Evaluating State-of-the-Art Free and Open Source Static Analysis Tools against Buffer Errors in Android Apps

***Abstract*—Android is the dominant platform in mobile app markets, and enhancing its apps security is a considerable area of research. Android malware (introduced intentionally by developers) has been well studied and many tools are available to detect them. However, little attention has been directed to address vulnerabilities caused unintentionally by developers in Android apps. Static analysis has been one way to detect such vulnerabilities in traditional desktop and server side desktop. Therefore, our research aims at assessing static analysis techniques that could be used by Android developers. Our preliminary analysis revealed that `Buffer Errors` are the most frequent type of vulnerabilities that threaten Android apps. Also, we found that `Buffer Errors` in Android apps have the highest risk on Android that affects data integrity, confidentiality, and availability. Our main study therefore tested whether state-of-the-art static analysis techniques could detect `Buffer Errors` in Android apps. We investigated 17 static analysis techniques that are designed to detect `Buffer Errors`. The study shows that the free and open source state-of-the-art static analysis tools do not efficiently discover `Buffer Error` vulnerabilities in Android apps. We analyzed the techniques carefully to see why they could not discover `Buffer Errors` and found that the lack of semantic analysis capabilities, inapplicability to Android apps, and the gap between native code and other contexts were some of the reasons. Thus, we concluded that there is a need to build better free and open source static analysis tools for detecting `Buffer Errors` in Android apps.**

I. Introduction

Android dominates the mobile market as it has gained tremendous popularity recently [1]. The main Android market, Google Play Store, included 2.4 million apps and 65 billion downloads as of September 2016 [2][3]. Users assume that app markets guarantee the security of offered apps [4]. Yet, app markets typically do not fully ensure the security of the apps they offer [5]. In fact, apps hosted on these markets may possess malware or vulnerabilities (that were introduced unintentionally by developers). A study by Symantec shows that 17% of all Android apps were actually malware in disguise [6]. Yet, every app could contain an unintentional vulnerability. Such vulnerabilities could be exploited by attackers to leak private data, modify the software or data, or deny availability of systems and data causing substantial economic loss.

Studies indicate that the number of vulnerabilities are rapidly rising [7][8]. According to Risk Based Security [7], in 2015 the number of vulnerabilities increased by 77% compared to the vulnerabilities reported in 2011. Another study showed that nearly 75% of tested mobile apps showed at least one critical or high-severity security vulnerability [9].

Studying Android vulnerabilities is therefore a very important area of research. Beside the security concerns, Android apps that have vulnerabilities are likely to get negative reviews and being abandoned by users [10]. Thus, Android developers need to ensure that their apps are secure. In the desktop and server side software, static analysis tools are a common proactive method to find security vulnerabilities in source code, early during the coding phase [11]. This leads to cost savings (a key benefit of static analysis tools), as the earlier a vulnerability is detected, the cheaper it is to fix [12]. In fact, applying a static analysis approach to Android apps to detect vulnerabilities could potentially ensure quality and reliability of the apps and hence the app market. Thus, static analysis tools are common in both maintenance and evolution of software.

However, we do not know how effective static analysis tools are in detecting vulnerabilities in Android apps. *This research, thus, aims to study state-of-the-art static analysis techniques that can be used to detect Android vulnerabilities during the coding process.* In our study we ask the following RQs.

Motivational RQ: What are the most common vulnerabilities in Android apps? From data in the National Vulnerability Database (NVD) [13], we found that `Buffer Errors` are the most frequent vulnerability that happens in Android apps. They also have the highest risk that compromises integrity, confidentiality, and availability.

Case study RQ: Are state-of-the-art static analysis tools for `Buffer Errors` able to detect vulnerabilities reported in the wild for Android apps? In this study, we investigated 17 static analysis techniques that could discover `Buffer Errors`. Out of the 17, we tested 6 free and open source static analysis tools on 9 real world `Buffer Errors`, and it was found that none of the studied tools could efficiently detect the reported `Buffer Errors`.

All our empirical data (including the vulnerabilities from NVD and the Android apps with the vulnerabilities) are available for download [14].

II. Motivational RQ: What are the most common vulnerabilities in Android apps?

A. Motivation

Since examining the efficiency of static analysis tools on all types of vulnerabilities is beyond the scope of a single study, we wanted to find the most frequent type of vulnerability that occur in Android apps. Therefore, in this RQ we identify Android vulnerability trends by examining the published vulnerabilities in the wild. Software security vulnerability records are maintained by multiple security vulnerability databases, such as NVD, Open Source Vulnerability Database (OSVDB) [15], CERT [16], and Bugtraq (BID) [17]. These databases utilize the Common Vulnerabilities and Exposures Identifier (CVE-ID) [18], which is a unique number for a publicly recognized security vulnerability, as an identifier of a vulnerability record. NVD is synchronized with CVE-ID and its records are mainly based on CVE information. Thus, NVD is preferred by researchers as it is the most comprehensive database. Therefore, NVD was selected to construct the dataset for this RQ.

B. Methodology

1) Data Gathering: In this phase, the relevant vulnerability records of Android from 2008 (when Android was first released) to 2015 have been extracted from NVD using an automated web-scraping tool. The "Android" keyword was used to filter the NVD database and extract relevant Android vulnerability reports. For each vulnerability record, the CVE-ID, original release date, last revised date, description, CVSS v2 base score, impact score, exploitability score, access vector, access complexity, authentication, impact type, and vulnerability type were retrieved. 2,089 records were extracted in the initial data gathering.

2) Removing Irrelevant Data: The dataset has been cleaned in order to get more accurate results. All 2,089 entries have been examined manually to check whether they are accurately related to Android; if not, the record is excluded from the dataset. Four records were found unrelated to Android: CVE-2015-3906, CVE-2015-3815, CVE-2012-1344, and CVE-2011-1001.

3) Data Processing: In this phase, we traced each vulnerability record manually in order to collect more information, such as whether the vulnerability has been confirmed or patched, and how it was discovered. This kind of information was obtained from the software vendor websites and other resources, such as BID, OSVDB, CERT, and Japan Vulnerability Note (JVN) [19]. Based on the obtained information, we further sanitized the dataset. We classified the vulnerability records into multiple categories. Vulnerability record categories with hyperlinks of the acquired information could be found in our dataset in columns "Record Category" and "Record Category URL". The categories are as follow:

- **Confirmed and patched:** An advisory has been published by the vendor that explains the vulnerability and provides patching information. We found that 556 vulnerability records fall into this category.
- **Reported and patched:** The vulnerability has been reported to the vendor and patches were released. We found 67 vulnerability records fall into this category.
- **Proof of concept and patched:** A proof of concept has been demonstrated by the reporter, and it is indicated that the vendor has released patches. We found 5 vulnerability records belong to this category.
- **Confirmed but not patched:** The vulnerability has been confirmed by the vendor, however no patches were provided. We found 5 vulnerability records belong to this category.
- **Proof of concept but not patched:** A proof of concept has been demonstrated by the discoverer, and it is indicated that the vendor has not released patches. We found 30 vulnerability records fit into this category.

While categories that are excluded from the study are:

- **Not enough information:** In case that the vulnerability report does not include enough information, such as a proof of concept, patching information, or confirming from the vendor, then the record is excluded from the dataset. 33 records were excluded as they do not include enough information.
- **Large-scale experiment:** An automated large scale experiment has been conducted by Dormann [20] to test whether Android apps properly validate SSL certificates provided by HTTPS connections. The study was conducted on 23668 Android apps and 13 Android libraries, as a result 5.9% (1379 apps and 10 libraries) have been reported to be vulnerable. 23.2% of the tested apps were found vulnerable because of a vulnerability in the libraries. These reports have been found in NVD from 9/8/2014 to 10/29/2014 as `Cryptographic Issue` vulnerability type. Those records were excluded in order to produce more balanced and meaningful results. 1389 vulnerability records belong to this category.

4) Data Classification: After cleaning the dataset, 663 records reminded. Next, the vulnerability records were examined manually in order to categorize them into two groups: Android platform vulnerabilities, and Android app vulnerabilities. Android app vulnerabilities are any vulnerability that could be triggered within Android apps. All other vulnerabilities are considered as Android platform vulnerabilities, for example, vulnerabilities that reside in the Linux kernel of the Android system.

C. Results

Android platform related vulnerabilities are 187, and Android app vulnerabilities are 476, representing 72% of the all vulnerabilities in our cleaned dataset (663).

1) Vulnerability Type Trend: We aim to tackle vulnerabilities at app level, but not at the OS level. Thus, an important question of this empirical study is "what is the most frequent type of vulnerability that occurs historically in Android apps". To reveal the trend of Android vulnerabilities, NVD classification of vulnerability type was utilized, which

is based on the Common Weakness Enumeration (CWE) [21]. CWE is a list of software flaw types that is maintained by the MITRE Corporation and used by security organizations and researchers. In the studied dataset, there are 24 different type of flaws that occurred in the Android platform, and 20 types of them occurred in the Android apps. The percentage of each vulnerability type has been calculated in order to uncover the trending vulnerability in Android apps. As shown in Table I, `Buffer Error` is the dominant vulnerability in Android apps – 28.6% of all discovered Android app vulnerabilities. `Permissions, Privileges, and Access Control` (18.1%) and `Information Leak/Disclosure` (11.3%) vulnerabilities are the second and third most frequent vulnerabilities in Android ecosystem respectively. `Insufficient Information` (12.4%) type indicates that there is insufficient information about the vulnerability to categorize it. Such case usually happens when vendors confirm a vulnerability but decline to release certain details about the vulnerability.

Although, `Permissions, Privileges, and Access Control` and `Information Leak/Disclosure` are dominating the Android security research community [22], rarely are `Buffer Error` vulnerabilities related to Android apps being discussed (See Section VII-B for more details). Since `Buffer Error` is the most frequent vulnerability that occurred historically in Android apps, further analysis is needed to recognize how severe and dangerous `Buffer Error` vulnerabilities are.

Vulnerability Type	CWE-ID	Total	%
Buffer Errors	CWE-119	136	28.6%
Permissions, Privileges, and Access Control	CWE-264	86	18.1%
Insufficient Information	NVD-CWE-noinfo	59	12.4%
Information Leak / Disclosure	CWE-200	54	11.3%
Cryptographic Issues	CWE-310	28	5.9%
Input Validation	CWE-20	23	4.8%
Cross-Site Scripting (XSS)	CWE-79	16	3.4%
Numeric Errors	CWE-189	15	3.2%
Path Traversal	CWE-22	15	3.2%
Other	NVD-CWE-Other	11	2.3%
Resource Management Errors	CWE-399	8	1.7%
Code Injection	CWE-94	7	1.5%
Authentication Issues	CWE-287	4	0.8%
Cross-Site Request Forgery (CSRF)	CWE-352	3	0.6%
Improper Access Control	CWE-284	3	0.6%
Security Features	CWE-254	3	0.6%
Credentials Management	CWE-255	3	0.6%
Code	CWE-17	2	0.4%
Data Handling	CWE-19	1	0.2%
OS Command Injections	CWE-78	1	0.2%

TABLE I: Vulnerability Types Distribution in Android Apps

2) Buffer Error Severity Trend: In order to analyze how severe discovered `Buffer Error` vulnerabilities are, the Common Vulnerability Scoring System (CVSS) [23] was employed. CVSS is an open standard used to assess the severity of security vulnerabilities, and it is platform and technology independent. We used CVSS Base score version 2 standards, which was calculated in NVD database. We found during our analysis that 97% of the `Buffer Error` vulnerabilities have high risk. These results show that `Buffer Errors` have serious implications on the security of Android apps and its end users. Also, 99.26% of `Buffer Errors` in Android apps are remotely exploitable and no authentication is required at all. Thus, it could be concluded that the `Buffer Errors` in Android apps are easy to exploit. Finally, we found that 95.6% of `Buffer Error` vulnerabilities completely impacted the target app in terms of confidentiality, integrity, and the availability.

Due to space constraints, we omitted some aspects of `Buffer Errors` analysis. However, all details could be found here [14]

> Buffer Errors are the most common vulnerability in Android apps. They also have the highest risk that compromises integrity, confidentiality, and availability. As a result, we concluded that we need to focus our main study on `Buffer Errors`.

Determining the most impactful vulnerability affecting Android apps (which involved hundreds of hours of manual work), is not the focus of the study. However, we believe that the empirical evidence shown above not only motivates the rest of our study, but is a useful contribution to the research community that looks to solve the issue of `Buffer Errors` in Android apps.

III. Background

Before we examine state-of-the-art static analysis tools, we want to give some background about the Android app architecture and `Buffer Errors`.

A. Android Application Architecture

Android apps are unlike standard applications in two important ways. First, Android apps run in a security sandbox to manage application memory. Hence, each app represents a different process with a unique UID, and it is executed in isolation from other apps with restricted permissions. Second, Android apps are framework-based and event-driven. Thus, Android apps and Android OS communicate through callbacks. Android apps have no single entry point, the so called main method, though there are multiple entry points. These entry points represent components that can be used by other apps if needed or called by the Android OS. There are four types of components in Android apps: activities are a single focused user interface, services run background tasks, content providers act as a database storage, and broadcast receivers listen for framework events. The components of an application could be executed in any order, and each component has a complete lifecycle [24].

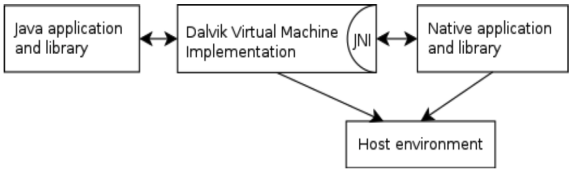

Fig. 1: JNI in Android Applications [27]

Android apps are typically written in the Java programming language. However, the Android Native Development Kit (NDK) that was provided by Google, allows developers to implement Android apps using native programming languages, such as C and C++. In fact, native code allows developers to use existing third party libraries, and allows hardware specific optimization of performance critical code. Indeed, Zhou et al. [25] in 2012 report that 4.52% of Android applications use native code. In 2014, Qian et al. indicted that 16.46% of tested Android apps uses native code [26]. NDK enables Android developers to combine native code with an Android app using Java Native Interface (JNI). JNI is a foreign function interface (FFI) by which a program written in Java can call routines or make use of services written in native code, such as C/C++ and assembly; yet, JNI can be also utilized to invoke Java objects from native code.

Both the Java code and native code of an Android app run within the same process. Thus, the native code still adheres to the entire application permissions set in the manifest file. As shown in Figure 1 from [27], the Java code is managed and executed by the Dalvik Virtual Machine (DVM) [1], while native code is not restricted to DVM and manages itself throughout the lifetime of the application. This requires further responsibilities on developers, such as memory management tasks, hence `Buffer Errors` occur in native code due to improper memory management. To execute Android apps effectively, the native components are also expected to carefully interact with Java components. If this interaction is not appropriately managed, the native components can cause errors within the application that could crash the entire virtual machine [28].

B. Buffer Errors

Android supports multiple processor architectures, such as ARM, Intel x86, and MIPS. However, the common CPU in Android is ARM architecture based [29]. Like other architectures, ARM based CPU `Buffer Errors` usually occur on the stack or on the heap. Stack is adjacent blocks of memory that is controlled by Android OS, and it is used for storage of local variables, and return addresses, and passing extra arguments to subroutines when there are inadequate argument registers available. Stack-based buffer overflow happens when an application writes extra data outside an intended memory address on the program's call stack. This type of vulnerability is a serious one as the stack contains the return addresses for all function calls. If the affected application is running with higher privileges, or receives data from untrusted sources then the defect is a potential security vulnerability. Stack-based buffer overflow could result in corrupting local variables, crashing the application, or executing arbitrary code.

The heap memory is managed by the process itself by particular APIs in Bionic library, which uses Doug Lea's dlmalloc allocator, such as realloc(), calloc(), malloc(), and free() [30]. Heap memory is used to create dynamic data objects, and to store objects that must live longer than one function's lifetime [31]. Buffers allocated on the heap are subject to the same boundary checking issues as those located on the stack. Heap memory has two types of chunks, allocated chunk and free chunk. Each chunk has some control metadata, such as head, front pointer, and back pointer. Heap-based buffer overflow usually happens when an application writes extra data to a heap chunk buffer, which leads to corruption of the control sections of the chunk. This leads the memory allocator to an undefined state, and potentially, crashing the application or even executing arbitrary code.

In fact, `Buffer Errors` in Android do not differ much from other platforms as vulnerability characteristics. What is distinct is that static analysis tools that support other native projects might not be applied well in Android. As `Buffer Errors` in Android apps could involve communication from Java to native code through JNI. Yet, most current static analysis tools focus on one language per analysis, thus they would not contain the whole picture when analyzing a portion of the program. Another issue is that Android apps do not have a main method, but they compromise multiple entry points which could involve different call graphs than traditional apps.

IV. Case Study

A. Studied Buffer Error Vulnerabilities

In our case study we want to test state-of-the-art static analysis tools against `Buffer Error` vulnerabilities in Android apps. While we could write our own toy Android apps and inject `Buffer Error` vulnerabilities, we feel that it would be a biased experiment. Hence, we mined the vulnerability records from NVD to identify example vulnerabilities in open source Android apps that we can use as case study subjects. We found 9 `Buffer Error` vulnerability records in 3 open source Android apps (Google Chrome, Android browser, and Mozilla Firefox). Thus by examining the static analysis tools against existing popular Android apps with real word vulnerabilities makes our experiments, results and conclusions stronger. In this section, we describe each of the vulnerabilities categorized within corresponding reason in Table II and follow up with a discussion on some of the common attributes among all of them.

1) Buffer Size Miscalculation:

- **CVE-2008-0985**
  A remote attacker could cause a heap-based buffer overflow by persuading a victim to visit a malicious web site that

[1]In current Android versions, Java code is managed and executed by Android runtime (ART). As all we demonstrate here applies to both DVM and ART, we will only refer to the DVM in the rest of the paper for simplicity.

CVE ID	Reason	C/C++	Affected app	Source	Sink (Container)	Data flow
CVE-2008-0985	Buffer size miscalculation	C++	Android Browser	User (Gif image)	Class	Inter-procedural
CVE-2017-5014	Buffer size miscalculation	C++	Google Chrome	User (Image)	uint32_t pointer	Inter-procedural
CVE-2016-5182	Lack of boundary checking	C++	Google Chrome	User (Bitmap image)	Smart pointer	Inter-procedural
CVE-2014-1705	Lack of boundary checking	C++	Google Chrome	User (JS code manipulation)	Function template	Inter-procedural
CVE-2014-3201	Lack of boundary checking	C++	Google Chrome	User (Scroll size)	Smart pointer class template	Inter-procedural
CVE-2014-1710	Lack of boundary checking	C++	Google Chrome	User (GPU Command Buffer)	Class	Inter-procedural
CVE-2012-4190	Null pointer dereference	C	Mozilla Firefox	Same app	int pointer	Inter-procedural
CVE-2016-5200	Incorrectly applied type rules	C++	Google Chrome	User (JS code manipulation)	Class	Inter-procedural
CVE-2016-5199	Off by one error	C	Google Chrome	User (video file)	Struct	Inter-procedural

TABLE II: Studied Buffer Error Vulnerabilities in Android apps

contains GIF components. The vulnerability occurs at the GIF library in the WebKit framework in Android web browser. It fails to properly sanitize input which is a .gif file before copying it to an inadequately sized memory buffer. The problem occurs due to allocating buffer size based on the logical screen width and height field of the GIF header. However, the buffer is filled in with bytes based on the real width and height of the GIF image.

- **CVE-2017-5014**
  A remote attacker could cause a heap-based buffer overflow in Google Chrom by persuading a victim to visit a malicious web site that contains crafted image components. The overflow happens during image processing in Skia that miscalculates the buffer size of the image.

2) Lack of Boundary Checking:

- **CVE-2016-5182**
  A remote attacker could cause a heap-based buffer overflow by persuading a victim to visit a malicious web site that include a crafted bitmap. The Google Chrome rendering engine Blink fails to render that particular size causing `Buffer Error`.
- **CVE-2014-3201**
  A remote attacker could cause a buffer overflow by persuading a victim to visit a malicious web site that embeds another document using iframe. The embedded page specifies large dimensions for ::webkit-scrollbar and embeds an image with ::-webkit-scrollbar-corner. The Google Chrome rendering engine Blink fails to render that particular size causing `Buffer Error`.
- **CVE-2014-1705**
  A heap-based buffer overflow vulnerability was found in the Google V8 JavaScript engine, which is an open source JavaScript engine written in C++. It exists within handling of TypedArray objects. The vulnerability occurs due to missing bounds checking for the length of ArrayBuffer when manipulated using js defineGetter method, which is then fed to the TypedArray object during initializing. This may allow an attacker to read and write data to any memory address which could be leveraged to arbitrary code execution in the Google Chrome sandbox process.
- **CVE-2014-1710**
  In this vulnerability, Google Chrome does not check boundary of a certain location is within a shared-memory segment. This allows remote attackers to cause GPU command-buffer memory corruption and a denial of service. The GPU command-buffer is the way in which Chrome communicates to the GPU either OpenGL or OpenGL ES, which are APIs for rendering 2D and 3D vector graphics. User interaction is required to exploit this vulnerability and cause GPU process to crash; in such that the user should open a carefully malicious a crafted page that has scripts to dynamically modify the structure of the web page after load time.

3) NULL Pointer Dereferences:

- **CVE-2012-4190**
  This vulnerability was reported by a user that was complaining about Mozilla Firefox app crashing in Android in CyanogenMod kernel. The developers took a month to figure out the exact problem. The issue was triggered because of the Cairo library, written in C, which was calling the FreeType library from the system path instead of calling it from in-tree causing memory corruption. So when FreeType library is initially created, it has non-NULL module pointers. However, at some later point, one of the pointers has become NULL. The vulnerability was patched by forcing Cairo library to use Mozilla in-tree setlcdfilter of FreeType. Thus, calling a system function instead of using local function led to NULL pointer dereference.

4) Incorrectly Applied Rules:

- **CVE-2016-5200**
  A heap-based buffer overflow vulnerability was found in Google V8 JavaScript engine, that allowed a remote attacker to it by persuading a victim to visit a crafted HTML page. The Typer in V8 incorrectly applied type rules when using asm optimizer that could cause an out of bound read/write.

  5) Off by One Error:
- **CVE-2016-5199**
  A remote attacker could cause a heap-based buffer via a crafted video file. The vulnerability occurs in Google Chrome due to an off by one error that leads to an allocation of zero size in FFmpeg MP4 decoder causing corrupting a number of pointers.

B. Common Attributes of Buffer Errors in Android Apps

It was found that all the studied `Buffer Error` vulnerabilities are in client-side apps, such as web browsers. Also, 7 out of 9 of the studied `Buffer Error` vulnerabilities are C++ based and they have some common characteristics. For instance, input is read from untrusted sources, untrusted input is inadequately validated, or lack of boundary checking. In addition, most of the studied vulnerabilities involve pointer indirection. Also, `Buffer Errors` in our case study are inter-file/inter-procedural, as the buffer is allocated in one function in one file, and overflow in another function in a different file. In addition, it was found that these vulnerabilities occur through common attack surfaces in web browser, such as JavaScript, Hypertext Transfer Protocol (HTTP), Hypertext Markup Language (HTML), Document Object Model (DOM), and Cascading Style Sheets (CSS).

C. Examined Static Analysis Methods

Static analysis techniques examine the program statically without executing it. They can analyze either the source or binary code of the program. In our study, we only focus on methods that support source code analysis. As `Buffer Errors` is a well-known problem, multiple static analysis methods and tools have been already proposed. We investigated 17 popular static analysis tools to detect `Buffer Errors`. We gathered some techniques by reviewing research work that evaluated methods that detect `Buffer Errors` [32] [33] [34] [35] [36]. Also, we studied some other static analysis tools and methods that are popular in the wild, such as Clang Static Analyzer [37] and Frama-C [38]. When we studied and collected tools and methods that discover `Buffer Error`, we found that none of these tools was implemented specifically for Android. However, they are general tools for multiple platforms.

We classified the studied static analysis methods based on classification by Shahriar and Zulkernine [35]. Shahriar and Zulkernine [35] classified static analysis methods that target `Buffer Errors` based on several features, such as inference algorithm, analysis sensitivity, analysis granularity, and target language [35]. The underlying inference algorithm refers to how does the static analysis method infer potential vulnerabilities methodically by analyzing program code. Inference methods are categorized into four types: string pattern matching, tainted data flow, constraint, and annotation. String pattern matching method is one of the simplest methods of static analysis tools (e.g., RATS, and Flawfinder). This technique tokenizes program source code to identify a well-known set of tokens or library function calls that could cause a `Buffer Error`. The other three types; tainted data flow, constraint, and annotation involve more advanced analysis to understand the semantic of the source code [35]. Analysis sensitivity indicates how the static method uses pre-computed information based on program code before running the inference algorithm. On the other hand, analysis granularity refers to the granularity level of program code at which an inference is carried out. Table III shows the studied `Buffer Error` static analysis methods classified based on [35]. Also, we categorized the static analysis methods into two categories: Open source and Commercial.

In this study, we want to determine which of the state-of-the-art static analysis techniques could potentially detect the nine Android `Buffer Error` vulnerabilities described in Table II. As our study revealed that most of Android `Buffer Errors` occurred in C++ language, thus all static analysis techniques that target C language could not detect these kind of vulnerabilities. So we excluded 5 out of 17 static analysis techniques that target C language, such as Splint, BOON, ARCHER, and UNO. We found that there are 12 tools could analyze C++ and could potentially detect studied vulnerabilities.

Six out of the 12 static analysis techniques are commercial tools that we did not test (We tested CodeSonar, see Section VI for more details). We are in the lengthy process of acquiring an academic license for Parasoft C/C++test, Polyspace Bug Finder, Coverity, and Klocwork. However, in this paper we stick to the six static analysis tools, which are free and open source. The reasoning is that even though we as academics can get some of the commercial tools for a lower price, such pricing schemes are not available for an app developer. The actual cost of such tools can be up to thousands of U.S. dollars, while the average cost of building an app (for a small development firm of 2-3 people, which is a non-trivial population in the app market), is about 6,000 - 200,000 USD. Therefore, using such expensive commercial static analysis tools may not be viable in the budget for all mobile app developers. Therefore, in this paper we only evaluate the six free and open source static analysis tools.

V. RQ: Are state-of-the-art static analysis tools for Buffer Errors able to detect vulnerabilities reported in the wild for Android apps?

Static analysis tools are a typically adopted solution by developers to keep project costs down. In this section, we evaluate the efficiency of the six free and open source static analysis tools.

Method Name	Type	Language	Inference Algorithm	Sensitivity	Granularity	Could Find Buffer Errors in our Case Study?
Polyspace Bug Finder [32] [38] [39]	Commercial	C/C++	Constraint: abstract interpretation	Flow	Inter-procedural	Yes
Parasoft C/C++test	Commercial	C/C++	String pattern matching, Constraint: symbolic execution	Flow	Inter-procedural	Yes
Klocwork Insight [39]	Commercial	C/C++	Unpublished	Flow, path	Inter-procedural	Yes
Coverity [39]	Commercial	C/C++	Unpublished	Flow, path, context	Inter-procedural	Yes
PVS-Studio [38]	Commercial	C/C++	Constraint: symbolic execution, annotations	Flow, path, value range	Inter-procedural	Yes
CodeSonar [38]	Commercial	C/C++	Constraint: symbolic execution, taint data flow	Flow, path	Inter-procedural	Yes
ASTREE [33][38]	Commercial	C	Constraint: abstract interpretation	Context	Inter-procedural	No
ARCHER [32]	Open source	C	Constraint: symbolic execution	Flow, path, context, alias	Inter-procedural	No
BOON [32][33]	Open source	C	Constraint: integer range	N/A	Inter-procedural	No
Splint [32][34][33]	Open source	C	Annotation	Flow	Intra-procedural, inter-procedural	No
UNO [32][38][34]	Open source	C	Annotations	Flow, path	Inter-procedural	No
Flawfinder [34][33]	Open source	C/C++	String pattern matching	N/A	Token	Yes
RATS [34][33]	Open source	C/C++	String pattern matching	N/A	Token	Yes
Cppcheck [34][33]	Open source	C/C++	Constraint: integer range	Flow, context	Inter-procedural	Yes
Clang Static Analyzer [37]	Open source	C/C++	Constraint: symbolic execution, annotation	Flow, path	Inter-procedural	Yes
Farma-C [38]	Open source	C/C++	Constraint: abstract interpretation, annotations	Flow, value range, point-to	SDG	Yes
IKOS [40]	Open source	C/C++	Constraint: abstract interpretation	Flow, path, point-to	Inter-procedural	Yes

TABLE III: State-of-the-art Static Analysis Tools to Detect Buffer Errors

A. Methodology

1) Collecting open source static analysis tools that detect `Buffer Error`:

We gathered the six open source static analysis tools that discover `Buffer Error`. Our study tests the following tools that support C++: IKOS [40], Frama-C [41], Clang Static Analyzer [37], Cppcheck [34], Flawfinder [34], and RATS [34]. Table IV summarizes all tested tools and their versions.

Tool Name	Version Number
RATS	2.4
Flawfinder	1.31
Cppcheck	1.72
Clang Static Analyzer	279.1
IKOS	1.2
Frama-C	Aluminium-20160502

TABLE IV: The Studied Free and Open Source Static Analysis Tools That Detect Buffer Errors in C++

2) Collecting the source code of the vulnerable apps: We then gathered the source code of open source Android apps that have `Buffer Error` vulnerabilities in our dataset.

The following source code versions of Android apps were collected and checked:

- CVE-2008-0985: Android web browser's webkit rendering engine webkit-522-android-m3-rc20
- CVE-2012-4190: Mozilla Firefox web browser 16.0
- CVE-2014-1705: Google Chrome web browser 33.0.1750.165 V8 JavaScript engine
- CVE-2014-1710: Google Chrome web browser 33.0.1750.15
- CVE-2014-3201: Google Chrome web browser 37.0.2062.94 Blink rendering engine.
- CVE-2016-5182: Google Chrome web browser 54.0.2840.85 Blink rendering engine.
- CVE-2016-5199: Google Chrome web browser 55.0.2883.83
- CVE-2016-5200: Google Chrome web browser 55.0.2883.83 V8 JavaScript engine
- CVE-2017-5014: Google Chrome web browser 56.0.2924.86

3) Running static analysis tools against the source code of the vulnerable apps: All vulnerable source code of each app was tested through the static analyzers. Each tool was run several times using different options and flags. Then, automated scripts have been built to run tests automatically and save results to files.

4) Analyzing the results: in this step, we analyzed the results that were stored to files. Since most of the tools could report errors with the source code file path, error type, and error line number, we opened the source code files and traced and analyzed all reported errors. All collected source code, scripts, and results could be found here [2]

B. Results

It was found that Flawfinder, RATS, and Cppcheck static analysis tools are easy to use. They could be executed through a command line interface, and accepting a list of files or projects directories to test with a set of options as parameters. Also, these tools show their results by default in the

[2]To be added after double blind revision

system's command line. The results contain the files path, the lines of code that are suspected to have vulnerabilities, and descriptions of the potential issues.

Clang Static Analyzer needs to be integrated into the build process. Frama-C and IKOS were built on top of LLVM/Clang, they accept source files and they are not meant to be integrated into a build process. To analyze source files with Frama-C, source files in our case study need to be preprocessed first, then preprocessed C++ files could be fed to Frama-C using Frama-Clang plugin. IKOS was developed by NASA to analyze flight systems, and it could convert C++ source files to LLVM bitcode first which could be analyzed by the tool then. Similar to Frama-C the files need a lot of preprocessing settings when analyzing source files in complex apps. However, setting preprocessing for source files manually is hard to be achieved with large and complex apps such as Google Chrome and Mozilla Firefox. In addition, running the analysis on large apps with IKOS and Frama-C requires a lot of time as these two tools are based on abstract interruption. Which make sense that these two tools were designed to target individual source files, but not to be integrated into a build system. Table V shows the results of our analysis. In Table V we use the 'X' symbol to indicate when a tool cannot determine the specific vulnerability and a '✓' when it can. As we can see from Table V that none of the tools were able to detect any of the studied vulnerabilities.

C. Discussion

In this subsection we want to discuss possible reasons for why the studied free and open source static analysis tools could not detect the nine vulnerabilities. In order to do that we look at characteristics of the techniques underlying the tools (in the context of `Buffer Error` vulnerabilities). In total we present six such characteristics (the number in the enumeration below corresponds to the superscript in Table V):

CVE ID	RATS	Flaw finder	Cpp check	Clang Static Analyzer	IKOS	Frama-C
CVE-2008-0985	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2012-4190	$X^{2,3}$	$X^{2,3}$	$X^{2,4}$	$X^{2,5}$	X^{6}	X^{6}
CVE-2014-1705	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2014-1710	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2014-3201	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2016-5182	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2016-5199	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2016-5200	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$
CVE-2017-5014	$X^{1,2,3}$	$X^{1,2,3}$	$X^{1,2,4}$	$X^{1,2,5}$	$X^{1,6}$	$X^{1,6}$

TABLE V: Tested Static Analysis Tools Results. X means that the a specific tool was unable to detect the corresponding vulnerability. The superscript numbers indicates the reason in the enumerated list below which explains why the tool could not find the vulnerability.

1) Unable to examine tainted data from code written in another language: In our case study, almost always the tainted data (data from an untrusted source) comes from Java (except CVE-2012-4190), while the static analysis tools only examine the C++ native code. None of the six tools examined are able to determine tainted data that comes from code written in Java.
2) Unable to keep track of pointer operations: The tool is unable to keep track of data when a buffer is manipulated using non trivial pointer operations (CVE-2017-5014), or when a null-pointer is dereferenced (CVE-2012-4190). Almost all of the studied nine vulnerabilities involves some sort of pointer operation.
3) Simple lexical analysis only instead of semantic analysis: RATS and Flawfinder only perform simple pattern matching, which is basically tokenizing the source code and looking for tokens that are well-known to be involved in Common Weaknesses Enumeration list (CWE). They mostly focus on tokens that could potentially be a source or sink to buffer overflow such as dangerous functions like the **strcpy** function. However, the sink has more compound settings than the well-known set of tokens. For example, in CVE-2008-0985, the buffer is copied to the sink inside a loop that involves pointer indirection. Although, string pattern matching could be a powerful technique that could be enhanced to catch this kind of complicated sink by combining high level semantic analysis methods, it should not be used solely. In general, Flawfinder and RATS, have high false positive rate, since they always report vulnerabilities based on simple lexical analysis without semantically analyzing the code.
4) Inadequate semantic analysis: Cppcheck is an example of a static analysis tool that uses inadequate semantic analysis. Unlike RATS or Flawfinder, it uses semantic analysis based on Abstract Syntax Trees (AST) and it also utilizes control-flow analysis. However, it does not always perform control flow analysis on all situations and sometimes it assumes that all statements are reachable. Having inadequate semantic analysis may lead to miss problems related to buffer allocation and validation or pointer dereferencing. For the CVE-2012-4190 vulnerability, Cppcheck discovered two NULL pointer dereference issues as they are passed as parameter and used without checking if they are NULL (we checked and found that they are true positives, but not reported to NVD). However, it did not discover the NULL pointer dereference that was discovered in CVE-2012-4190 and reported to NVD.
5) Limited scope of analysis: Clang Static Analyzer performs a sophisticated high level AST analysis. However, since it uses the AST generated by Clang, the analysis is performed at the compilation unit level. That is, it uses inter-procedural analysis, however it does not support inter-procedural analysis for cross-translation-unit. In fact, almost all of the vulnerabilities in our case study are inter-file/inter-procedural.
6) Tied to a specific compiler: IKOS and Frama-C uses a special compiler based on Clang/LLVM compiler framework that supports both C and C++. Therefore, they cannot understand the C/C++ code written for the Android platform. These tools, however, perform sophisticated

semantic, data flow, control flow, pointer operation, and inter-procedural analysis in a robust fashion. However, because they are unable to analyze the native code in Android apps, they are unable to accurately determine the `Buffer Errors` in our case study.

From the reasoning above we can see that an ideal static analysis tool to detect `Buffer Errors` in Android apps will:

- performs inter-language analysis to be able to understand both Java and native code contexts.
- utilizes tainted data flow as inference algorithm.
- employs sophisticated semantic, data flow, control flow and pointer operation analysis
- employs Inter-procedural analysis

VI. Threats to Validity

One possible threat to external validity is that we did not test many commercial tools that use their own parser which might be more accurate. Though, we tried to analyze studied vulnerabilities using CodeSonar. We got an academic license of CodeSonar, which only allows to analyze one million lines of code. However, the analyzed apps in our study includes millions lines of codes.

In addition, knowing that most of the studied vulnerabilities involve pointer indirection, we know from past research that it could be hard to discover them by most of the commercial tools [42]. For instance, the Heartbleed is a pointer indirection vulnerability in OpenSSL [43] that was not discovered by commercial tools, such as Coverity Code Advisor, Code Sonar, Klocwork, and Veracode [42]. Commercial tools were not able to find Heartbleed as they are unsound. In fact, Heartbleed vulnerability was hard to be found due to using multiple level of pointer indirection and the complexity of the execution path from the buffer allocation to buffer misuse which is similar to our case. Additionally, we qualify that our experiments are done only on free and open source tools (the reason for which is given in Section IV-C), by clearly stating "free and open source" in the title, abstract and conclusion to avoid any misunderstanding.

There is a threat that the vulnerability records we analyzed may not be related to Android. Hence. we manually inspected and studied all collected records from NVD to ensure that they related to the Android ecosystem and removed all biased records. Also, we manually reviewed static analysis tool results to guarantee the correctness. In order to address any threat to internal validity (i.e., mistakes we could have made) and for the ability for anyone to replicate our experiments, we provide all the data in our experiments. [3]

VII. Related Work

A. Android Vulnerabilities Analysis

To support the development of vulnerability detection models, several studies have been conducted to understand the patterns of the vulnerability in Android software. Huang et al. [44] studied the mobile vulnerability market to reveal the unique vulnerability patterns of mobile software. The study shows that the vulnerabilities in the Android market are more exploitable than in the entire market; and the exploitation impact is higher based on CVSS metrics. Our research took further steps to reveal the pattern of these vulnerabilities in Android apps.

B. Android Static Analysis Tools

Although, extensive work has been conducted to introduce static analysis tools for Android vulnerabilities and malicious behavior, most of the studies focus on permissions and information leakage issues and do not address `Buffer Error` [22]. For instance, the SCanDroid tool [45] performs a data-flow analysis of installed Android apps, to track inter-component communication through intents in order to detect the potential violation of permission through a coalition of applications. Similarly, Quire [46] tracks permissions through the IPC call to prevent privilege attacks among applications. In addition, number of tools have been developed to detect private data leakage. TaintDroid [47] preforms dynamic taint analysis, while FLOWDROID [48], LeakMiner [49], and AndroidLeaks [50] are static analysis approaches to detect data leakage. VulHunter [51] is a static analysis framework to support vulnerability detection for Android apps by extracting information from applications. It aims to detect five types of vulnerabilities that are related to information leakage and permissions violations. In fact, there is a lack of research on static analysis tools that target native code in Android apps. Such mapping between the native and Java contexts will improve the discovery of Android vulnerabilities by constructing a better call graph. The issue of the lack of static analysis methods that link Java code to native code has been recently highlighted [52] [53]. Our research looks at `Buffer Error` vulnerabilities. The above tools do not address `Buffer Error` vulnerabilities, and we showed that no open source static analysis tools could statically determine `Buffer Error` vulnerabilities in Android apps.

C. Dynamic Analysis for Buffer Errors in Android

`Buffer Error` is a well-known problem that still resurfaces. Several run-time approaches have been proposed to mitigate `Buffer Error` vulnerabilities in Android. Hardware-based No eXecute (NX) technique has been added to Android 2.3 Gingerbread to prevent code execution on the stack and heap. Another approach is using compiler extensions, such as ProPolice, which is improved over StackGuard [54], to protect against stack-based buffer overflow. ProPolice has been introduced in Android 1.5 CupCake [55]. In Android Jelly Beans version 4.2 and later, a new feature has been added that protects against `Buffer Errors`, as all applications and system libraries during compile time are checked with FORTIFY_SOURCE feature [55], which detects and stops a certain types of `Buffer Errors`. Defense side obfuscation is another approach to mitigate `Buffer Errors`. An example of this approach is Address Space Layout Randomization

[3] To be added after double blind reviews.

(ASLR) that randomizes memory addresses of stack and heap each time the memory is allocated for a process, such that finding executable code becomes unreliable. ASLR has been introduced in Android 4.0 Ice Cream Sandwich [55]. However, overcoming the above mentioned techniques is possible and this was discussed in [56] [57]. Real exploitation and code execution have already been proven in the wild [58]. *In fact, the dynamic approaches that prevent* `Buffer Error` *attacks do not eliminate the vulnerabilities from the source code.* While we studied static analysis tools that could potentially identify the issue earlier. One advantage of static analysis is that security vulnerabilities can be removed before code is deployed, which reduces the cost of the risk.

D. Static Analysis for Buffer Errors in C/C++

Several static analysis prototypes have been introduced in academia to detect `Buffer Errors`, yet the majority of the proposed methods target C programming language. For instance, Mjolnir [59], MACKE [60], MESCs [61], Wagner et al. [62], Kim et al. [63], Hackett et al. [64],CSSV [35], Vulncheck [65] [35], Avots et al. [66], and Livshits and Lam [67] are all academic prototype that only focus in C. However, it is noticed that little attention has been directed lately to study and propose methods to target C++ which could be able to address the OOP model, such as Marple [68] and Li et al. [69].

E. Inter-language Static Analysis

In fact, studying inter-language analysis techniques has been investigated in the past. Su and Wassermann [70] introduced an algorithm for checking type safety across a foreign function interface, such as between OCaml and C. The system prevents foreign function calls in C from introducing type and memory bugs into an safe language such as OCaml. Siefers et al. [71] presented static analysis techniques to detect bugs in programs using JNI. Their analysis detects bugs such as exception handling, memory leaks, and invalid local references. Li and Tan [72] proposed a static analysis framework to examine exceptions and report errors in JNI programs. Their framework can be applied to other foreign function interfaces, including the Python/C interface and the OCaml/C interface. Tan and Croft [73] have conducted an empirical security study on the native code portion of Suns JDK 1.6. They used ITS4, Flawfinder, and Splint to carry out the analyses. It was mentioned the importance of building inter-language analysis across Java and C, as most existing tools are limited to code written in a single language. However, our study focuses on `Buffer Errors` particularly which have not been considered before in such tools. Also, our study suggests that unlike Java, when building a tool for Android apps, the multiple entry points should be considered as well.

F. Evaluation of Static Analysis Tools for Buffer Errors

Although, a few studies have evaluated the use of static analysis tools for `Buffer Error` detection [32] [33] [34] [74], to the best of our knowledge, this is the first empirical study that evaluated these tools against known `Buffer Error` in the Android apps domain. Our results expose the need for more advanced static analysis tools to detect `Buffer Errors` in Android apps taking into consideration Android app nature.

VIII. Conclusions

Our study found that `Buffer Errors` were the most frequent type of vulnerability occurring in Android apps, and they are easy to exploit. Therefore, we decided to study the effectiveness of state-of-the-art static analysis tools for detecting `Buffer Error` vulnerabilities in Android apps. Our findings indicate that there is a lack of free and open source static analysis tools that target Android, particularly to detect `Buffer Errors`. To the best of our knowledge, there is no free and open source static analysis tool for Android that analyzes both native code (which may introduce `Buffer Errors`), and Java code (which may originate the unsanitized input). Thus, general static analysis tools for C++ are usually used by Android developers. However, current free and open source static analysis tools for C++ could not detect all `Buffer Errors` in Android.

Also, our study found some pattern of characteristics for `Buffer Errors` that occurred in Android apps, such as occur within C++, due to pointers indirection, untrusted input travels from Java to C++ where buffer is misused. Thus, by utilizing these patterns, static analysis tools could be built to work more efficiently. The experimental results show that such an evaluation brings an important contribution characterizing an effective static analysis tool to detect `Buffer Errors` in Android apps. Therefore, we conclude that an efficient static analysis technique for detecting `Buffer Errors` in Android (1) should perform a taint analysis that traces inputs to a program from outside, (2) should involve inter-language analysis (this could effect other modern apps in other platforms that include native C++ as library), (3) has better understanding of the code semantics and involve pointer operation analysis (this is a general problem that effect apps in all kinds of platforms.), and (4) should perform inter-procedural analysis to analyze data travel cross procedures.

REFERENCES

[1] "Smartphone os market share, 2015 q2." [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] Statista.com, "Number of available applications in the google play store from december 2009 to september 2016," 2016. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[3] ——, "Cumulative number of apps downloaded from the google play as of may 2016 (in billions)," 2016. [Online]. Available: https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/

[4] Kantar Media, "Apps Environment: Research Report," *Ofcom*, 2014. [Online]. Available: http://stakeholders.ofcom.org.uk/binaries/research/telecoms-research/Apps_Environment.pdf

[5] N. J. Percoco and S. Schulte, "Adventures in BouncerLand," *Proc. of Black Hat USA*, 2012.

[6] "2015 Internet Security Threat Report," *Internet Security Threat Report*, vol. 20, no. April, p. 119, 2015. [Online]. Available: https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf

[7] "Vulndb quickview 2015 vulnerability trends." [Online]. Available: https://www.riskbasedsecurity.com/vulndb-quickview-2015-vulnerability-trends/

[8] Y. Younan, "25 Years of Vulnerabilities: 1988-2012," 2012. [Online]. Available: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf

[9] D. Childs, A. Gilliland, B. Gorenc, H. Goudey, A. Gunn, A. Hoole, and J. Lancaster, "HPE Security Research - Cyber Risk Report 2016," p. 76, 2016.

[10] D. Research, "Mobile app usage and abandonment survey," Jan 2015.

[11] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 580–586.

[12] M. Soni, "Defect prevention: Reducing costs and enhancing quality," *iSixSigma. com*, vol. 19, 2006.

[13] "National vulnerability database (nvd)." [Online]. Available: https://nvd.nist.gov/

[14] B. Aloraini and M. Nagappan, "Evaluating free and open source static analysis techniques against buffer errors in android apps," 2017. [Online]. Available: https://github.com/BushraAloraini/Android-Vulnerabilities

[15] "Osvdb." [Online]. Available: https://blog.osvdb.org/

[16] "Cert." [Online]. Available: http://www.cert.org/

[17] "Securityfocus." [Online]. Available: http://www.securityfocus.com/

[18] "the common vulnerabilities and exposures identifier (cve)." [Online]. Available: https://cve.mitre.org/

[19] "Japan vulnerability note." [Online]. Available: https://jvn.jp/en/

[20] W. Dormann, "Finding android ssl vulnerabilities with cert tapioca," Sept 2014. [Online]. Available: https://insights.sei.cmu.edu/cert/2014/09/-finding-android-ssl-vulnerabilities-with-cert-tapioca.html

[21] "The common weakness enumeration(cwe)." [Online]. Available: https://cwe.mitre.org/

[22] A. Sadeghi, H. Bagheri, J. Garcia, and s. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016.

[23] "The common vulnerability scoring system(cvss)." [Online]. Available: https://www.first.org/cvss/v2/guide

[24] Android, "Application fundamentals," 2016. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html

[25] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets." in *NDSS*, vol. 25, no. 4, 2012, pp. 50–52.

[26] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 180–191.

[27] S. Liang, *The Java Native Interface Programmers Guide and Specification*. Reading, Massachusetts: Addison Wesley Longman, Inc., 1999, vol. 56.

[28] O. Cinar and G. Allen, *Pro Android C++ with the NDK*. Springer, 2012.

[29] G. Sims, "Arm vs x86 key differences explained!" NOVEMBER 2014. [Online]. Available: http://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/

[30] E. Naughton, "The bionic library: Did google work around the gpl?" no. March 2011, pp. 1–10, 2011.

[31] ARM, "Procedure Call Standard for the ARM Architecture," pp. 1–33, 2015.

[32] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, p. 97, 2004.

[33] T. Hofer, "Evaluating Static Source Code Analysis Tools," Master's thesis, School of Computer and Communications Science, Switzerland, 2010.

[34] L. Torri, G. Fachini, L. Steinfeld, V. Camara, L. Carro, É. Cota, P. Box, and P. Alegre, "An Evaluation of Free / Open Source Static Analysis Tools Applied to Embedded Software," 2010.

[35] H. Shahriar and M. Zulkernine, "Classification of static analysis-based buffer overflow detectors," in *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, June 2010, pp. 94–101.

[36] K. Vorobyov and P. Krishna, "Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches," *Proc. SSV*, pp. 1–7, 2010.

[37] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "Begbunch: Benchmarking for c bug detection tools," in *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, ser. DEFECTS '09. New York, NY, USA: ACM, 2009, pp. 16–20.

[38] "Static source code analysis tools for c," 2014. [Online]. Available: https://spinroot.com/static/

[39] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, no. C, pp. 5–21, 2008.

[40] G. Brat, J. A. Navas, N. Shi, and A. Venet, "IKOS : A Framework for Static Analysis based on Abstract Interpretation (Tool Paper)," pp. 271–277, 2014.

[41] "Frama-c." [Online]. Available: http://frama-c.com/index.html

[42] J. A. Kupsch, , and B. P. Miller, "Why do software assurance tools have problems finding bugs like heartbleed?" 2014.

[43] "The heartbleed bug." [Online]. Available: http://heartbleed.com/

[44] K. Huang, J. Zhang, W. Tan, and Z. Feng, "An Empirical Analysis of Contemporary Android Mobile Vulnerability Market," 2015, pp. 182–189.

[45] A. P. Fuchs, A. Chaudhuri, and J. Foster, "SCanDroid : Automated Security Certification of Android Applications," *Technical report, University of Maryland*, vol. 10, no. November, p. 328, 2009.

[46] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: lightweight provenance for smart phone operating systems," vol. 271, no. 2012, 2011, p. 23.

[47] "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[48] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269.

[49] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*, Nov 2012, pp. 101–104.

[50] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 291–307.

[51] C. Qian, X. Luo, Y. Le, H. K. Polytechnic, and G. Gu, "Vulhunter: Toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan 2015.

[52] P. Lantz and B. Johansson, "Towards bridging the gap between dalvik bytecode and native code during static analysis of android applica-

tions," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2015 International*, Aug 2015, pp. 587–593.

[53] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupe, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[54] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, January 1998, p. 16.

[55] "Security enhancements in android 1.5 through 4.1." [Online]. Available: https://source.android.com/security/enhancements/enhancements41.html

[56] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android hacker's handbook*. John Wiley & Sons, 2014.

[57] H. Marco-Gisbert and I. Ripoll, "On the effectiveness of NX, SSP, RenewSSP and ASLR against stack buffer overflows ," in *IEEE 13th International Symposium on Network Computing and Applications*, 2014.

[58] G. Gong, "Exploiting Heap Corruption due to Integer Overflow in Android libcutils," in *blackhat 2015*, 2015.

[59] M. Weber, V. Shah, and C. Ren, "A case study in detecting software security vulnerabilities using constraint optimization," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 1–11.

[60] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "MACKE Compositional Analysis of Low - Level Vulnerabilities with Symbolic Execution," *Ase*, pp. 780–785, 2016.

[61] R. Gjomemo, P. H. Phung, E. Ballou, K. S. Namjoshi, V. N. Venkatakrishnan, and L. Zuck, "Leveraging Static Analysis Tools for Improving Usability of Memory Error Sanitization Compilers," 2016.

[62] D. Wagner, J. S. Foster, E. a. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *The 2000 Network and Distributed Systems Security Conference*, 2000.

[63] Y. Kim, J. Lee, H. Han, and K. M. Choe, "Filtering false alarms of buffer overflow analysis using SMT solvers," *Information and Software Technology*, vol. 52, no. 2, pp. 210–219, 2010.

[64] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," *Proceeding of the 28th international conference on Software engineering - ICSE '06*, p. 232, 2006.

[65] A. I. Sotirov, "Automatic vulnerability detection using static source code analysis," Ph.D. dissertation, Citeseer, 2005.

[66] D. Avots, M. Dalton, V. Benjamin, and M. Lam, "Improving software security with a C pointer analysis," *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 332–341, 2005.

[67] V. B. Livshits and M. S. Lam, "Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 317–326, 2003.

[68] "Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector," *Analysis*, pp. 272–282, 2008.

[69] L. Li, C. Cifuentes, and N. Keynes, "Practical and Effective Symbolic Analysis for Buffer Overflow Detection," *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 317–326, 2010.

[70] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 372–382.

[71] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the native beast of the jvm," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 201–211.

[72] G. Tan and J. C. An, "empirical security study of the native code in the JDK," *Security*, vol. pages, pp. 365–377, 2008.

[73] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *International Conference on Information Security*. Springer, 2010, pp. 346–360.

[74] D. Pozza, R. Sisto, L. Durante, and A. Valenzano, "Comparing lexical analysis tools for buffer overflow detection in network software," in *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, 2006, pp. 1–7.