techniques used by software testers and in software audits are essential to understand. Software has a crucial role to play in information systems, not least MBS. Some software is critical and cannot be allowed to fail due to catastrophic repercussions, such as those found in automotive or aerospace applications [15]. For the purposes of MBS security, software vulnerabilities are the primary way that adversaries interact and cause harm to the assets that need to be secured and as such reliable methods of assessing software security are needed.

Software auditing can be difficult to do as there is a lack of requisite skills in quality assurance testing [58]. The focus of quality assurance testing on software functionality or features is also prioritized, while security is often not thought of as important or considered second to functionality [58]. There are resources published which aim to train people in performing software security testing to meet this potential gap.

Available software auditing books and other resources teach how to plan and formulate attacks against a variety of software by taking into consideration some of the most common attack patterns, such as identifying data inputs and analyzing them for input sanitization [6, 34]. These techniques include the use of debuggers to step through a software execution to understand how data is manipulated and passed between functions, understanding the

technical implementation of a variety of common exploits such as buffer overflows, input fuzzing, and exploiting known vulnerabilities in certain software libraries.

One specific book, “The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities”, examines three important aspects of software auditing: code comprehension, candidate point, and design generalization [24]. Code comprehension is when an auditor simply looks at the source code and tries to discover vulnerabilities by [24]:

- tracing malicious input to see if it has unintended effects
- analyzing specific modules and classes
- analyzing algorithms used by the program that could be exploited

However, this approach can be long and tedious. Aside from the fact that it requires a person to do the analysis, it is also inefficient as there are no filters in place to limit the code that needs to be audited. Candidate points serve as a potential remedy to this. A list of potential issues or dangerous snippets the code is first created [24]. Then the auditor looks through the code for the target application in an attempt to find instances of these potentially problematic code snippets [24]. The final strategy is to work from a top-down approach, similar to some frameworks proposed in IS auditing. Design generalization has the auditor looking at the high level code to figure out how the software operates, then works from the high level functionality of each module, class, main function, to the individual function calls to determine how they need to be constrained(or sanitized) to enforce only expected behaviors [24]. I used a variety of tools which employ these techniques to scan for vulnerabilities within the MBS.

Software code auditing can be further subdivided into two general categories, static and dynamic [31]. Static analysis techniques center around analyzing the source code of the application itself and can be simple searches for certain function names to more involved processes which extract specific code segments which could prove dangerous [31]. It is worth mentioning that static code auditing can be done on both source code and with only compiled machine code. Static analysis can be run to look for function calls in high level programming languages, or a series of assembly instructions that are found in commonly exploited libraries.

Dynamic audits are used to discover possible vulnerabilities while the application is running during run-time [31]. Dynamic analysis, as it involves running the software, can potentially be said to be “easier” for an auditor who is not familiar with code auditing. This is because static analysis tools may be accurate, but there are still false positives. New auditors may not be familiar with the source code, or even machine code, and as such may find it difficult to determine if a flagged code snippet is actually a cause for concern. Dynamic analysis has the potential to mitigate those concerns. Tools and techniques such as input fuzzers and vulnerability scanners, which can be run against a running process mean that even relatively inexperienced IT staff can understand what is occurring, and how it is relevant to the security of the software being audited.

Static Code Auditing

For software dependency and availability, static analysis frameworks try to abstractify the source code they have been fed. Static analysis tools can operate using 3 stages: preprocessing, parsing, and analysis [15]. Numerous low level techniques are used by programmers and security researchers to check for vulnerabilities in software. Typical problems that occur at the software level of security audits are vulnerabilities such as buffer overflows, known vulnerabilities which exist in common software libraries, string formatting, code injection, symbolic link abuse, and race conditions [31]. A great example of how most static code audits work is with works which used static code analysis to aid in intrusion detection systems. The system calls of a program are parsed from the source code or machine code and then the relationships between each are assigned based on a number of factors such as the system calls potential for harm and the order in which system calls appear in, as well as which functions call other functions themselves [73]. By creating this model of how a software operates from only its source, common attack life cycles which involve the retrieval of additional exploit code from a remote source. By creating a behavioral model of an application from its source code, then comparing it to a running application whose system calls are being monitored, an effective intrusion detection system is created. This particular work also happens to show

how dynamic code auditing, the monitoring of system calls made by running applications, can work in conjunction with static code analysis to bring better results.

Dynamic Code Auditing

Dynamic code auditing can be less technically challenging. One application of dynamic code auditing exists in the realm of voting machine software audits, where a different approach has been proposed. The proposed research dynamic auditing approach involves running an audit software which screenshots and parses out the UI after ballots are cast and maintains its own log of votes [30]. Discrepancies between the audit software and the target software’s vote tallies are used as an indication that the voting software is not behaving within expectations. Other dynamic analysis methods run a client-server scheme, applications are installed and ran on a mobile device, with system calls being tracked to analyze potentially malicious code in dynamically linked libraries [76]. These dynamic methods of software auditing still seem to rely at least initially on some static analysis or preprogrammed baseline so that the abnormal activity recorded can be flagged as suspicious and potentially malicious.

Reverse Engineering

Reverse engineering is a notable form of software security auditing, however it does not neatly fit into either static or dynamic analysis as it draws on techniques belonging to both categories. The aim of reverse engineering was to understand a software or a set of code. Originally, this was done for software development and maintenance [17]. Poorly documented software may need to have their capabilities extended, modified to function on a new platform, or just re-documented properly [17]. Reverse engineering techniques were used by developers to learn how poorly documented software operates so that the necessary changes and updates could be made.

From a security standpoint, reverse engineering is also a useful tool as it can be used to identify and patch software vulnerabilities. For example, adversaries use reverse engineering techniques to learn how software operates to identify and exploit new vulnerabilities.

Companies may use reverse engineering to steal features from a competitor’s software to implement themselves.

For this research, reverse engineering techniques are not within scope. Reverse engineering a system for vulnerabilities could require a highly technical skill set as well as a large amount of time. As the goal of this framework is a set of procedures which can be used by an average IT staff member with limited time, resources, knowledge, skillset, and abilities, including reverse engineering as a step within the proposed framework would be counter productive.

2.2 Power Management Systems and Battery Management Systems

An abstract model of BMS can be made out of three layers, the physical layer, battery management system layer, and finally the application layer[45]. The abstracted architecture for BMS systems is shown in Figure 2.1.

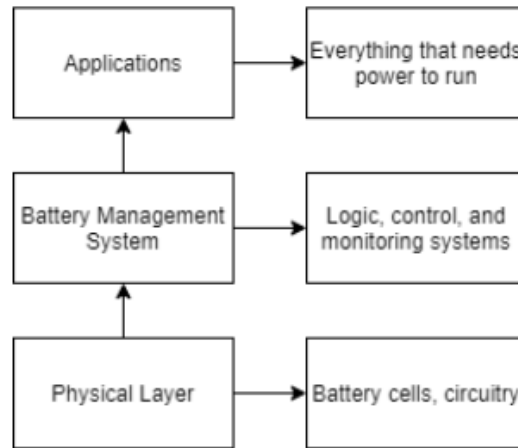


Figure 2.1. Abstracted BMS architecture.

This provides a general understanding of how to logically separate batteries and their management system from the rest of the device it serves. The battery itself is considered a part of the physical layer. Any circuitry, software, or other components related to the monitor and control of the battery within the mobile device is considered the battery management system layer. Finally, the remaining components of the mobile device, such as the CPU,

screen, camera, operating system, drivers, software libraries, framework, applications, etc. are part of the application layer. One limitation of this architecture is that many components are all grouped into the same layer which may make this layer overly broad to the point where it is not a useful abstraction of MBS.

BMS systems in general have several key functions they serve at a high level. These functions include: monitoring battery cells, protecting and ensuring battery safety, estimating the available battery charge, tracking the number of charge cycles to estimate the end-of-life of the cell, balancing multicell batteries during charging, managing the heat generated by the cells, and managing the charging rate to be fast yet safe [64]. At a lower technical level, numerous technologies and advances have been made to allow the power systems and OS to operate together seamlessly. The Advanced Power Management (APM) standard was introduced to allow the computer and BIOS system to include power management features [16]. This was then succeeded by the Advanced Configuration Power Interface (ACPI) standard, which provided a further layer of abstraction so that operating systems and power management software could be developed independently of each other, but still remain compatible and improve power management capabilities [3]. This also allowed manufacturers to expand power management options without having to worry about the higher level systems breaking.

In addition to these technical advances, a system called smart battery data (SMDData) was created that allows rechargeable batteries to communicate via bus to the device to provide more information allowing device users understand the charge state of the battery [16, 68] and serves as the first stepping point where possible malicious interference can occur.

From a security perspective, the concept of battery deprivation attacks has been around for some time. The earliest example found was an attack from 1999 termed “sleep deprivation torture”, where networking devices which are designed to sleep during a period of low usage are prevented from doing so, draining the battery and causing a denial-of-service [69]. More generally, the threats posed to the power system are adversaries who seek to subvert various power saving features that designers and manufacturers implement to extend battery duration [16].

At this level, there does not seem to be remote access capabilities which can be taken advantage of by adversaries. As such, any attacks against MBS would likely need to begin at higher layers.

2.2.1 Security Research

Current security focused research in BMS has seen proposals which take advantage of the reporting capability of modern battery systems to defeat malicious adversaries. In fact, the advances made with ACPI and expanded battery monitoring have created a variety of defense possibilities [16]. A BMS system was used as part of an intrusion detection system (IDS) [36]. The proposed system monitors the energy consumption, battery temperature, and other parameters to detect abnormally high levels of activity which could indicate malicious code being executed. Other research has discovered that attacks could not only be detected, but a signature of the attack can be created based on its pattern of battery consumption [16] to create an attack detection and classification system. E-Android is a tool which monitors the energy drain on Android mobile devices [29]. This tool was created as existing energy accounting systems in Android do not capture all energy expenditures. E-Android helps with the monitoring of energy consumption and better reporting, so users understand how the battery charge is being consumed. This tool extends the existing Android framework to collect more data that is used by the system to account for collateral energy consumption [29]. While this feature is useful for detecting possible malicious activities, it is not able to find vulnerabilities prior to their exploitation.

2.3 Android Architecture and Security

To comprehensively assess the MBS, all potential attack surfaces need to be identified so that they can be addressed within the proposed framework. First, an abstract architecture of Android was created to formulate a plan of attack on how the framework would approach the various components related to MBS in a mobile device. Android’s official documentation includes a useful abstract view of the entire Android operating environment and divides everything into six layers. These layers are shown in Figure 2.2 [56].

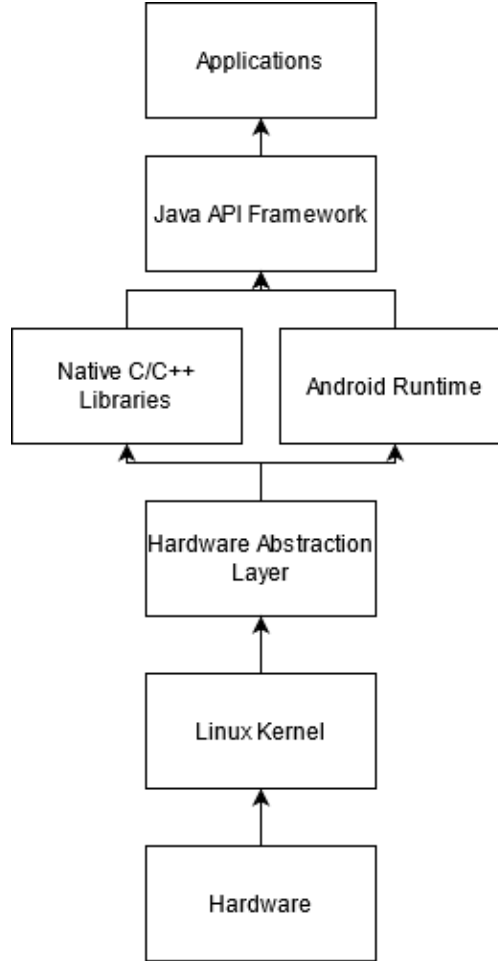


Figure 2.2. General architecture of Android.

Combining this model with the previous abstract model for BMS, I now have a full model representing a comprehensive attack surface for MBS. Each component in their respective layer is then matched with an appropriate analysis methodology which uses a code auditing technique. The proposed architecture is shown in Figure 2.3.

Now, I examine each layer to identify what they consist of, where possible vulnerabilities may exist, and how it may be possible to audit these components in practice.

2.3.1 Applications and High Level Layers

Android developers have access to a number of options made available by Android through Application Programming Interfaces (API) to allow them to manage the power their apps

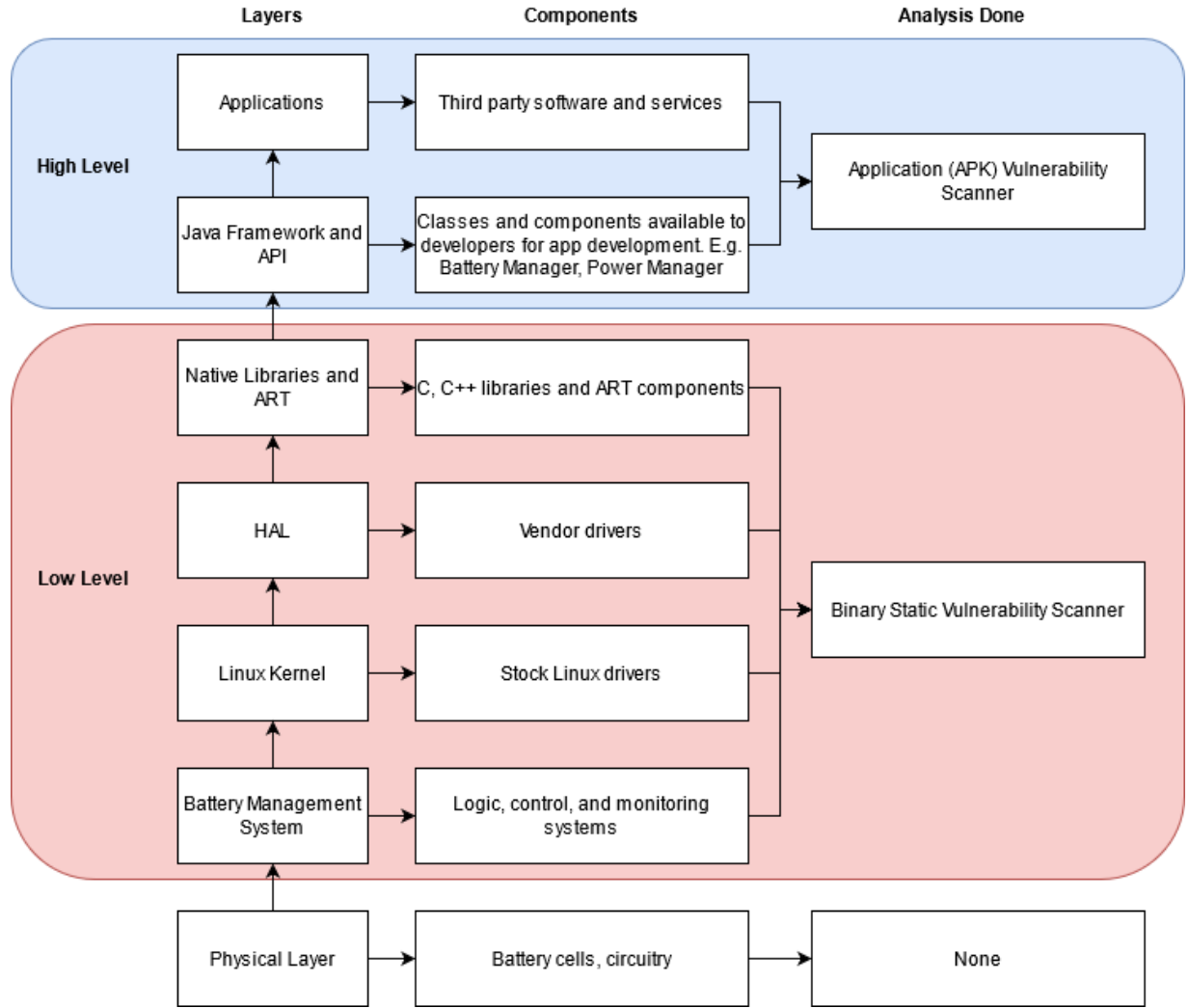


Figure 2.3. Proposed architecture of MBS system for Android representing all possible attack vectors, relevant components of data at each level, and analysis methods.

consume[54]. These features could potentially be taken advantage of by an attacker, either by creating an app which purposefully ignores the use of these features or by finding methods to exploit the apps to cause them to stop adhering to the power conservation settings they are coded for. The Android features related to power saving and management are as follows [54].

Android Power Saving Mode

Decreases performance of the phone by throttling power consumption through normal app functionality reduction. Attacks using this power saving mode may not be very effective, but misinformation is also a potential attack outcome and as such this mode needs to be investigated.

Doze

Doze mode is a feature of Android intended to help preserve a mobile devices battery. While a mobile device is not charging and the screen is off, the phone enters an idle state where applications are only periodically allowed to execute tasks [19].

Application Standby

Android will disable an application from much of its activity and only allow the app to run processes once a day or until opened manually by users. Only applications which have not been used by the user for a while will be subject to this form of throttling.

Wake Lock

The phone and applications will continue running even if the screen is off or blank.

Adaptive Battery and Brightness

Android uses a machine learning algorithm in Android Pie and later versions. This machine learning algorithm feature is called Adaptive Battery and Adaptive Brightness and is enabled by default on devices which are capable. This feature attempts to predict which apps are used most often by the user and throttle apps that the algorithm expects will not be opened. The code for this system is under the Device Health Services subsystem. This application is also responsible for battery depletion prediction and as a result has a lot of OS system calls implemented to enable it the functionality it needed.

All of these features can be potentially leveraged or circumvented to decrease battery life and increase battery charge consumed by an attacker.

Security Research

For Android applications, there are a number of proposed attacks that have already been researched. The Doze mode has been subverted by researchers to negate any potential charge saving benefits the feature provides. Researchers introduced malware would force the phone to change out of the idle state, thereby causing higher battery load even when the phone was set to an idle power state [19]. The experiments results showed the phone's battery under attack had lost 32% charge over 3 hours, as compared to 5% without malware interfering with doze mode.

In a different track, other potential attacks could involve misrepresenting the charge state to the user. The Toast and system alerts [51, 71] capability of Android has been subverted for malicious purposes. Android scan and pay is a feature that uses Toasts to display a QR code to conduct financial transactions between users [72]. However, fake QR codes of attacker's wallets can be injected instead of legitimate wallet addresses using Toasts, causing money to be sent to the wrong person [72]. These system notifications and Toasts may also be used to mislead users into believing falsehoods about their battery charge level. For example, Toasts could notify a user that their battery is low when it is not, forcing them to plug in an already charged phone. More seriously, a user could be misled into thinking their phone is fully charged, when in fact it is about to shutoff. This could lead to a more significant impact if the user was depending on the device for business or important communications. Even the minor inconvenience's cumulative effects in aggregate can have a substantial impact if it affects many people across a long period of time, and the loss in productivity should not be ignored.

Other high-level battery drain attacks have been researched, such as exploiting the insecure cellular communication protocol MMS [63] and playing media encoded in energy hungry formats over web pages [27]. Some of these attacks are capable of being launched without user intervention or action being required.

2.3.2 Java Framework and API's

This layer of API's is designed for developers to access underlying Android features and functionalities. These API's include various services that can be examined for possible vulnerabilities, including: Views, Resource Manager, Notification Manager, Activity Manager, Content Provider [56]. Of particular interest is one API called Battery Manager which is responsible for checking battery information and charging status [13]. The Power Manager API is also of special importance as it allows developers to control the power state of the mobile device [59]. As these are API's written in Java, software, auditing techniques would likely be effective at examining this layer and any components related to MBS.

2.3.3 Native Software Libraries and Android Run-time

This layer of the Android architecture consists of two distinct yet important parts. Native software libraries are written in C or C++ [56]. These libraries are used to run the core components of the Android OS and are of crucial importance when considering the security of BMS. The Android run-time itself hosts applications in their own virtual machines as independent processes [56].

2.3.4 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is an API layer that allows software and hardware to communicate [47]. This layer mainly contains code that was created by the hardware vendors themselves [47]. The components at this layer that were examined are comprised mainly of software and interfaces.

2.3.5 Android Kernel

The Android kernel represents a possible attack surface. The kernel and drivers need to be examined to determine if they contain possible software vulnerabilities. Consulting the Google source code for the Android kernel, driver directories related to power can be found. These relevant directories and their functions are shown in Table 2.1 [7]:

Table 2.1. MBS relevant driver directories in Android kernel

Driver Directory	Relevance
<i>/acpi</i>	Add's ACPI support to the Linux kernel.
<i>/power</i>	Three subdirectories for: advanced voltage scaling, <i>/reset</i> for device restart and shutdown, and <i>/supply</i> .
<i>/supply</i>	Allows power supply (includes batteries, AC, USB) monitoring by userspace applications via the <i>sysfs</i> and <i>uevent</i> .
<i>/powercap</i>	Interface to allow kernel subsystems to expose power capping settings to the user space with consistency.
<i>/regulator</i>	Generic (within the branch and manufacturer specific) interface to voltage and current regulators within the Linux kernel. Provides voltage and current control to client or consumer drivers and provide status information to user space applications through a <i>sysfs</i> interface

In addition, there is the *sysfs* pseudo file system provided by the Linux kernel that exposes information from lower level firmware, subkernel modules, hardware devices and their device drivers from the Linux kernel space to the user space using a traditional seeming file system [65, 70]. While this method of accessing data is available to applications, it is not recommended [65]. This system should be taken into consideration when assessing possible attack surfaces but is out of scope for this research as it is not software, vulnerability based but related to exposure of possibly sensitive information. However, it should be said that *sysfs* is a relatively secure system as it is explicitly designed to be difficult for users and attackers, to abuse by emphasizing the use of established and secure API's [65] so as an attack surface the *sysfs* file system may not be promising.

Of particular interest in this layer are the Android specific drivers that, among other things, support the function of power management [47]. In the past, drivers which provide functionality specific to Android have demonstrated vulnerabilities caused by issues related to the driver which enables Android Shared Memory (ashmem) [21, 47].

2.4 Previous Work and Related Frameworks

In terms of auditing frameworks designed for Android and mobile devices, there are previous works that can be relied upon for guidance. One work looked at 63 publicly released Android malware samples and analyzed them to determine their attack vectors and delivery

methods [47]. This relatively small sample is far from indicative of the true distribution of exploit attack vectors, however it helps to establish which components within Android are most likely to be targeted or vulnerable. Of the 63 samples analyzed, 30.6% of exploits used applications as a delivery method, 59.7% ran through shells, and 9.7% of exploits were capable of being executed remotely [47]. The surveyed exploits can be categorized based on the attack vector used and are shown in Table 2.2.

Table 2.2. Exploits in Android by attack vector as a percentage of 63 total samples. Data sourced from [47].

Exploit attack vector	Percentage
File system	9.5
System component	25.3
Linux kernel	17.4
Vendor driver	39.6
Trusted execution environment	7.9

A large proportion of exploits surveyed used vendor drivers, system components, and the Linux kernel. Specifically, vendor driver exploits took advantage of memory corruption attacks, Linux kernel attacks used kernel libraries and driver interfaces to cause memory corruption and start services [47]. System component exploits has a slightly more nuanced history and future. Abuse of daemons for exploitation generally stopped by 2014, while all other system component exploits took advantage of native libraries from 2015 onwards [47]. The Linux kernel and vendor drivers, as well as native libraries, are important elements that have been taken into account in the proposed framework.

Other work in Android security auditing developed a framework to analyze Android applications for application vulnerabilities using reverse engineering and gives insight on what tools could be used for this process. VAnDroid is proposed as an effective tool which is capable of detecting sensitive information leaks, intent spoofing, and unauthorized intent receipt [50]. In addition to the proposed VAnDroid framework, a variety of other related works and tools capable of detecting similar vulnerabilities were also explored that can be considered for inclusion in the new framework [50].

Other proposed security frameworks operate by using data from the National Vulnerability Database (NVD) to evaluate projects on the Maven repository to detect vulnerabilities [5]. Using the framework, vulnerabilities are detected by extracting features from projects and vulnerabilities and inferences are drawn between them using a semantic web [5].

2.5 Summary

To recap, in this chapter a comprehensive overview of various system auditing techniques and the two systems which were audited was made. The various frameworks from literature aimed at discovering software vulnerabilities were used to develop the proposed framework.

3. FRAMEWORK AND METHODOLOGY

Based off the information gained from literature and previous work, a general framework is created to audit MBS. The framework is then refined to the point where it can be used in practical applications.

3.1 Framework

The framework for analyzing MBS was constructed based on current knowledge of BMS and MBS. Each component of MBS was matched with an appropriate auditing techniques to find vulnerabilities. From the information gathered, the basic procedure for applying the framework is laid out in Figure 3.1.

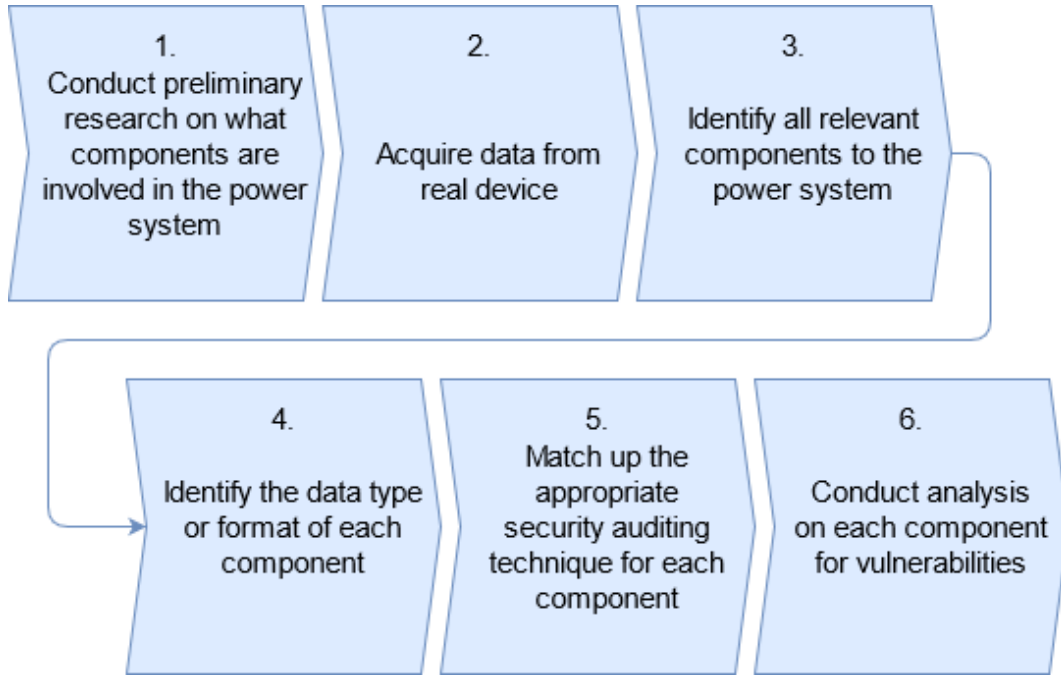


Figure 3.1. Proposed general framework to conduct power system audit

This generalized procedure has a high level of abstraction which makes it limited in usefulness for practical application. The proposed general framework is thus further refined and expanded on to include the tasks that were be performed in order to find vulnerabilities in the power system of a mobile device. This detailed framework which is much more practically useful is shown in Figure 3.2.

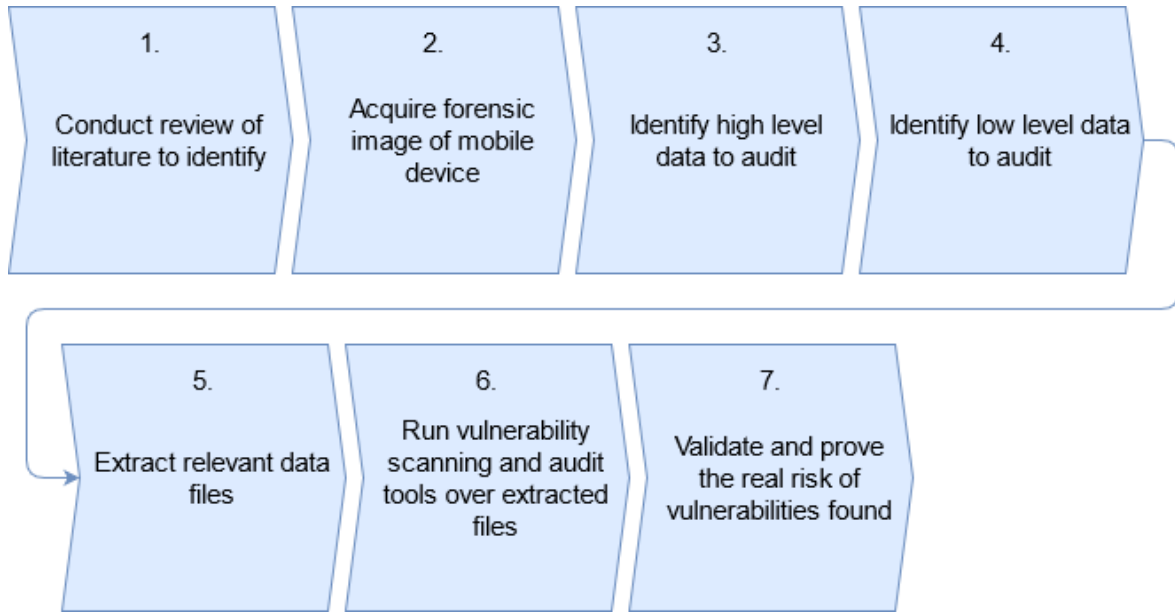


Figure 3.2. Detailed framework to conduct audit developed from General framework.

3.2 Methodology

The following details the steps taken to conduct the case study where the framework is applied.

3.2.1 Data Acquisition

To obtain the firmware, libraries, and drivers needed to conduct the security audit, I acquired a physical image of a Samsung Galaxy A50. This forensic copy of the mobile device should contain all the relevant files to be audited. UFED Cellebrite 4PC was used to acquire an image of a Samsung A50 mobile device. As the firmware of the device is the focus of this research, there is no need to factory reset the device to clear user data. Instead, the device was powered on to ensure it was operational, then an image was taken to acquire the firmware files needed for investigation. The procedure followed is listed below:

1. Obtained an already rooted Samsung A50. The Samsung device was rooted previously using Magisk.
2. Launched UFED Cellebrite 4PC

3. Recorded build number and other device details
4. Imaged device using UFED Cellebrite 4PC with following procedure:
 - (a) Started UFED Cellebrite 4PC application on workstation
 - (b) Selected acquire a mobile image
 - (c) Attached mobile device to workstation via USB cable
 - (d) Selected the A50 device that was detected by UFED Cellebrite 4PC
 - (e) Selected the full image option
 - (f) Followed all on screen prompts given by UFED Cellebrite 4PC to image the device

Information related to the acquired device is listed in Appendix [A.1](#).

Two factors are important to discuss regarding data acquisition. The first is the effect of rooting the device. Rooting the device grants tools and the user access to parts of the device which require special privileges. This could be a valuable and important step to take, as certain files are usually only available or readable when using special privileges. These files are usually important to the operation of the mobile device and as such may be stored in protected areas. As such, rooting the device so that the acquisition tool may copy all files in these protected locations is a requirement. This work does use a rooted device. The second factor to consider is the acquisition type used being logical or physical. Generally speaking, the files being studied come preinstalled on the user's device, or were manually installed by the users themselves. With this in mind, a logical acquisition should be sufficient in gathering all the required data as a logical image is able to capture all non-deleted files on the file system. However, as the appropriate tools were available, a physical image was taken of the device anyway. The supposedly benign software being auditing most likely does not rely on these deleted files to operate, thus recovered deleted files are likely not necessary.

3.2.2 Analysis Procedure

These are the steps taken during the case study to perform the vulnerability assessment. Any code or data processing step has been made available at the Github repository: [Repository Link](#).

With the actual device data in hand, it was valuable to revisit the detailed framework to further refine it based on the raw data available. The refined procedure for conducting the audit after the data was acquired and previewed is shown in Figure 3.3

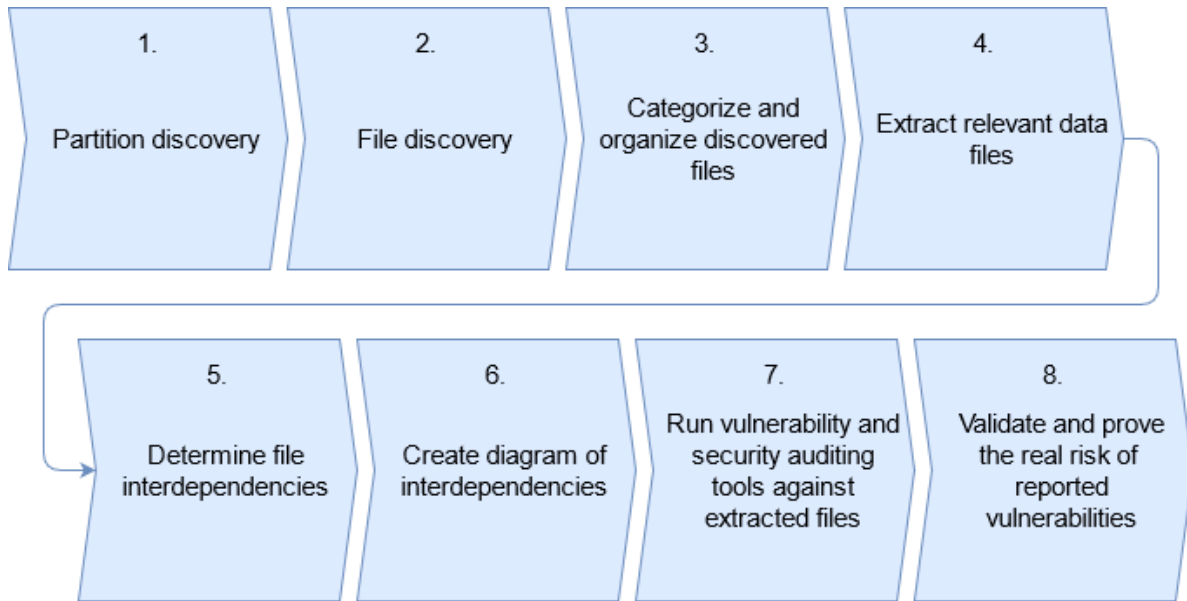


Figure 3.3. Further refined set of procedures to follow taking the acquired data into account

As this work seeks to answer the research question as well as provide a framework to be used as a roadmap for future research in this domain, the detailed process of how the investigation is conducted will be reported in Section 4.

4. FRAMEWORK EVALUATION AND EXPERIMENTAL RESULTS

This section will go over the results of applying the framework. I first explain the investigative process that application of the framework resulted in. This process consists of a series of steps and tasks which can be used by others. Followed by the results of the vulnerability scanning completed on both the high and low level components.

4.1 Framework Evaluation and Analysis

When the procedure in Figure 3.3 was followed, there were two distinct workflows for both the high level and low level components that were used in order to obtain the components necessary for analysis. Due to the different types of files in either level, separate vulnerability analysis tools had to be used as well.

4.1.1 High Level Investigation

There are numerous applications available on the Google Play store which purports to increase battery life [2, 10, 11, 12, 38]. The large variety of battery and power system related apps made it difficult to choose one over another when deciding on an application to audit. As such I looked for and found a preinstalled application which contained functionality related to the power system.

The device image was searched for all files with the *.apk* extension. A short list of candidate applications was created based on the names of the packages, then further online research was conducted to find an application with power system relevance. The *SamsungDeviceHealthManagerService.apk* package was a preinstalled, power system relevant application and was used for the high level vulnerability assessment. The package was found at the path */vol_vol28/system/priv-app/SamsungDeviceHealthManagerService/SamsungDeviceHealthManagerService.apk*. This package seems to be an application installed by Samsung which monitors and manage various aspects of the mobile device's health, the most relevant aspect for this audit being battery usage [43, 75].

I first used *qark*, the Quick Android Review Kit, a tool “designed to look for several security related Android application vulnerabilities...in source code or packaged APKs” [61]. The tool claims to look for 17 types of security vulnerabilities, including but not limited to: improper certificates, data leaks from Activities, Intents whose data can be intercepted, and use weak encryption [61]. However, after installing the tool, technical issues prevented it from successfully decompiling and scanning the *.apk* file.

Oversecured was then used to scan the application for vulnerabilities. Oversecured is an online based mobile application security scanner [8]. After signing up, it allows users to submit up to 3 APK files for free vulnerability scanning. More critically, the Oversecured interface also shows the user code snippets from the disassembled APK file that it detected as vulnerability risks. This feature was important as it would allow the researcher to further validate each vulnerability by showing where the vulnerability exists in the code. The results of this tool’s vulnerability scan are explained in Section 4.

Steps were taken to attempt to validate the discovered vulnerabilities by testing the reliability of the tool itself. Two known malicious and safe APK files were taken from online sources and uploaded to Oversecured in order to attempt to establish a baseline for how many vulnerabilities should be expected. A copy of a known malicious apk file was retrieved from <https://github.com/ashishb/android-malware/blob/master/malbus/19162b063503105fdc1899f8f653b42d1ff4fcfcdf261f04467fad5f563c0270.apk>, a github repository which hosts samples of Android malware. A copy of an application known to be relatively safe, and security aware, Signal was then downloaded from <https://www.apkmirror.com/apk/signal-foundation/signal-private-messenger/signal-private-messenger-5-7-5-release/>.

To double check that both of these files can be relied upon as a malicious and non-malicious baseline *.apk* file, they were uploaded to VirusTotal, a popular website which scans uploaded files for malware [1]. The Signal application APK was cleared as containing no malware, while the malicious APK triggered a positive malware identification result from 32 out of 62 of VirusTotal’s security vendors scanners. The reports for the malicious and Signal APK scan results from VirusTotal can be seen at: <https://www.virustotal.com/gui/file/19162b063503105fdc1899f8f653b42d1ff4fcfcdf261f04467fad5f563c0270/detection> and <https://www.virustotal.com/gui/file/0b7399e22215959dfe2a02a46c7d11ceb26f2e281d01433744c498ca272c7b>

[ce/detection](#). With confirmation that the files are truly malicious and non-malicious, they were uploaded to Oversecured to scan for vulnerabilities. The vulnerability analysis reports for these baseline files are shown in Section 4.2. At this point, all investigative steps necessary to get from device image to vulnerability scanning for the high level component is completed.

4.1.2 Low level Investigation

The low level components include all the components in the red section of Figure 2.3. These were the steps taken to identify them within the forensic image and use them for vulnerability analysis and scanning.

Partition Discovery

The forensically acquired device image contained multiple partitions, a total of 34, any of which could contain relevant power system files that need to be examined.

It would be difficult and unfeasible to manually review and analyze each and every single partition to locate the files needed. As such, online research was done to identify the functions and differences between each partition within the device image. Information available on the Android Developers website details the purpose of each partition, which identified the *system* and *vendor* partitions as the most likely candidates for where the relevant power system components could be found [60].

File Discovery

Within the power system relevant partitions, there are many files which are not relevant to power systems. As such, further steps were taken to figure out which files within both partitions should be included for vulnerability auditing.

Android uses the the bootstrap process: Boot ROM, BootLoader, Kernel, Init Process, zygote, system server [46]. A more detailed breakdown of the boot process shows that the first service directly related to the power system is the kernel, followed by init processes, and

finally system services such as the *power manager*. From this bootsequence's use of *power manager*, I decided to look for instances of this in the device.

On top of tracing the boot sequence to find relevant services and libraries, I also manually reviewed the contents of the partitions to find relevant files for auditing purposes. The contents of the partitions were exported from the data source into a folder while retaining the directory structure of the original image. The Ubuntu utility tool *tree* was then used to print out the directory and file names. Because Autopsy is able to perform file carving, deleted files and slack space for files were also exported as a separate independent files. However, these files do not truly exist and are products of Autopsy's forensic analysis so they were ignored. A small excerpt of the results from the *tree* tool is shown in Figure 4.1.

```
1 vol28(system)/
2 | 102087-$Unalloc
3 |   | Unalloc_102087_377544704_5813616640
4 | 12360-ueventd.rc
5 | 12362-vendor
6 ...
7 | 517-storage
8 | 520-sys
9 | 523-system
10 |   | apex
11 ...
12 |   |   | lib64
13 |   |   |   | android.hardware.graphics.allocators@2.0.so
14 |   |   |   | android.hardware.graphics.allocators@2.0.so-slack
15 |   |   |   | android.hardware.graphics.allocators@3.0.so
16 |   |   |   | android.hardware.graphics.allocators@3.0.so-slack
17 |   |   |   | android.hardware.graphics.bufferqueue@1.0.so
18 |   |   |   | android.hardware.graphics.bufferqueue@1.0.so-slack
19 |   |   |   | android.hardware.graphics.bufferqueue@2.0.so
20 |   |   |   | android.hardware.graphics.bufferqueue@2.0.so-slack
21 ...
```

Figure 4.1. Curated excerpt of *tree* tool output showing some of the files discovered in the exported *system* partition. Many more files with varying filenames and extensions were found but are not shown in the image.

From this data file, I created a list of all file extensions which was used to determine which files would contain executable code which could contain vulnerabilities. The file extensions that were found are shown in Table 4.1.

Table 4.1. Extensions of files found within the partitions

.so	.spi	.qmg	.apk
.tlbin	-slack	.dbg	.pem
-service-exynos	-service	.hcf	.xml
.bin	-service-slack	.gz	.cil
.rc	-service-armnn		

Of these file extensions discovered, Shareable Object (.so) files [74] are likely to be the strongest candidate for auditing as they are shared library files that other programs reference and use to import functionality. Looking at the .so fileheader, the string “ELF” is visible which denotes a commonly used Linux binary executable format [33]. A process is needed to find the .so files related to the power system for further security analysis.

I created such a process to automate as much file discovery as possible. The full file discovery process that was used is illustrated in Figure 4.2. An explanation for each step is provided below.

There are many .so files found which perform various important functions in the phone, however some files have filenames which are obviously related to or used by the power system and so were included in the vulnerability assessment. Some examples of these power system relevant files are shown in Figure 4.3

There seemed to be a naming convention being used with several keywords denoting a power system relevant file. A list of all the power system keywords, such as “battery”, “charging” or “power” was compiled. This list was expanded upon as new file dependencies were discovered. The full list of keywords used for the first round of file discovery are shown in Table 4.2.

Table 4.2. List of keywords used in first round of file discovery.

android.hardware.power	battery	batterymanager
batteryservice	charge	energy
power	health	

After running the keyword search, non-useful files, such as graphics files like “battery_fail.png” were included in the results. Also included in the results were various useful

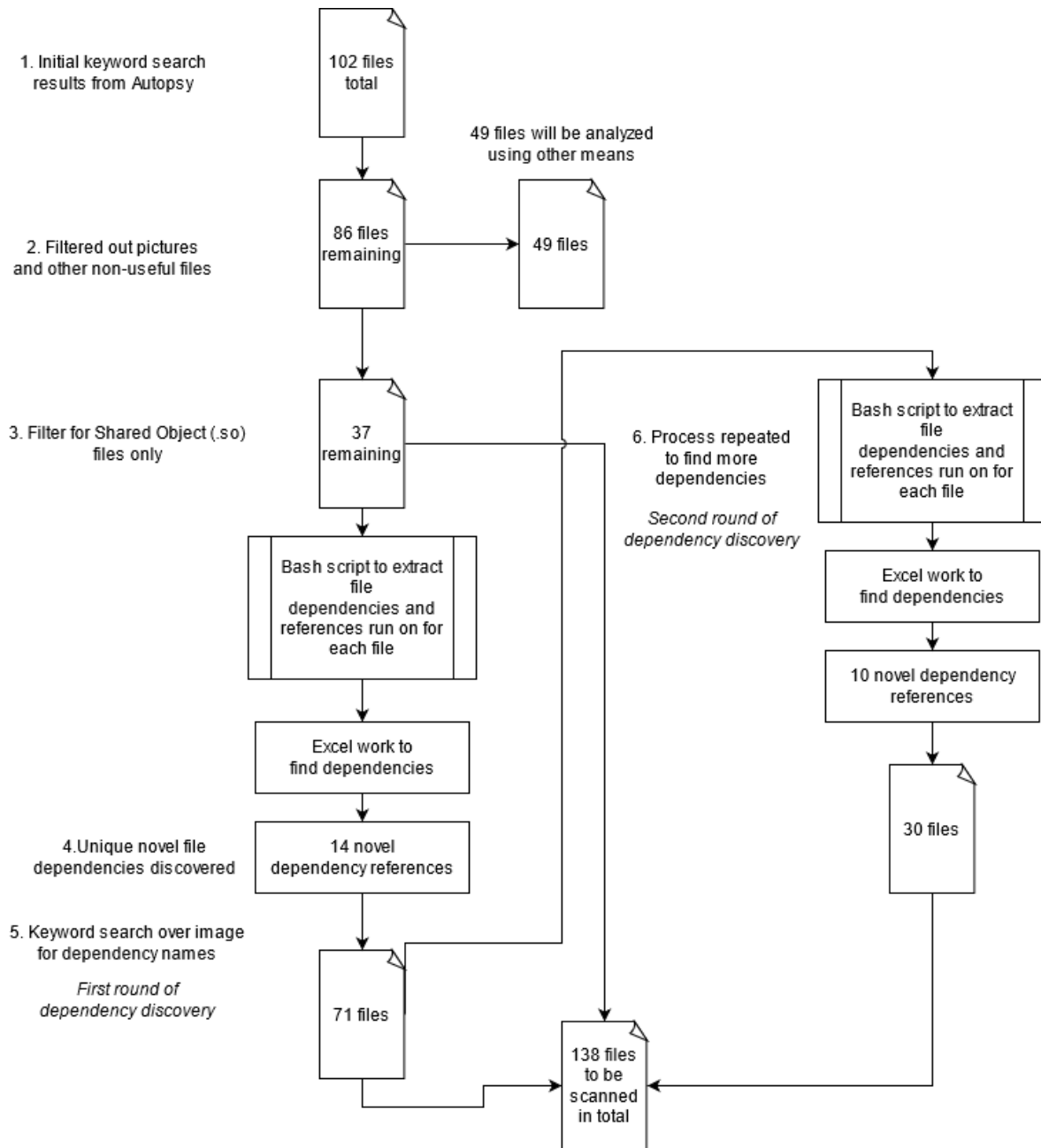


Figure 4.2. The full file discovery process used to find files which need to be audited using automated Android security scanners

file types such as databases, logs, executables, that could contain information related to the power systems operation. However, these files are out of scope for this research. After removing the irrelevant files, there were 37 files remaining that became the main focus of the auditing efforts and are referred to in this research as the **critical power files** or **critical**

```

1  ...
2  |   |— android.hardware.power@1.0.so
3  |   |— android.hardware.power@1.0.so-slack
4  |   |— android.hardware.power@1.1.so
5  |   |— android.hardware.power@1.1.so-slack
6  |   |— android.hardware.power@1.2.so
7  |   |— android.hardware.power@1.2.so-slack
8  ...
9  |   |— android.hardware.power.stats@1.0.so
10 |   |— android.hardware.power.stats@1.0.so-slack
11 ...

```

Figure 4.3. Curated examples of obvious power system related files found in the tree output of the *system* partition

files for investigation. Figure 4.4 depicts a subset of the critical files that were found using this method.




















Source File	△ S	C	Category	File Path	Size
 android.hardware.power.stats@1.0.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/android.hardware....	79412
 android.hardware.power@1.0.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/android.hardware....	65864
 android.hardware.power@1.1.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/android.hardware....	70608
 android.hardware.power@1.2.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/android.hardware....	70812
 android.hardware.power@1.3.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/android.hardware....	75432
 libpower.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/libpower.so	23784
 libpowermanager.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/libpowermanager.so	20888
 libSAG_VM_Energy_v217.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/libSAG_VM_Energy...	13008
 vendor.samsung.hardware.miscpower@2.0.so	!			/img_blk0_sda.bin/vol_vol28/system/lib/vendor.samsung.h...	61224
 android.hardware.power.stats@1.0.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/android.hardwar...	102056
 android.hardware.power@1.0.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/android.hardwar...	88376
 android.hardware.power@1.1.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/android.hardwar...	93488
 android.hardware.power@1.2.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/android.hardwar...	94064
 android.hardware.power@1.3.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/android.hardwar...	99016
 libpower.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/libpower.so	24168
 libpowermanager.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/libpowermanage...	21992
 libSAG_VM_Energy_v217.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/libSAG_VM_Ener...	41064
 vendor.samsung.hardware.miscpower@2.0.so	!			/img_blk0_sda.bin/vol_vol28/system/lib64/vendor.samsung...	79632
 battery_error spi	!	⚠		/img_blk0_sda.bin/vol_vol28/system/media/battery_error.spi	6648

Figure 4.4. Example of discovered power system related critical files

I looked at the critical files to discover what their dependencies were. The dependencies of each executable critical power file are included as strings close to the beginning of

each file. The Figure 4.5 illustrates the dependency being recorded as a string for the *android.hardware.power@1.0.so* file.

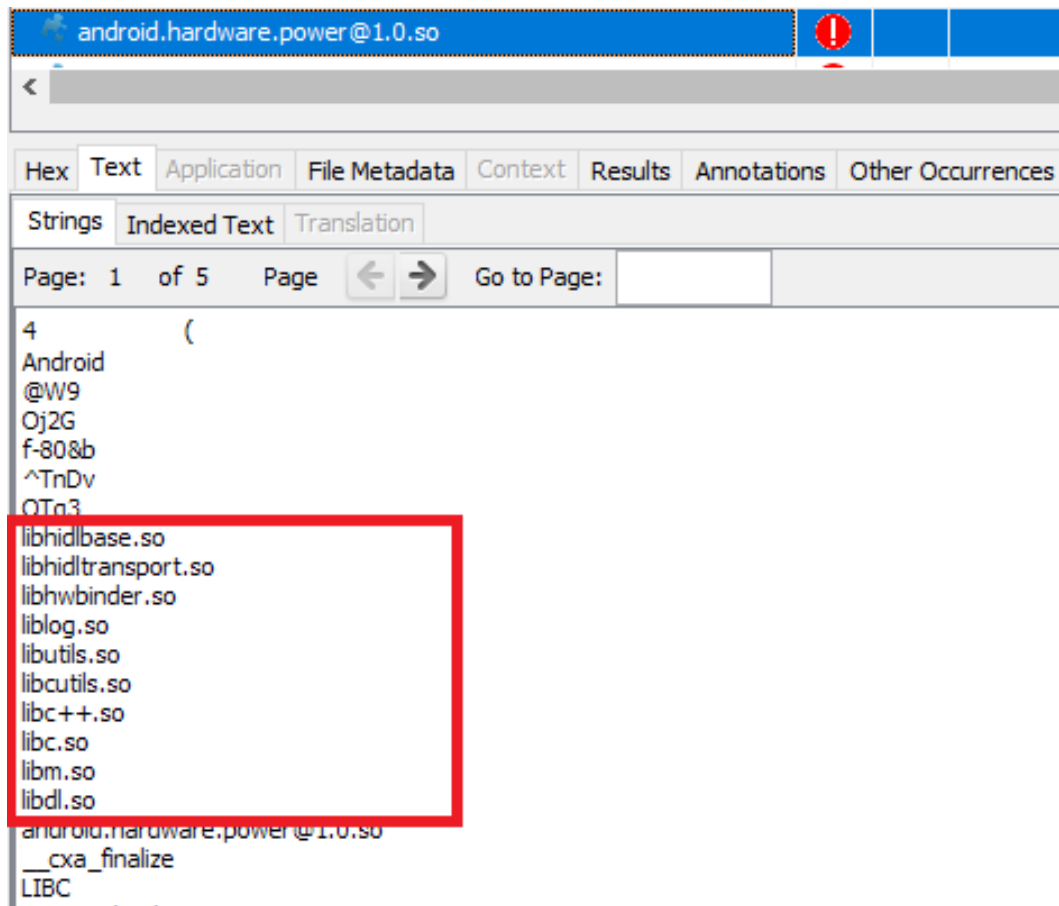


Figure 4.5. Within each shareable object library file, dependencies are recorded as a string

Further sub-dependencies of some files such as *libc.so* are more granular and consist of references to other compiled standard library files and C++ interface files designated with the *.cpp* extension. Figure 4.6 shows these C++ references within the standard library files.

Searching for these *.cpp* files does not find them anywhere on the image. It can be assumed then that these files are not standalone library files which can be extracted and audited. As such, no further steps were taken with regard to dependency references to *.cpp* files.

A bash script was created to automate this process and is shown in Figure 4.8. The bash script looked for dependencies in each *.so* file, and dumped the data to a CSV file. Excel

```
Select Ubuntu
spectre@IDENTITY-KNOWN [~/repo/MS_Thesis/auto_dep_walk/pow_dep_L1]$strings 7993-libc.so | egrep ".cpp|\.so"
ld-android.so
libdl.so
libc.so
bionic/libc/bionic/bionic_allocator.cpp
bionic/libc/bionic/system_property_set.cpp
bionic/libc/async_safe/async_safe_log.cpp
libnetd_client.so
couldn't open libandroidicu.so: %s
libandroidicu.so
libc_malloc_debug.so
libc_malloc_hooks.so
heaprofd_client.so
pty.cpp
getpriority.cpp
sched_getaffinity.cpp
getentropy.cpp
wmempcpy.cpp
tdestroy.cpp
fortify.cpp
fpclassify.cpp
pthread_key.cpp
pthread_mutex.cpp
bionic_futex.cpp
__libc_current_sigrtmax.cpp
ftw.cpp
new.cpp
iconv.cpp
clearenv.cpp
recv.cpp
sched_getcpu.cpp
icu.cpp
stdio_ext.cpp
gdtoa_support.cpp
lfs64_support.cpp
abort.cpp
assert.cpp
accept.cpp
reboot.cpp
umount.cpp
mount.cpp
```

Figure 4.6. Dependencies of standard library files are largely reliant on *.cpp* files

was used to process the CSV data to identify new dependencies which have not yet been analyzed. An example of the Excel worksheet used is shown in Figure 4.7.

The Excel worksheet uses the Excel functions: *concat*, *countif*, *unique*, *if*, *search*, and *isnumber* functions to create dependency - origin pairs, and find novel dependencies which had not yet been encountered. The novel dependencies were then added into Autopsy as new keywords to extract the relevant *.so* files from the forensic image.

This file discovery process was repeated to find the dependencies of dependencies until I reached a total of 558 origin-dependency relationships and 36 dependency files. During the final round of file discovery, there were 7 more novel dependencies that could have been followed, however this round of discovery resulted in many of the same dependencies being found from previous rounds. With the number of files already included, there is sufficient

	A	B	C	D	E	F	G	M	N	O	Q	R	S
1	id	name	origin	dependency	CONCAT	UNIQUE		Leaves	Unique Leaves		Are .so files?	Novel?	
2	1	libc_malloc_debug.so	libc_mallk	libunwindstack	libc_malloc_debug.so,lib	CONCAT		libunwind:Leaves					
3	2	libc_malloc_debug.so	libc_mallk	libc.so	libc_malloc_debug.so,lib	libc_malloc_debug.s	libc.so	libunwindstack.so			libunwindstack	libunwindstack.so	
4	3	libc_malloc_debug.so	libc_mallk	libm.so	libc_malloc_debug.so,lib	libc_malloc_debug.s	libm.so	libc.so			libc.so		
5	4	libc_malloc_debug.so	libc_mallk	libdl.so	libc_malloc_debug.so,lib	libc_malloc_debug.s	libdl.so	libm.so			libm.so		
6	5	libc_malloc_debug.so	libc_mallk	libc_malloc_det	libc_malloc_debug.so,lib	libc_malloc_debug.s		libdl.so			libdl.so		
7	6	libc_malloc_debug.so	libc_mallk	/system/lib64/lib	libc_malloc_debug.so,s	libc_malloc_debug.s	/system/lib						
8	7	libc_malloc_debug.so	libc_mallk	bionic/libc/asyn	libc_malloc_debug.so,bi	libc_malloc_debug.s	bionic/libc	/system/lib64/libunwir	/system/lib64/libunwindstack.so				
9	8	libc_malloc_debug.so	libc_mallk	/system/lib64/lib	libc_malloc_debug.so,s	libc_malloc_debug.s	/system/lib	bionic/libc/async_safe,					
10	9	libc_malloc_hooks.so	libc_mallk	libc++.so	libc_malloc_hooks.so,lib	libc_malloc_hooks.s	libc++.so	/system/lib64/libc_ma	/system/lib64/libc_malloc_debug.s				
11	10	libc_malloc_hooks.so	libc_mallk	libc.so	libc_malloc_hooks.so,lib	libc_malloc_hooks.s	libc.so	libc++.so			libc++.so		
12	11	libc_malloc_hooks.so	libc_mallk	libm.so	libc_malloc_hooks.so,lib	libc_malloc_hooks.s	libm.so	libbase.so			libbase.so		
13	12	libc_malloc_hooks.so	libc_mallk	libdl.so	libc_malloc_hooks.so,lib	libc_malloc_hooks.s	libdl.so	libprocinfo.so			libprocinfo.so	libprocinfo.so	
14	13	libc_malloc_hooks.so	libc_mallk	libc_malloc_hoc	libc_malloc_hooks.so,lib	libc_malloc_hooks.s		ashmemd_aidl_interfa	ashmemd_aidl	ashmemd_aidl_int			
15	14	heapprof_client.so	heapprof	libbase.so	heapprof_client.so,lib	libc_malloc_hooks.s	libbase.so	libbinder.so			libbinder.so		
16	15	heapprof_client.so	heapprof	libprocinfo.so	heapprof_client.so,lib	heapprof_client.so	libprocinfo	libutils.so			libutils.so		
17	16	heapprof_client.so	heapprof	libunwindstack	heapprof_client.so,lib	heapprof_client.so	libunwind	system/ashmemd/ashr					
18	17	heapprof_client.so	heapprof	libc++.so	heapprof_client.so,lib	heapprof_client.so	libc++.so	liblog.so			liblog.so		
19	18	heapprof_client.so	heapprof	libc.so	heapprof_client.so,lib	heapprof_client.so	libc.so	external/libunwind_llv					
20	19	heapprof_client.so	heapprof	libm.so	heapprof_client.so,lib	heapprof_client.so	libm.so	/system/lib/libc_mallo	/system/lib/libc_malloc_debug.so				
21	20	heapprof_client.so	heapprof	libdl.so	heapprof_client.so,lib	heapprof_client.so	libdl.so	/system/lib/libunwind	/system/lib/libunwindstack.so				
22	21	heapprof_client.so	heapprof	heapprof_client	heapprof_client.so,he	heapprof_client.so		libcgroupprc.so	libcgroupprc	libcgroupprc	libcgroupprc	libcgroupprc	
23	22	heapprof_client.so	heapprof	bionic/libc/asyn	heapprof_client.so,bi	heapprof_client.s	bionic/libc	/system/lib64/libunwir	/system/lib64/libunwindstack.so				

Figure 4.7. Excel worksheet to process dependency data.

power system relevant files to perform automated scans. There are a total of 138 .so files to scan, including duplicate libraries compiled for different architectures. .

Dependency Mapping

The list of dependencies in CSV format generated from the file discovery process was used for creating a dependency map. However, due to the large number of critical power files, as well as their dependencies, it would be unfeasible to draw the diagrams by hand.

For this dependency mapping it should be noted that there are multiple critical power files with the exact same filenames, but seem to serve the same purpose. Take the following examples of files along with their MD5 hashes as shown in Table 4.3.

Table 4.3. Example of similarly named dependency with different contents.

Filepath and filename	MD5 Hash
/system/lib/android.hardware.power@1.0.so	2e4bc658f8315645113e367e0906c4b0
/system/lib64/android.hardware.power@1.0.so	57ccc2aaf5a42ffbb78c80477889eb5e

While both files have the same filename, “android.hardware.power@1.0.so”, they are stored in different locations. More importantly, their contents are also different, as shown by

```

GNU nano 4.8
#!/bin/bash

# Given a directory, scans all file within the directory and pulls out
# any references to .so files. Creates a relationship between
# the reference and the file that it was extracted from

# Set output file name
OUTPUT_FILE=dency_list.csv
ID=1

# Get directory of files needing to be parsed from cmd line
directory_path=$1

# Find all Object files in target directory
files=`ls $directory_path | egrep "*\.so"`

# Prepare output file to be imported into draw.io
#cat drawio_preamble.txt > $OUTPUT_FILE
rm $OUTPUT_FILE
touch $OUTPUT_FILE
#echo "" >> $OUTPUT_FILE

# Create csv headers
echo "id,name,origin,dependency" >> $OUTPUT_FILE

# Process all target files in directory
for file in $files
do
    # Search for all .so and .cpp references in file
    dependencies=`strings $directory_path$file | egrep "\.so|\.cpp"`

    echo "$dependencies"

    # Process all dependencies found per library file
    for dep in $dependencies
    do
        # Clean up numbers inserted by Autopsy into filenames
        filename=`echo $file | awk '{print substr($0,index($0,"-")+1)}'`

        # Print dependency relationship into output file
        echo "$ID,$filename,$filename,$dep" >> $OUTPUT_FILE
        ((ID++))
    done
done

```

Figure 4.8. Script used to pull dependencies from all critical files and sub-dependencies

the difference in their respective MD5 hashes. The different files are also stored in separate directories, “lib” and “lib64”. It is reasonable to assume that one set of files would be used for 32 bit systems, while the other would be used by systems with 64 bit architectures. Regardless of which specific architecture the so file is for, the dependencies should remain consistent

for the various compiled binaries across architectures. As such, dependency mapping only includes one instance of each file and ignore duplicates with separate architectures.

The CSV file of dependencies from the file discovery process was used to create the diagram using Maltego after converting the data to an *.xlsx* format. The finished diagram was checked for accuracy and extra, erroneous data points were manually removed as they were clearly artifacts generated by the data extraction process. An example of an erroneous data point is shown in Figure 4.9.

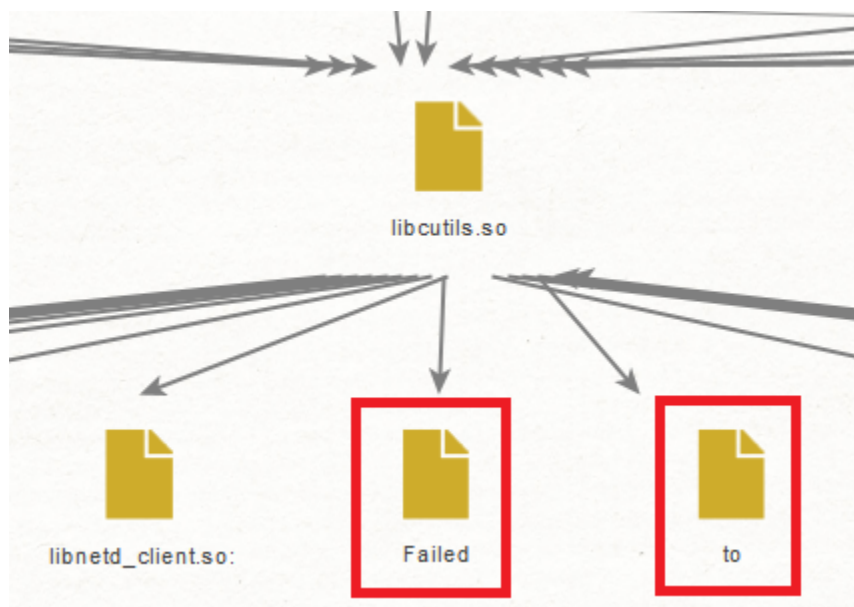


Figure 4.9. Boxed in red, two examples of erroneous dependencies manually removed from final diagram. Erroneous entries were introduced by the bash script’s extraction of dependency information from *.so* files

A few *.so* were randomly selected and their diagrams were manually verified to ensure accuracy. For this research, the following files were checked: *libm.so*, *ld-android.so*, and *libutils.so*. This was accomplished by checking the dependencies of the file being audited were extracted properly from the original *.so* file, then using Maltego’s “Select Children” and “Select Parent” functions to view all dependencies and parents of the audited file itself. With this procedure, a map of all dependencies and how they interact was created and is shown in Figure 4.10.

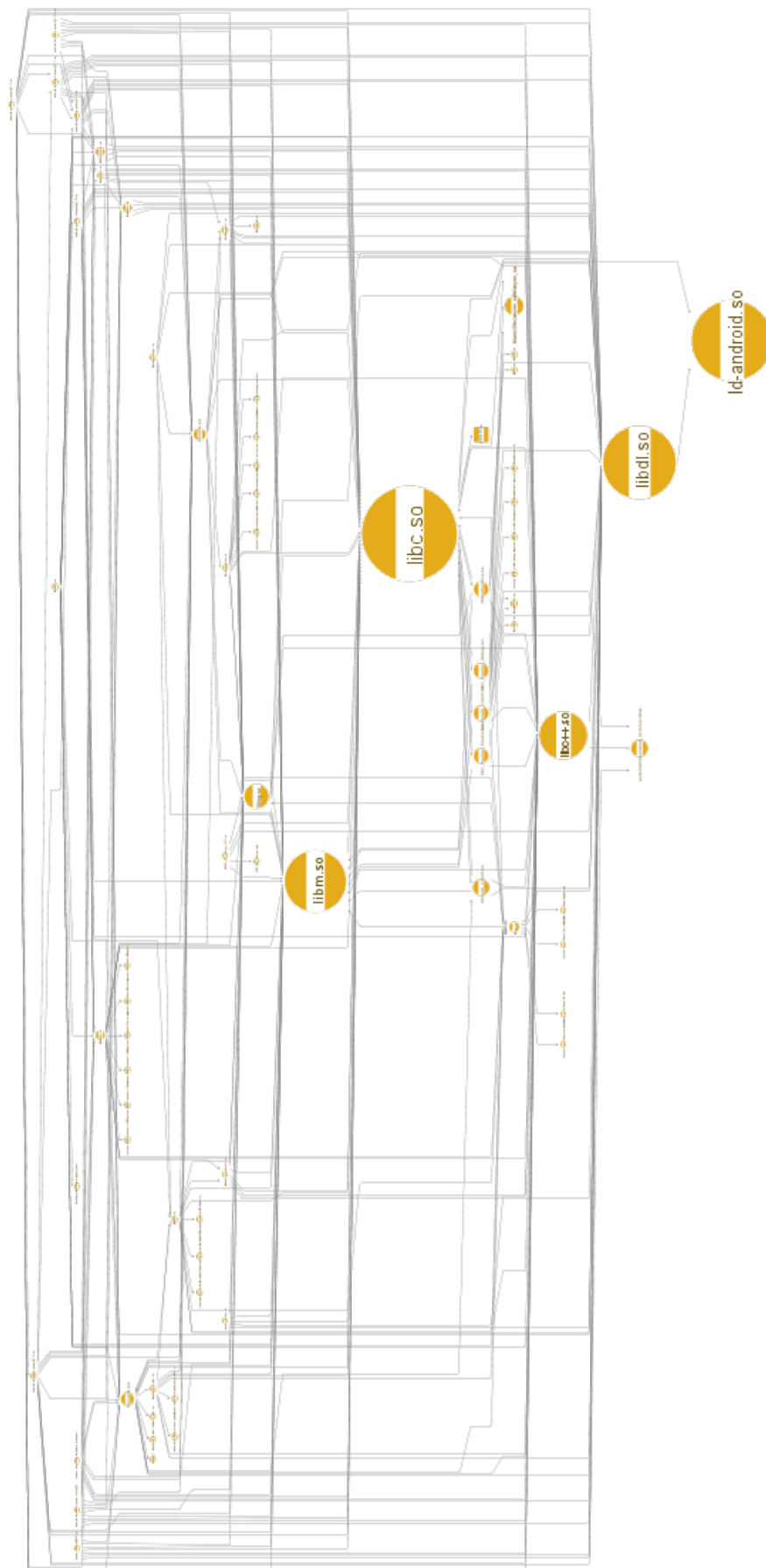


Figure 4.10. Final diagram of interdependencies used to understand the BMS system being audited

With this graph, I identified the top 5 most dependent libraries are the common libraries *libc.so*, *ld-android.so*, *libdl.so*, *libm.so*, and *libc++.so*. These files are the most critical for vulnerability assessment as they are highly referenced by all other power relevant files discovered.

With this diagram, the interdependencies between the critical files are extracted from the mobile device. These diagrams were used to quickly identify which libraries are most commonly used and referenced to gain an understanding of the low level MBS.

File Analysis

Before conducting the audit of these critical files, a validation check was performed to ensure these files contained the relevant MBS drivers. The *6023-android.hardware.power@1.0.so* files was loaded into an online binary disassembler, <https://onlinedisassembler.com/odaweb/>, which claims to support a wide range of Object file formats [22].

From the disassembled file, I noticed a large number of references to a string “HIDL”. HIDL is the “[HAL]... interface description language (IDL) to specify the interface between a HAL and its users” [32]. Considering these were supposed to include the main power system drivers, which should be responsible for the operations of the hardware itself and not necessarily the communication bridge between hardware and the OS, so many references to HIDL was not expected. I found the source code for this file at <https://android.googlesource.com/platform/hardware/interfaces/+master/power/1.0/default/Power.cpp>. An inline comment which states that methods from the *android.hardware.power@1.0.so* file are included in this C++ file. The method names were also consistent between the source code and strings present in the disassembled file. Two examples are shown in Figure 4.12 of the methods *setInteractive* and *powerHint* in Figure 4.11. There is a strong likelihood that the source code seen here is the source code used to compile the binary *.so* files which were believed to be drivers. Review of the source code shows that this code is still quite abstract and simple. The *setInteractive* and *powerHint* methods are good examples as they contain very little logic which would be capable of operating hardware as drivers are supposed to.

0x6851	HIDL::IPower::powerHint::passthrough
0x6876	android.hardware.power@1.0::IPower
0x6899	Null synchronous callback passed.
0x68bb	HIDL::IPower::powerHint::server
0x68db	getPlatformLowPowerStats: _hidl_cb not called,
0x6923	HIDL::IPower::setInteractive::client
0x6948	HIDL::IPower::setInteractive::server
0x696d	HIDL::IPower::getHashChain::passthrough
0x6995	HIDL::IPower::getDebugInfo::passthrough
0x69bd	HIDL::IPower::unlinkToDeath::passthrough
0x69e6	HIDL::IPower::linkToDeath::passthrough
0x6a0d	HIDL::IPower::debug::passthrough

Figure 4.11. Some of the HIDL references present in the file *6023-android.hardware.power@1.0.so*

```

40 // Methods from ::android::hardware::power::V1_0::IPower follow.
41 Return<void> Power::setInteractive(bool interactive) {
42     if (mModule->setInteractive)
43         mModule->setInteractive(mModule, interactive ? 1 : 0);
44     return Void();
45 }
46
47 Return<void> Power::powerHint(PowerHint hint, int32_t data) {
48     int32_t param = data;
49     if (mModule->powerHint) {
50         if (data)
51             mModule->powerHint(mModule, static_cast<power_hint_t>(hint), &param);
52         else
53             mModule->powerHint(mModule, static_cast<power_hint_t>(hint), NULL);
54     }
55     return Void();
56 }

```

Figure 4.12. Method names from source code match strings discovered in the *.so* file

At this stage, it seemed that more discovery was necessary to locate and extract the driver files.

Drivers and Kernel Analysis

Conducting further online research on the Android kernel itself, I found a possible answer as to where the missing power driver files may be located. The Android operating system, based on the Linux kernel, is a monolithic kernel [67].

The Android kernel being monolithic means that all necessary hardware drivers, including the drivers for the battery and power systems, have been compiled and integrated with the core kernel executable itself [67]. The drivers for the battery, charger, and overall power management system that were needed have been compiled into the same binary file, along with the kernel.

I extracted what I believe to be the kernel file from the device image. Internet forum posts claim that the kernel itself is normally stored on the boot partition of the device [35, 41]. Investigating the boot partition on the image of the device reveals a data carved file which seems very likely to be the kernel and seems to corroborate the answers found online. String artifacts found in this carved file, shown in Figure 4.13, refer to the kernel information as well.

This kernel file was extracted and added to the list of 138 critical files for vulnerability scanning.

Scanning and Auditing

There were 139 files in total which were extracted from the device image. These files represent what I believe to be the kernel, hardware drivers, and most of the middleware of C and C++ libraries that are used by higher level applications to interact with the kernel and OS. The files were analyzed using a vulnerability scanner for binary executables.

Unfortunately, there was difficulty in sourcing a free tool capable of performing the vulnerability scanning for these 139 files. While this framework necessitates the user to conduct their own research, due to the limited time of the researchers, it is not possible to

/img_blk0_sda.bin/vol_vol17/\$CarvedFiles						
Table Thumbnail Summary						
Name	▼	C	Size	Mo...	Change...	Location
✖ f0022716.elf			10485760	0000...	0000-00-...	/img_blk0_sda.bin/vol_vol17/\$CarvedFiles/f0022716.elf
<						
Hex Text Application File Metadata Context Results Annotations Other Occurrences						
Page: 1 of 640	Page	← →	Go to Page:		Jump to Offset 0	Launch i
0x00000000:	7F 45 4C 46	02 01 01 00	00 00 00 00	00 00 00 00	.ELF.....	
0x00000010:	03 00 B7 00	01 00 00 00	D0 02 00 00	00 00 00 00	
0x00000020:	40 00 00 00	00 00 00 00	90 08 00 00	00 00 00 00	@.....	
0x00000030:	00 00 00 00	40 00 38 00	04 00 40 00	0E 00 0D 00@.8...@....	
0x00000040:	01 00 00 00	05 00 00 00	00 00 00 00	00 00 00 00	
0x00000050:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
0x00000060:	18 08 00 00	00 00 00 00	18 08 00 00	00 00 00 00	
0x00000070:	08 00 00 00	00 00 00 00	02 00 00 00	04 00 00 00	
0x00000080:	08 07 00 00	00 00 00 00	08 07 00 00	00 00 00 00	
0x00000090:	08 07 00 00	00 00 00 00	F0 00 00 00	00 00 00 00	
0x000000a0:	F0 00 00 00	00 00 00 00	08 00 00 00	00 00 00 00	
0x000000b0:	04 00 00 00	04 00 00 00	B8 02 00 00	00 00 00 00	
0x000000c0:	B8 02 00 00	00 00 00 00	B8 02 00 00	00 00 00 00	
0x000000d0:	18 00 00 00	00 00 00 00	18 00 00 00	00 00 00 00	
0x000000e0:	04 00 00 00	00 00 00 00	50 E5 74 64	04 00 00 00P.td....	
0x000000f0:	44 06 00 00	00 00 00 00	44 06 00 00	00 00 00 00	D.....D.....	
0x00000100:	44 06 00 00	00 00 00 00	2C 00 00 00	00 00 00 00	D.....,.....	
0x00000110:	2C 00 00 00	00 00 00 00	04 00 00 00	00 00 00 00	,.....	
0x00000120:	03 00 00 00	07 00 00 00	00 00 00 00	02 00 00 00	
0x00000130:	06 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
0x00000140:	00 00 00 00	03 00 00 00	04 00 00 00	05 00 00 00	
0x00000150:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
0x00000160:	00 00 00 00	00 00 00 00	00 00 00 00	03 00 07 00	
0x00000170:	D0 02 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
0x00000180:	6A 00 00 00	11 00 F1 FF	00 00 00 00	00 00 00 00	j.....	
0x00000190:	00 00 00 00	00 00 00 00	2E 00 00 00	12 00 07 00	
0x000001a0:	E4 05 00 00	00 00 00 00	54 00 00 00	00 00 00 00T.....	
0x000001b0:	44 00 00 00	12 00 07 00	3C 06 00 00	00 00 00 00	D.....<.....	
0x000001c0:	08 00 00 00	00 00 00 00	01 00 00 00	12 00 07 00	
0x000001d0:	D0 02 00 00	00 00 00 00	9C 00 00 00	00 00 00 00	
0x000001e0:	17 00 00 00	12 00 07 00	6C 03 00 00	00 00 00 00l.....	
0x000001f0:	78 02 00 00	00 00 00 00	00 5F 5F 6B	65 72 6E 65	x.....__kerne	
0x00000200:	6C 5F 67 65	74 74 69 6D	65 6F 66 64	61 79 00 5F	l_gettimeofday._	
0x00000210:	5F 6B 65 72	6E 65 6C 5F	63 6C 6F 63	6B 5F 67 65	_kernel_clock_ge	
0x00000220:	74 74 69 6D	65 00 5F 5F	6B 65 72 6E	65 6C 5F 63	tttime.__kernel_c	
0x00000230:	6C 6F 63 6B	5F 67 65 74	72 65 73 00	5F 5F 6B 65	lock_getres.__ke	
0x00000240:	72 6E 65 6C	5F 72 74 5F	73 69 67 72	65 74 75 72	rnal_rt_sigretur	
0x00000250:	6E 00 6C 69	6E 75 78 2D	76 64 73 6F	2E 73 6F 2E	n.linux-vdso.so.	
0x00000260:	31 00 4C 49	4E 55 58 5F	32 2E 36 2E	33 39 00 00	1.LINUX_2.6.39..	

Figure 4.13. Carved data from *boot* partition contains a file with the ELF binary header and references to a “kernel”

conduct an extensive and comprehensive review of all tools currently available for binary vulnerability scanning. The list of tools that were discovered and attempted are:

- vuls - Difficult to scan arbitrary files; the main purpose is to scan live machines and running services
- vulnscan - Vulnerability database not retrievable
- cwe_checker - Installed, but freezes when the scan starts, no error messages
- bap (Binary Analysis Platform) - Tool for manual static analysis, cannot automatically find vulnerabilities
- BugScam - Unable to process ELF files
- Oversecured - Free scan limit prohibits scanning the necessary files; not designed for arbitrary executable files but APK files

4.2 High Level Vulnerability Results

The vulnerability reports from Oversecured for the target Samsung Device Health Manager application are shown in Table 4.4. The reports for the baseline malicious and benign applications are shown in Tables 4.5 and 4.6 respectively.

Table 4.4. Security scan results from Oversecured for *SamsungDeviceHealthService.apk* file

Category	Instances	Percent of Total
High Severity	12	0.63
Medium Severity	0	0.00
Low Severity	7	0.37

Table 4.5. Security scan results from Oversecured for malicious file

Category	Instances	Percent of Total
High Severity	12	0.04
Medium Severity	100	0.34
Low Severity	188	0.62

Table 4.6. Security scan results from Oversecured for Signal

Category	Instances	Percent of Total
High Severity	41	0.06
Medium Severity	403	0.48
Low Severity	365	0.46

From the raw numbers of vulnerabilities reported by Oversecured, it is difficult to say how many of these flagged high severity vulnerabilities are real. Looking at the Signal application, known for being reasonably secure and privacy focused, the application has more vulnerabilities reported than the others. The number of reported high severity vulnerabilities between the battery application and the malicious application was exactly the same at 12 occurrences. While this framework is supposed to find vulnerabilities in MBS, the number of vulnerabilities being reported in these three applications may indicate that false positives are being reported. As such, these numbers alone are not sufficient in determining whether these are real vulnerabilities.

Another attempt at validating the discovered vulnerabilities was to create a POC for each one to determine if they were truly exploitable security gaps. Oversecured reports vulnerabilities in the manner shown in Figures 4.14 and 4.15.



Figure 4.14. Detailed vulnerability report for a hard coded token vulnerability

It is likely possible for an experienced software developer and security researcher to understand the vulnerable code being depicted in Figures 4.14 and 4.15, understand where this code is located, how this function is accessed within the the program, and have the



Figure 4.15. Detailed vulnerability report for a dynamic broadcast receiver registration

skills necessary to create an exploit which takes advantage of the depicted vulnerability. However, based on the information provided, the researcher faces substantial difficulty in generating a working POC for the vulnerabilities. The researcher has limitations due to available knowledge, skills, and time which prevent them from pursuing the next steps in developing a working POC.

4.3 Low Level Vulnerability Results

139 files were discovered which likely included the relevant drivers, kernel, native libraries, and HAL components. Vulnerability scanning was not successfully performed as the researcher was unable to find an appropriate tool to conduct the scanning. The methodology attempted by applying the framework unfortunately did not produce a list of possible vulnerabilities within the low level components of MBS. Possible alternative methodologies which could be worth exploring, both in the file discovery process as well as the vulnerability scanning process, are suggested in Section 5 to improve the framework's methodology.

5. DISCUSSION

In this section, I discuss how the research question was answered and the framework overall. Challenges faced will also be discussed as well as the limitations and useful aspects of the framework.

5.1 Research Question

The research question for this study was:

“How effective of a framework can I currently create to discovery and identify real vulnerabilities in the power systems of mobile devices?”

From the results, I was able to get a list of possible vulnerabilities from the high level analysis of an Android application. However, this list of vulnerabilities were not validated by creating a POC exploit due to limitations in the researchers skillset, as well as the time available for this work. No vulnerabilities were found for low level analysis of the MBS due to technical limitations faced. Therefore the first hypothesis is accepted, the framework is no better than random chance at identifying vulnerabilities at 50%.

5.2 Validity

With regards to validity of results, this research has a strong level of validity. In determining whether a system contains a vulnerability, creating a working POC exploit would provide strong direct evidence that a detected vulnerability poses a real security threat. Another aspect of this work that had high validity was the acquisition method for the data used. A forensic image of a working device would certainly capture all the files relevant to MBS.

One area of this research which had less validity was the file discovery process used. Using filenames to find relevant MBS files is an imperfect methodology. This method used could have missed the power system relevant components which were not captured with the keyword lists created. This was a known limitation in the work, as the researcher was unable to find literature which could provide better guidance on how to conduct the file discovery. As

such, this methodology was created by the researcher given the knowledge and skills available. However, improvements to the file discovery methodology used are definitely possible, and future research could take a look at more comprehensive and reasonable procedures.

5.3 Viability and Usefulness

With more work, this framework can be improved to become more useful. It can be used as a cost saving measure to allow inexperienced IT staff and researchers to perform the bulk of file discovery work, saving senior staff many hours. The resulting files can then be passed on to more senior staff and experienced researchers to conduct the vulnerability analysis and determine true from false positives.

The discovery process developed for the case study, while limited, does seem capable of finding some of the necessary dependencies and libraries related to MBS. In addition, the bash script and Excel worksheet have been made available on Github. These tools can be used by other researchers to automate the file discovery process.

5.4 Challenges

Valuable lessons were learned about the power system of mobile devices as well as the proposed framework. These lessons learned can guide future research work in the domain.

File discovery was more difficult than anticipated. The resulting procedure used to discover files is limited and could be improved by using an application to pull running processes from a running device.

During the course of this research, there were many tools available to assess vulnerabilities in high levels of the architecture. Finding a relevant APK file is also relatively simple, especially when a specific target application is already in mind. Tools are available which can automate the process of finding vulnerabilities, however, the tools' reporting capabilities only results in a list of possible vulnerabilities. It is up to the user to proceed from there and develop a method to validate each reported vulnerability.

For the investigation of low level files, a limitation that was encountered by the researcher was the limited availability of tools needed to conduct the vulnerability scan. Two possible

reasons for this are 1) Android being an open source system means that there is no need for binary vulnerability scanning tools when the source code could be scanned instead, and 2) There are more pressing issues with regards to mobile security, primarily third-party application security, that vendors are focusing on developing tools to address instead of open source shared libraries. From this work, I suggest that vulnerability analysis for low level files be done by tracing them back to their original source code. The source code can then be scanned with the much more available source code vulnerability scanners. Validation of these vulnerabilities may still be required depending on the tool used.

6. CONCLUSION

This work has established a general auditing framework that can be used to help manufacturers and IT staff identify possible vulnerabilities within their products power systems.

The contributions of this work are listed in Table 6.1.

Table 6.1. Contributions of this work and the discoveries that have been made toward each

Goal	Contribution
Proposal of a novel auditing framework designed to ensure the security of mobile battery systems.	The proposed framework is limited in ensuring security of mobile battery systems. However the proposed framework has produced a reasonable method of discovering and analyzing mobile data relevant to MBS security which has not been done before.
Creation of a practical set of guidelines that can be followed by academic and independent security researchers interested in examining mobile subsystems for possible vulnerabilities.	A detailed procedure of how to apply this framework has been created. The steps taken during the case study of a Samsung device can be used in future research as a starting point for how to investigate other devices. This work enables future research and the findings in the case study can guide new researchers into more promising directions.
Identify components related to MBS for Android devices.	This work has identified various components related to the mobile battery system that has never been done before. Some of the shared libraries, relevant drivers, and applications were discovered, identified, and extracted. New methods for investigating low level files which could be more promising were also proposed for future work.
A process to create diagrams of the intercomponent communication to understand BMS and highlight possible high-risk areas of vulnerability.	A detailed diagram of the interrelated dependencies for various components and libraries used by the power system and management system was created. The methodology used to create this diagram has not been seen before and can be scaled up easily through the use of the scripts and procedures used.

The framework has a limited ability to discover vulnerabilities. While it does not discover vulnerabilities, it does provide a reasonable method to discover relevant files and does obtain a list of possible vulnerabilities which can be further analyzed.

This research has the potential to guide different forms of future research in the BMS area. Future work in this area can look at other methodologies at file discovery and vulnerability

assessment. An improved file discovery methodology could use a custom application to pull running processes from a device to find power related services. Back tracing these services to relevant power system files could then be done to perform file discovery. A new method of vulnerability analysis could also be attempted in future research. This new method would find the `.so` files and trace them back to their original, publicly available source code. The more readily available source code vulnerability scanners could then be used to scan the original source code of each library file instead of using binary vulnerability scanners. Future work could also be done to examine the effectiveness of currently available and modern tools in conducting vulnerability assessments. Other possible future work includes examining the non-executable power related files on mobile devices, such as databases and logs, for sensitive data which could pose a security and privacy issue. Finally, research could also look into the development of new tools which are capable of vulnerability analysis for mobile devices.

REFERENCES

- [1] URL: <https://www.virustotal.com/>.
- [2] *AccuBattery - Apps on Google Play*. URL: .
- [3] *Advanced Configuration and Power Interface (ACPI) Specification*. 2019.
- [4] Indrasena R Aenugu et al. “Battery data management and analytics platform using blockchain technology”. In: *2020 IEEE Transportation Electrification Conference & Expo (ITEC)*. IEEE. 2020, pp. 153–157.
- [5] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. “SV-AF—a security vulnerability analysis framework”. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2016, pp. 219–229.
- [6] Mike Andrews and James A Whittaker. *How to break web software: Functional and security testing of web applications and web services*. Addison-Wesley Professional, 2006.
- [7] *Android mainline Drivers*. URL: <https://android.googlesource.com/kernel/common/refs/heads/android-mainline/drivers/>.
- [8] *Automated Mobile App Security*. URL: <https://oversecured.com/>.
- [9] K Nasihin bin Baharin et al. “Third party security audit procedure for network environment”. In: *4th National Conference of Telecommunication Technology, 2003. NCTT 2003 Proceedings*. IEEE. 2003, pp. 26–30.
- [10] *Battery - Apps on Google Play*. URL: .
- [11] *Battery Doctor - Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.aee.tools.batterysaver>.
- [12] *Battery HD - Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=ch.smalltech.battery.free>.
- [13] *BatteryManager: Android Developers*. URL: <https://developer.android.com/reference/android/os/BatteryManager>.

- [14] Gomanth Bere et al. “Blockchain-Based Firmware Security Check and Recovery for Battery Management Systems”. In: *2020 IEEE Transportation Electrification Conference & Expo (ITEC)*. IEEE. 2020, pp. 262–266.
- [15] Bruno Blanchet et al. “A static analyzer for large safety-critical software”. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 196–207.
- [16] Timothy K Buennemeyer et al. “Battery exhaustion attack detection with small hand-held mobile computers”. In: *2007 IEEE International Conference on Portable Information Devices*. IEEE. 2007, pp. 1–5.
- [17] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. “Achievements and challenges in software reverse engineering”. In: *Communications of the ACM* 54.4 (2011), pp. 142–151.
- [18] John Chatzakis et al. “Designing a new generalized battery management system”. In: *IEEE transactions on Industrial Electronics* 50.5 (2003), pp. 990–999.
- [19] Ting Chen et al. “Silent Battery Draining Attack Against Android Systems by Subverting Doze Mode”. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2016, pp. 1–6.
- [20] Song Ci et al. “Dynamic reconfigurable multi-cell battery: A novel approach to improve battery performance”. In: *2012 Twenty-Seventh Annual IEEE Applied Power Electronics Conference and Exposition (APEC)*. IEEE. 2012, pp. 439–442.
- [21] *CVE-2011-1149 Detail*. URL: <https://nvd.nist.gov/vuln/detail/CVE-2011-1149>.
- [22] *Disassembler.io Online Disassembler*. URL: <https://onlinedisassembler.com/static/home/index.html>.
- [23] Daniel H Doughty and E Peter Roth. “A general discussion of Li ion battery safety”. In: *Electrochemical Society Interface* 21.2 (2012), p. 37.
- [24] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.
- [25] Tasnimun Faika et al. “A blockchain-based Internet of Things (IoT) network for security-enhanced wireless battery management systems”. In: *2019 IEEE Industry Applications Society Annual Meeting*. IEEE. 2019, pp. 1–6.

- [26] Kim Fenrich. “Securing your control system: the” CIA triad” is a widely used benchmark for evaluating information system security effectiveness”. In: *Power Engineering* 112.2 (2008), pp. 44–49.
- [27] Ugo Fiore et al. “Multimedia-based battery drain attacks for android devices”. In: *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. IEEE. 2014, pp. 145–150.
- [28] Bogdan Cristian Florea and Dragos Daniel Taralunga. “Blockchain IoT for Smart Electric Vehicles Battery Management”. In: *Sustainability* 12.10 (2020), p. 3984.
- [29] Xing Gao et al. “E-Android: A new energy profiling tool for smartphones”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 492–502.
- [30] Sujata Garera and Aviel D Rubin. “An independent audit framework for software dependent voting systems”. In: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007, pp. 256–265.
- [31] Jon Heffley and Pascal Meunier. “Can source code auditing software identify common vulnerabilities and be used to evaluate software security?” In: *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*. IEEE. 2004, 10–pp.
- [32] *HIDL: Android Open Source Project*. URL: <https://source.android.com/devices/architecture/hidl>.
- [33] Frank Hofmann. *Understanding the ELF File Format*. 1968. URL: .
- [34] Greg Hoglund and Gary McGraw. *Exploiting software: How to break code*. Pearson Education India, 2004.
- [35] Hypoturtle. *Where is the kernel image on an Android device*. 2012. URL: <https://forum.xda-developers.com/t/where-is-the-kernel-image-on-an-android-device.1790941/>.
- [36] Grant A Jacoby and NathanielJ Davis. “Battery-based intrusion detection”. In: *IEEE Global Telecommunications Conference, 2004. GLOBECOM'04*. Vol. 4. IEEE. 2004, pp. 2250–2255.
- [37] A Jossen et al. “Reliable battery operation—a challenge for the battery management system”. In: *Journal of Power Sources* 84.2 (1999), pp. 283–286.

- [38] *Kaspersky Battery Life: Saver Booster - Apps on Google Play*. URL: <https://play.google.com/store/apps/details?id=com.kaspersky.batterysaver>.
- [39] Chol-Ho Kim et al. “A modularized two-stage charge equalizer with cell selection switches for series-connected lithium-ion battery string in an HEV”. In: *IEEE Transactions on Power Electronics* 27.8 (2012), pp. 3764–3774.
- [40] Taesic Kim et al. “An overview of cyber-physical security of battery management systems and adoption of blockchain technology”. In: *IEEE Journal of Emerging and Selected Topics in Power Electronics* (2020).
- [41] kwagjj. *linux kernel not seen from android filesystem?* 2017. URL: <https://android.stackexchange.com/questions/137937/linux-kernel-not-seen-from-android-filesystem>.
- [42] Minkyu Lee et al. “Wireless battery management system”. In: *2013 World Electric Vehicle Symposium and Exhibition (EVS27)*. IEEE. 2013, pp. 1–5.
- [43] Abner Li. *You can now easily reset Adaptive Brightness in Android 9 Pie - 9to5Google*. 2019. URL: <https://9to5google.com/2019/01/15/android-pie-adaptive-brightness-reset/>.
- [44] Edward C Lo and Mike Marchand. “Security audit: a case study [information systems]”. In: *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No. 04CH37513)*. Vol. 1. IEEE. 2004, pp. 193–196.
- [45] Anthony Bahadir Lopez et al. “A security perspective on battery systems of the internet of things”. In: *Journal of Hardware and Systems Security* 1.2 (2017), pp. 188–199.
- [46] *Managing Boot Time: Android Open Source Project*. URL: .
- [47] Huasong Meng et al. “A survey of Android exploits in the wild”. In: *Computers & Security* 76 (2018), pp. 71–91.
- [48] Kit Murdock et al. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P’20)*. 2020.
- [49] Alyssa Newcomb. *Samsung Finally Explains the Galaxy Note 7 Exploding Battery Mess*. 2017.
- [50] Atefeh Nirumand, Bahman Zamani, and Behrouz Tork Ladani. “VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique”. In: *Software: Practice and Experience* 49.1 (2019), pp. 70–99.

- [51] *Notifications Overview: Android Developers*. URL: <https://developer.android.com/guide/topics/ui/notifiers/notifications>.
- [52] Justin J Ochoa et al. “Blockchain-as-a-Service (BaaS) for Battery Energy Storage Systems”. In: *2020 IEEE Texas Power and Energy Conference (TPEC)*. IEEE. 2020, pp. 1–6.
- [53] Cyril Onwubiko. “A security audit framework for security management in the enterprise”. In: *International Conference on Global Security, Safety, and Sustainability*. Springer. 2009, pp. 9–17.
- [54] *Optimize for battery life: Android Developers*. URL: <https://developer.android.com/topic/performance/power>.
- [55] Teresa Pereira and Henrique Santos. “A security audit framework to manage Information system security”. In: *International Conference on Global Security, Safety, and Sustainability*. Springer. 2010, pp. 9–18.
- [56] *Platform Architecture*. URL: <https://developer.android.com/guide/platform>.
- [57] Gregory L Plett. “Extended Kalman filtering for battery management systems of LiPB-based HEV battery packs: Part 2. Modeling and identification”. In: *Journal of power sources* 134.2 (2004), pp. 262–276.
- [58] Bruce Potter and Gary McGraw. “Software security testing”. In: *IEEE Security & Privacy* 2.5 (2004), pp. 81–85.
- [59] *PowerManager: Android Developers*. URL: <https://developer.android.com/reference/android/os/PowerManager>.
- [60] *Product Partitions: Android Open Source Project*. URL: <https://source.android.com/devices/bootloader/partitions/product-partitions>.
- [61] *Quick Android Review Kit*. URL: <https://github.com/linkedin/qark>.
- [62] Khaled Rabieh et al. “A secure and cloud-based medical records access scheme for on-road emergencies”. In: *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE. 2018, pp. 1–8.
- [63] Radmilo Racic, Denys Ma, and Hao Chen. “Exploiting MMS vulnerabilities to stealthily exhaust mobile phone’s battery”. In: *2006 Securecomm and Workshops*. IEEE. 2006, pp. 1–10.

- [64] Habiballah Rahimi-Eichi et al. “Battery management system: An overview of its application in the smart grid and electric vehicles”. In: *IEEE Industrial Electronics Magazine* 7.2 (2013), pp. 4–16.
- [65] *Rules on how to access information in sysfs*. URL: <https://www.kernel.org/doc/html/latest/admin-guide/sysfs-rules.html>.
- [66] Cody Shell et al. “Implementation of a wireless battery management system (WBMS)”. In: *2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings*. IEEE. 2015, pp. 1954–1959.
- [67] Gary Sims. *What is a kernel - Gary explains*. 2016. URL: <https://www.androidauthority.com/what-is-a-kernel-gary-explains-681744/>.
- [68] *Smart Battery Data Specification*. 1998.
- [69] Frank Stajano and Ross Anderson. “The resurrecting duckling: Security issues for ad-hoc wireless networks”. In: *International workshop on security protocols*. Springer. 1999, pp. 172–182.
- [70] *sysfs(5) — Linux manual page*. URL: <https://man7.org/linux/man-pages/man5/sysfs.5.html>.
- [71] *Toasts overview: Android Developers*. URL: <https://developer.android.com/guide/topics/ui/notifiers/toasts>.
- [72] Enis Ulqinaku, Julinda Stefa, and Alessandro Mei. “Scan-and-Pay on Android is Dangerous”. In: *arXiv preprint arXiv:1905.10141* (2019).
- [73] David Wagner and R Dean. “Intrusion detection via static analysis”. In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE. 2001, pp. 156–168.
- [74] *What are .a and .so files?* 2012. URL: <https://stackoverflow.com/questions/9809213/what-are-a-and-so-files>.
- [75] *What is Device care and how do I use it?* URL: <https://www.samsung.com/uk/support/mobile-devices/how-do-i-use-device-care/>.
- [76] Yury Zhauniarovich et al. “Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications”. In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. 2015, pp. 37–48.

A. APPENDIX

A.1 Device Info

Table A.1. Device information for Samsung A50 used in case study

Device Name	Galaxy A50
Model Number	SM-A505
Serial Number	R58M84CX1QE
Android version	10
Baseband version	A505GDXU5BTC4
Kernel Version	4.14.62-RefinedKernel.QQ #3 Wed Mar 18 14:02:14 IST 2020
Build number	QO1A.190711.020.A505GUBU5BTC8
SE For Android Status	Enforcing; SEPF_SM-A505G_10_0010 Tue Mar 24 04:15:09 2020
Knox version	Knox 3.5 Knox API level 31 TIMA 4.1.0
Security software version	ASKS v3.1 Release 20200120 ADO v3.0 Release 20191001 SMR Mar-2020 Release 1
Android security patch level	March 1, 2020

A.2 Tools used

Acquisition Software

UFED Cellebrite 7.42.0.82 UFED

Forensic Analysis Software

Autopsy 4.17.0

bash - 5.0.17(1)-release

On Windows Subsystem for Linux - Ubuntu running - 4.4.0-19041-Microsoft #488-Microsoft

Mon Sep 01 13:43:00 PST 2020 x86_64 x86_64 x86_64 GNU/Linux

Diagramming and Reporting Software

Microsoft Excel

Maltego Community Edition v4.2.15

Audit and Analysis

Oversecured - <https://oversecured.com/>

Online dissassembler - <https://onlinedisassembler.com>