Towards Better Static Analysis Security Testing Methodologies

by

Bushra Aloraini

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Karthik Pattabiraman Associate Professor
Supervisor(s)	Meiyappan Nagappan Associate Professor
Internal Members	Michael Godfrey Professor
	Urs Hengartner Associate Professor
Internal-external Member	Mahesh Tripunitara Professor

Author's Declaration

Abstract

Software vulnerabilities have been a significant attack surface used in cyberattacks, which have been escalating recently. Software vulnerabilities have caused substantial damage, and thus there are many techniques to guard against them. Nevertheless, detecting and eliminating software vulnerabilities from the source code is the best and most effective solution in terms of protection and cost. Static Analysis Security Testing (SAST) tools spot vulnerabilities and help programmers to remove the vulnerabilities. The fundamental problem is that modern software continues to evolve and shift, making detecting vulnerabilities more difficult. Hence, this thesis takes a step toward highlighting the features required to be present in the SAST tools to address software vulnerabilities in modern software. The thesis's end goal is to introduce SAST methods and tools to detect the dominant type of software vulnerabilities in modern software. The investigation first focuses on state-of-the-art SAST tools when working with large-scale modern software. The research examines how different state-of-the-art SAST tools react to different types of warnings over time, and measures SAST tools precision of different types of warnings. The study presumption is that the SAST tools' precision can be obtained from studying real-world projects' history and SAST tools that generated warnings over time. The empirical analysis in this study then takes a further step to look at the problem from a different angle, starting at the real-world vulnerabilities detected by individuals and published in well-known vulnerabilities databases. Android application vulnerabilities are used as an example of modern software vulnerabilities. This study aims to measure the recall of SAST tools when they work with modern software vulnerabilities and understand how software vulnerabilities manifest in the real world. We find that buffer errors that belong to the input validation and representation class of vulnerability dominate modern software. Also, we find that studied state-of-the-art SAST tools failed to identify real-world vulnerabilities. To address the issue of detecting vulnerabilities in modern software, we introduce two methodologies. The first methodology is a coarse-grain method that targets helping taint static analysis methods to tackle two aspects of the complexity of modern software. One aspect is that one vulnerability can be scattered across different languages in a single application making the analysis harder to achieve. The second aspect is that the number of sources and sinks is high and increasing over time, which can be hard for taint analysis to cover such a high number of sources and sinks. We implement the proposed methodology in a tool called **So**urce **S**ink (SoS) that filters out the source and sink pairs that do not have feasible paths. Then, another fine-grain methodology focuses on discovering buffer errors that occur in modern software. The method performs taint analysis to examine the reachability between sources and sinks and looks for "validators" that validates the untrusted input. We implemented methodology in a tool called **B**uffer **E**rror **Finder** (BEFinder).

Acknowledgements

I would like to thank all the special people who helped me complete this life-changing experience and made my dream come true. To my knowledgeable supervisor, Dr. Mei Nagappan, I sincerely appreciate the guidance and support you offered me during my Ph.D. study. I am also profoundly thankful for your patience, encouragement, and the freedom you offered to explore research topics that fascinated me.

My deep gratitude also go to my committee chair, Dr. Patrick Mitran; my committee members: Dr. Michael Godfrey, Dr. Urs Hengartner, and Dr. Mahesh Tripunitara; and my external examiner, Dr. Karthik Pattabiraman, for their valuable time, encouragement, and insightful comments that helped me to improve my thesis. I immensely enjoyed your discussion during the defense and your intelligent questions and profound thoughts that provided me inspiration for future research.

I am sincerely grateful for my collaborators: Dr. Daniel M. German, Dr. Shinpei Hayashi, and Dr. Yoshiki Higo, for the collaboration and the feedback they gave me during the research project. It was an absolute pleasure working with such brilliant people who showed me the importance of perspectives.

I am thankful for all the Software Analytics Group members for the help provided and the joyful times we spent together at the University of Waterloo. I will miss our Friday lunches out but I want to take a break from burritos.

To my family, friends, and all the people who have supported me directly or indirectly-thank you all!

Dedication

This work is dedicated to all the free and courageous people.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We are witnessing an era where rapid technological innovations are reshaping our world. Many aspects of our daily life are shifting to be virtual. This paradigm leads to more connectivity among devices than ever before, meaning more sensitive data is shared. Consequently, this shift necessitates addressing cyberspace's security as an urgent challenge, since the security risks become higher, influencing everyone in cyberspace. Cybersecurity is the process of protecting and recovering networks, devices, software, and data from cyberattacks. The majority of cyberattacks, around 90%, are estimated to be a result of exploiting software vulnerabilities [21]. Nevertheless, the number of software vulnerabilities is rapidly rising [164][209][149]. According to Risk Based Security [164], in 2015, the number of vulnerabilities increased by 77% compared to the vulnerabilities reported in 2011. In 2017, the number of vulnerabilities that were assigned Common Vulnerabilities and Exposures Identifier (CVE-ID) was more than double that of 2016 [124][149].

A software vulnerability is a software flaw that may allow unauthorized operations, such as malicious code execution, privileges elevation, information leakage, or denial of service. A single software vulnerability can lead to devastating ramifications. In 2014, the *Heartbleed* vulnerability in the OpenSSL cryptography library occurred because of the missing bounds check of a buffer [29]. This vulnerability allows attackers to remotely steal the secret keys protected by the encryption protocol and potentially impersonate services and users. The cost of this incident is estimated to be US$500 million [114]. In 2015, the *Stagefright* set of buffer error vulnerabilities were exploited in the Stagefright library in Android OS 2.2+ [100]. This Stagefright vulnerability primarily occurred because of an integer overflow that led to a buffer overflow, allowing an attacker to remotely compromise Android phones with OS version 2.2+. The vulnerability is exploited by sending malformed

MMS messages or videos, leading to malicious code execution. This vulnerability impacted nearly one billion Android devices.

Many security solutions have been proposed to defend against cyberattacks due to software vulnerabilities; however, it is well-known that building secure software is the first line of defense. Using Static Application Security Testing (SAST) tools is one way that helps software developers to find software vulnerabilities early during the coding phase and fix them. This leads to cost-saving, which is a crucial benefit of SAST tools [174]. SAST tools statically examine the program code and provide the developer with warnings associated with the flawed lines of code that could potentially lead to vulnerabilities. The SAST tools classify each warning into one of many types and assign severity to them. For each warning, a human code reviewer decides if the warning poses a real defect that can be exploited by attackers (true positive) and, hence, the defect needs to be fixed, or the given warning cannot be exploited by attackers (false positive), and hence no further action is required. Sometimes, some software vulnerabilities exist in the code; however, due to the tool's limitations, no warning regarding those software vulnerabilities is generated (false negative). Another potential outcome of the tool is that no security warnings are reported because the source code is secure (true negative).

In an ideal setting, the SAST tool should be both *sound*, meaning it does not report any unreal problems (no false positives) and *complete*, meaning it does not miss any real problems (no false negatives). This cannot be achieved in practice for the fundamental reason expressed by Rice's theorem [162][163], which implies that the static analysis method must settle for approximation techniques. Usually, the approximation techniques are conservative. However, in practice, the SAST tools should balance between correctness and scalability. For this reason, SAST tools generate false positives and false negatives.

A very fundamental question to ask is that are SAST tools meeting the needs to find vulnerabilities in modern software? Modern software systems are becoming more complicated because of the demands for interconnectivity, integration, and platform compatibility. Since the new software development culture emphasizes component reuse, modern software is likely to depend on third-party components and libraries. Synopsys indicates that 99% of codebases audited in 2019 included open-source components [38]. Hence, a single vulnerability in one library can influence many other software projects that depend on it. This also indicates that software is heterogeneous (i.e., polyglot program), where one software is composed using different languages due to the availability of already written components in a different language. Another issue is that modern software may have a different architecture than traditional software. For instance, Android apps do not have a main method, but they comprise multiple entry points. Moreover, modern software is growing in size over time. The law of software envelopment by Zawinski [160] suggests that "Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can". Likewise, frameworks supporting the programming languages are expanding too and providing increasing APIs over time. Hence, finding vulnerabilities is more difficult in modern software systems [108].

Therefore, this thesis's primary goal is to shed light on vulnerabilities in modern software to build better SAST tools. To accomplish this goal, we acquire knowledge about the most significant type of vulnerability in modern software from the perspective of SAST tools, and the perspective of the individuals who find the vulnerabilities and report them to well-known vulnerability databases. Then, we gained insights about the precision and the recall of the SAST tools. Thus, we performed two empirical studies using real-world cases to obtain an in-depth understanding of state-of-the-art SAST tools' capabilities and spot their weaknesses. The observations that we acquired from the empirical studies point out the significance of one type of vulnerability that belongs to the *Input Validation and Representation* (IVR) class: *buffer errors*. Hence, we conducted further studies to solve this particular problem. Using the knowledge gained from such observation, we introduced two static analysis methods. One method is general for the IVR, and the other is proposed to detect buffer errors in modern software. Most of the studies in this thesis adopt Android apps as a subject to investigate modern software vulnerabilities and how SAST tools operate with modern software. In the next sections, an overview of the studies that have been conducted in this thesis is given.

1.1 Precision of SAST Tools

The goals of the first empirical study are to get a basic understanding of state-of-the-art SAST tools precision when applied to real-life projects, the types of warnings they tend to generate, and the types of underlying inference algorithms that can be effective to produce true warnings. In academia, the precision of SAST tools has been investigated mainly by using benchmarks. In this thesis, we adopted a different methodology. Instead of examining test suites, we adopted running the SAST tools with real projects. We then derived the developers' decisions about the warnings produced by SAST tools from observing the five-year history wherein the developers modified the lines of code associated with the warnings.

We carry out a large-scale empirical study where we examined 116 large and popular C++ projects using six different state-of-the-art open-source and commercial SAST tools that detect security vulnerabilities. To track a piece of code that has been tagged with a warning, we use a new state-of-the-art framework called cregit+ [95] that traces source code lines across different commits. The results demonstrate the potential of using SAST tools as an assessment tool to measure the quality of a product and the possible risks without any manual review of the warnings. Besides, the study shows that SAST tools that use the pattern-matching technique along with more advanced analysis methods detect the Input Validation and Representation (IVR) flaws precisely. This indicates that pattern-matching is a compelling method and is likely to help SAST tools spot real issues when combined with other advanced analysis methods.

1.2 Recall of SAST Tools

The second empirical study is a vulnerability-driven research approach, where we study real-world vulnerabilities in modern software. As an example of modern software vulnerabilities, we focus on real-world Android app vulnerabilities. We choose to focus on Android since it dominates the mobile market and has gained tremendous popularity [110]. The study starts with real-world software vulnerabilities in modern software that are detected by individuals. The study then investigates whether state-of-the-art SAST tools can recall these real-world software vulnerabilities, meaning detect them. This is important since modern software is more complex and more vulnerable than traditional software [187].

The analysis reveals that buffer errors, which belong to the IVR class of vulnerabilities, are the most frequent types of vulnerabilities that threaten Android apps. This is an unexpected finding since Android apps mainly use the Java programming language, and buffer errors are related to the native code (Assembly, C, and C++); however, Android apps can use native code. Afonso et al., [40] observe that 37% of tested apps use native code. Also, Sun and Tan [182] stated that 86% of the most popular Android apps include native code, which signifies the importance of addressing buffer errors in these apps. Our study's findings indicate that none of the state-of-the-art SAST tools were able to detect any of the studied software vulnerabilities. Therefore, we perform a qualitative analysis to understand the nature of the buffer error vulnerabilities by manually analyzing the source code.

We observe some common attributes of buffer errors in Android apps. We find that usually, the input is read from untrusted sources. Also, untrusted input is inadequately validated, or there is a lack of boundary checking. Another attribute we find is that buffer errors in our case study are inter-file/inter-procedural, as the buffer is allocated in one function in one file, and the untrusted input in another function in a different file. The inter-file/inter-procedural communication occurs in a cross-language manner, meaning that buffer errors occur in the native context, while the input comes from another context, such as Java language.

1.3 Filtering Sources and Sinks

The observations gained by the empirical studies indicate that the IVR class of vulnerabilities is a significant type of vulnerabilities that need to be addressed. An IVR vulnerability is a software security problem resulting from trusting input from outside the program. IVR issues include buffer errors, cross-site scripting, command injection, SQL injection, and others. Taint analysis is used to detect such issues to trace input from untrusted sources. Taint analysis usually involves finding a source, which is a method that accepts data from outsiders for IVR type of problem, and a sink, which is a program statement that has an operation that leads to the problem.

However, there are two major challenges that static analysis tools are facing. One challenge highlighted by the empirical studies shows that modern software is likely to have vulnerabilities scattered across languages in one software. Hence, static analysis methodologies need to focus on detecting vulnerabilities, including vulnerabilities involved in cross-languages communications. The second challenge is that as modern software evolves, thousands of source and sink APIs are introduced. Nevertheless, the current taint analysis methods cannot handle many sources and sinks. Thus, we propose a static analysis method to reduce the search space and help SAST tools to perform the taint analysis effectively without missing issues. The proposed analysis method uses a pattern-matching method found to be a powerful static analysis method in the empirical study that we conduct in this thesis.

Moreover, the analysis is based on connecting the call graphs of multiple languages used in a single software. Then, it determines the reachability of the source and sink pairs using the call graph. The goal is to provide a practical solution that provides all possible source and sink pairs reachable in the call graph, providing this to other static analysis tools to perform in-depth analysis. We implement the proposed methodology in a tool called SoS, and the results show a considerable reduction in the feasible source and sink pairs.

1.4 Finding Buffer Errors

Buffer errors that belong to the IVR class are the most common types of vulnerabilities in modern software [44]. In this study, a methodology to detect buffer errors in modern software is proposed. In order to detect buffer errors that are influenced by untrusted input, we need to find a source of the input that can influence a sink that performs read/write on a buffer, and then ensure that the untrusted input was validated/sanitized. The key to the IVR issues is that the validation is missing or not performed correctly. Given the discussion in the conducted empirical study about buffer error patterns in Android apps, validation was always the primary key to the problem. For instance, the boundary checking should ensure that the buffer operation is within the boundary of a buffer, either missing or done wrongly, such as the miscalculated buffer size.

Hence, the proposed model employs taint data-flow analysis based on the Sparse Value Flow Graph (SVFG) [181] that captures data dependencies in the source code. However, since there would be many sources and sinks, we use the output generated by SoS to only focus the analysis on the pairs that may have a feasible path. Once a source and a sink pair are found to be reachable in the SVFG, the analysis would ensure a validation has been performed. The analysis uses a simple constraint solver to evaluate the values against a set of constraints.

1.5 Thesis Statement and Contributions

Thesis Statement. Understanding SAST tool capabilities is a prerequisite to be improving and eventually helping developers find and eliminate software vulnerabilities. This understanding would be more beneficial if it is acquired from empirical research findings. The understanding of the SAST tools problem involves two aspects: one is to recognize the precision of the state-of-the-art SAST tools generated warnings using real-world projects, and the other is to understand how real-world vulnerabilities occur in real-world applications and how SAST tools recall them. This can help to build better tools that address the missing qualities of such SAST tools.

This thesis makes the following contributions:

- We study the state-of-the-art SAST tools generated warnings over time to understand their precision.
- We study real-world vulnerabilities found by individuals that occur in Android apps, and we test the state-of-the-art SAST tools against those vulnerabilities to perceive the recall of the tools.
- We discuss the pattern in which real-world vulnerabilities occur, and we describe the underlying algorithms for a SAST tool to detect real-world vulnerabilities.

- We introduce a general coarse-grain methodology to detect the source and sink pairs, which involve cross-languages communication. Also, we implement a tool called SoS to evaluate the proposed approach.
- We propose a fine-grain methodology to detect buffer errors based on the SVFG and constraint solver. Also, we implement a tool called BEFinder to assess the proposed methodology.

1.6 Organization

This thesis is further organized as follows: Chapter 2 discusses the background information and related work. Chapter 3 introduces an empirical study to understand the precision of state-of-the-art SAST tools. An empirical study to understand the recall of state-of-the-art SAST tools when applied to Android apps is given in Chapter 4. Based on the empirical studies findings in Chapter 3 and Chapter 4, we propose two methodologies in Chapter 5 and Chapter 6. Chapter 5 provides a general methodology to find the source and sink pairs in cross-language to provide only pairs with a feasible path. Chapter 6 discusses a methodology to detect buffer errors based on the SVFG. Chapter 7 summarizes our conclusions for the work presented in this thesis.

Chapter 2

Background and Related Work

This chapter provides background information and a literature review of work related to the topics discussed in the thesis. Since buffer errors are the dominant type of vulnerabilities in modern software, as observed in the empirical studies that we conduct in this thesis, extensive background information about this specific type of vulnerability is given. This type of vulnerability is discussed in the context of Android apps. This way, we show how this type of vulnerability is handled in modern software. We start by explaining how this particular type of vulnerability happens in the source code. Then, how this particular type of vulnerability can be exploited. Different defense mechanisms against buffer errors are given to shed some light on why these mechanisms are not enough to protect against this particular vulnerability; to illustrate how can attackers bypass them, hence eliminate the buffer errors from the source code is the optimal solution. Then, a discussion about the related work and different static analysis methods to detect buffer errors is provided.

2.1 Background Information

2.1.1 Buffer Errors in a Nutshell

Buffer errors continue to be problematic. This type of vulnerability was acknowledged and discussed in part by Anderson in 1972 [50]. In 1988, a stack-based buffer overflow vulnerability in the Unix *finger* daemon was exploited to propagate the first Internet worm, the so-called *Morris* worm, infecting about 10% of the devices connected to the Internet [60]. This is a classic buffer overflow (CWE-120: buffer copy without checking size of input). This occurs when software copies data into a smaller size buffer, resulting in overwriting memory locations.

A buffer consists of chunks of memory that store any type of information in a program. Buffer errors (referred to as CWE-119 in the Common Weaknesses Enumeration category) occur when software reads from or writes to a memory location outside the boundary of the intended buffer. Buffer errors usually happen because programmers assume that data fits into the buffer or improperly perform operations within the bounds of a memory buffer. There are various ways such a problem could happen; for instance, two programming modules read/write from/to a buffer, and due to wrong assumptions, buffer overrun may be triggered. This may result in crashing the program. This may hint an availability issue; however, this is not the type of problem to be solved in this thesis.

In this thesis, only buffer errors that can be exploited and controlled by outsiders are considered. We mean by "controlled" that malicious outsiders can inject tainted inputs, which lead to availability, confidentiality, or integrity problems. The issue occurs because the programmer does not validate the input from untrusted sources. Native programming languages, such as Assembly, C, and C++ (or sometimes, programming languages that are implemented based on a native language, such as Perl [14]), which allow direct memory manipulation, are prone to buffer errors since there is no original protection against accessing or overwriting data in the memory. The way in which buffer error vulnerabilities can be exploited differs based on the memory region location of the allocated buffer (this is discussed in detail in section 2.1.2). Also, exploiting buffer error techniques may vary depending on the mitigation mechanisms of the processor architecture and the operating system (this is discussed in section 2.1.3).

2.1.2 Buffer Error Exploiting Mechanisms

To understand how buffer errors are exploited, it is important to understand how the data memory is organized inside a process. A process is a program instance that is loaded into memory to be executed and managed by the OS. The exact layout of process memory depends on the OS and the implementation of the programming language (e.g., compiler, linker, and loader). Figure 2.1 shows process memory layout in a native program, which is generally divided into four segments: code, data, heap, and stack segments, each with its purpose. Typically, buffer errors in one segment could over-write data in the next segment in the memory. Here, we address each segment and its purpose and how buffer errors could be exploited:

1. *The code* or text segment stores the compiled binary code. This section is executable

and read-only in many architectures. A segment violation occurs when an attempt is made to write to this segment. In some architectures that allow self-modifying code, this segment is not read-only. This memory segment in the Unix-based systems is held at the lowest address, compared to other segments to restrict these segments from overrunning the code segment [168].

2. *The data* segment includes static/global variables that are initialized/uninitialized by the programmer (in the Unix systems, initialized variables are stored in the data section, while uninitialized variables are stored in the BSS segment). Data in this segment grows towards higher address spaces.

3. *The heap* segment is used to create dynamic data objects and store objects that must live longer than one function's lifetime. The heap is allocated by new, malloc, calloc, and realloc, among others, and removed by delete and free. This segment grows towards higher address spaces, and it is usually used when the size of the buffer required by the program is unknown or larger than the stack. Heap-based buffer errors in the heap could occur by manipulating heap management metadata stored before or after the actual data in a heap. The implementation of the heap differs depending on the OS and architecture. Some operating systems provide an allocator for malloc, while others provide functions to control specific regions of data [168]. Based on this, the heap exploiting varies based on the heap data structure. Hence, when heap data is overrun, the values in the management control block may be compromised. As a result, the memory management function (e.g., malloc and free in dlmalloc data structure) could be accessed to write arbitrary data at arbitrary memory locations or to execute malicious code (e.g., using unlinking or frontlinking attacks in dlmalloc or jemalloc heap corruption attacks [51]). This depends on the used memory management functions and their specific implementation.

4. *The stack* segment supports active subroutine execution of a computer program, and it is a last-in, first-out (LIFO) data structure. The stack is used to store local variables declared inside local subroutines and to store data related to subroutine calls, such as the return address and parameters. Stack variables usually grow towards lower address spaces in Unix-based systems. Buffer errors may lead to arbitrary code execution by altering a pointer to an address, such as modifying the return address (i.e., stack smashing), a frame- or stack-based exception handler pointer, function pointer, or other addresses to which control may be transferred [168]. Attackers typically perform stack-based buffer overflow by using an injected malicious code (e.g., stack smashing) or jumping to a new location in the code instruction (e.g., arc injection attacks).

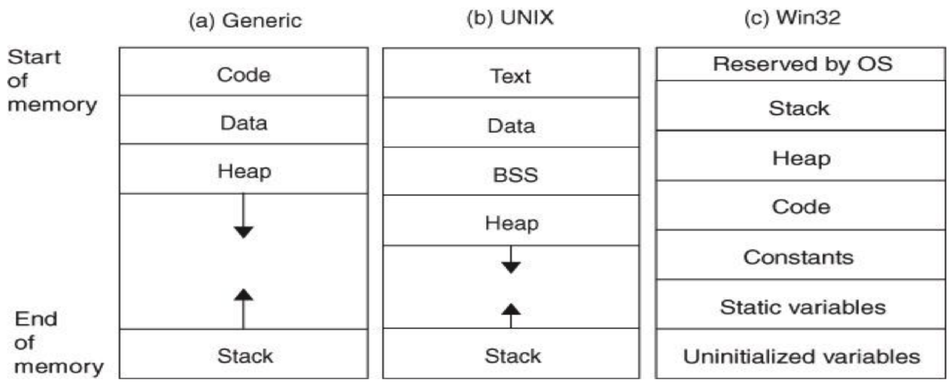


Figure 2.1: Memory layout of a process [168]

Generally, in the data/heap/stack segments, a buffer error may lead to a denial of service that crashes an entire application [168]. Additionally, in the data/heap/stack segments, a buffer over-read may reveal a variable value. On the other hand, a buffer over-write may occur and change a variable's value. Another way to transfer control to an arbitrary code given buffer errors is by using pointer subterfuge. As an illustration, overrunning pointers to data or functions in the data/heap/stack segments could transfer control to attacker-supplied shellcode [168]. In addition, pointer subterfuge may happen due to language features, such as longjmp(), atexit()/on_exit() functions defined in C standards [168]. Another example is that C++ allows the virtual functions to be overridden by a function of the same name in a derived class. This could be compromised by attackers to execute arbitrary code (i.e., VPTR smashing [20]). Moreover, pointer subterfuge could modify a pointer to exception handlers (e.g., Microsoft Windows Structured Exception Handling (SEH)). Beyond a single process, the *shared section* is a memory location that could be mapped into and out of multiple program address spaces to provide communication among them. Buffer errors have not been discussed in depth in this segment; however, they may occur, and buffer errors in this section may influence other applications that share the same objects [64].

In Android apps, the layout of process memory is organized differently. Android apps are written in Java or Kotlin code and sometimes may include the native code, and all of these components reside within the same virtual address space. Indeed, the nature of Android architecture requires the apps to share resources with the Android OS. The code segment holds app code and resources, such as *.dex* bytecode and *.jar/.so* libraries. Android apps have two different heaps: the Java heap and the native heap. The Java heap holds objects that are dynamically allocated from Java or Kotlin code. This segment is less likely to produce buffer errors.

The native heap, on the other hand, holds C/C++ objects that are dynamically allocated. This segment is always present even if the app does not use the native code because the Android framework uses native memory. This segment is where buffer errors could happen, similar to what we have seen before. Gong describes one way to exploit Android heap-based buffer errors [98]. In Android, the heap in the native code, is managed by particular APIs in the Bionic library [1] using the dynamic memory allocator, which is dlmalloc up to Kitkat and jemalloc since Lollipop; dlmalloc and jemalloc have different data structures and, thus, different exploit mechanisms [88] [51]. Moreover, the Android Run-Time (ART) model has a unified stack for native and Java code [27] (which is different from the old Dalvik model that had separate stacks for native and Java code). The stack in Android is also prone to typical stack-based buffer errors [54] [104]. Additionally, there is a shared memory segment (e.g., ashmem or mmap) that includes shared objects, among other processes prone to buffer errors. The next section provides some defensive mechanisms to protect against buffer errors.

2.1.3 Buffer Error Defense Mechanisms

Since buffer errors can lead to very dangerous vulnerabilities, multiple defense mechanisms have been proposed in the literature to protect against these types of vulnerabilities. We classify these techniques into two types: prevention techniques and elimination techniques.

2.1.3.1 Buffer Error Prevention Techniques

Prevention techniques block exploiting the vulnerability. These techniques usually terminate the program when buffer errors are detected during run-time. These techniques are provided as compiler extensions (e.g., StackGuard[72], ProPolice[85], and Return Address Defender (RAD)[67]) , OS modifications (e.g., Pax [28], LibSafe [203]), hardware modifications (e.g., processor-based [141]), or tools that capture buffer error attack symptoms (e.g., COVERS [134], TaintCheck [147]). As an illustration, Android has adopted some of the run-time protection mechanism features to mitigate buffer error attacks as follows:

- In Android 1.5 CupCake, ProPolice (also known as Stack Smashing Protector (SSP)) has been introduced, which is improved over StackGuard [72], to protect against

[1]https://android.googlesource.com/platform/bionic/

stack-based buffer errors [18]. ProPolice is a compiler feature that addresses stack smashing attacks by adding canary values just before the target address on the stack (arguments, return address, and previous frame pointer) [80]. Attempts to manipulate those addresses would lead to the canary values being changed, and then the program could be corrupted before the execution of malicious code could occur. Also, ProPolice reorders variables such that overrunning a buffer is less likely to damage other local variables. ProPolice can protect overwriting the arguments, the return address, the previous frame pointer, or local variables, but it cannot protect overwriting all types of local variables. For instance, the compiler cannot rearrange struct members [168]. This mechanism can also accidentally modify or crash the execution of non-malicious programs due to unknown bugs in the programs. Hence, this technique may not be applied to all applications on the platform [168]. Moreover, this mechanism does not protect against buffer errors occurring in the heap or data segments.

- In Android 2.3 Gingerbread, the hardware-level No eXecute (NX) technique has been added to prevent code execution on the stack and heap. However, this approach could be bypassed, such as by using a return-to-libc attack.

- In Android 4.0 Ice Cream Sandwich [18], Address Space Layout Randomization (ASLR) has been initially introduced, then ASLR was enhanced with newer versions of Android. ASLR randomizes memory addresses of the stack, heap, and libraries each time the memory is allocated for a process, such that finding executable code becomes unreliable. However, overcoming the mentioned techniques above is possible and sometimes simple [80] [137]. ASLR can be bypassed on a long-running process, such that attackers can discover the shellcode address without terminating the process using information leakage vulnerability. Another possible method is that an attacker can leverage the fact that Zygote's randomization makes identical address space layouts of all launched apps in one phone. Hence, an attacker could use a malicious app to leak the address space layout that would be identical to other launched apps. Real exploitation and code execution have already been proven in the wild [98].

- In Android 4.2 Jelly Beans and recent versions, a new feature has been added that protects against buffer errors, as all applications and system libraries during compile time are checked with the FORTIFY_SOURCE feature [18], which detects and stops certain types of buffer errors. This feature detects buffer errors in various functions that may perform unsafe operations on memory and strings; hence, it does not protect against all the types of buffer errors.

It is recognized that these prevention techniques should not be used solely and considered a second line of defense. These techniques may be bypassed as attackers get more intelligent, causing a denial of service (since the process is terminated when detected), or may involve performance penalties. Therefore, it is essential to detect buffer errors before software deployment. The next section discusses which methods detect buffer errors during the coding phase.

2.1.3.2 Buffer Error Elimination Techniques

Elimination techniques are mostly used for testing purposes of detecting and deleting the actual buffer error vulnerability from the source code, such as fuzzing, dynamic analysis, model checking, and static analysis. Fuzzing is an automated approach that injects unexpected random data as inputs to a running program until it crashes or fails, which may indicate the presence of vulnerabilities. Fuzzers can be black (e.g., Peach [13], Spike [22]), white (e.g., Dowser [103]), or grey-box (e.g., Munch [150]), depending on whether they are aware of program structure. Indeed, fuzzers tend to find simple bugs. Another run-time technique is dynamic analysis that monitors a running program, and once a buffer error is detected is logged to help developers to eliminate it (e.g, AddressSanitizer [169], Valgrind/Memcheck [26], Dr.Memory [6], Insure++ [12]). Model checkers are based on models that exhaustively check a program to prove properties in a mathematically precise manner [58]. In the case of buffer errors, model checkers prove properties such that every access to a specified buffer is safe (e.g., CBMC [122], BLAST [105]). Although model checkers can check for a rich set of properties, they often require much manual effort to build test harnesses. In general, the run-time approaches mentioned above are powerful; however, they involve some limitations due to running-time detection mode, such as performance overhead, inability to cover all the execution paths, and, sometimes, scalability drawbacks.

Contrarily, the static analysis method's goal is to find vulnerabilities. Static analysis tools do not have these drawbacks; thus, they have been prevalent for finding buffer errors. Studies show that the static analysis method is better than model checking in finding bugs [84]. That is because the static analysis method has less running-time and finds more errors. Engler and Musuvathi [84] mentioned that most of the static analysis tools require compiling the code, while model checkers require running on a carefully crafted environment model. Further, static analysis tools can cover all execution paths in the code, while other run-time methods only examine paths that are triggered during run-time. Generally, the static analysis method may outperform other methods in terms of scalability, code coverage, and less time and memory overhead. In the next section, an in-depth discussion about the static analysis method for buffer errors is provided.

2.1.4 Static Analysis Methods

Static analysis methods examine the program statically without executing it to detect software vulnerabilities. Many static analysis problems are undecidable as a result of Rice's theorem that indicates that SAST tools must use approximation techniques. Static analysis methods can analyze either the source or the binary code of the program. In our study, we only focus on tools and methods that support the source code analysis. Before we review state-of-the-art SAST tools, we provide an explanation of how different SAST tools that detect buffer errors are classified.

2.1.4.1 Classification of Static Analysis for Buffer Errors

Shahriar and Zulkernine [170] classify static analysis methods that target buffer errors based on inference algorithms, analysis sensitivity, analysis granularity, and target languages [170].

- *Inference method* is the main component of a static tool that indicates how the method systematically infers potential vulnerabilities. The inference methods are categorized into four types: string pattern matching, tainted data-flow, constraint-based, and annotation [170]. The string pattern matching method is one of the simplest methods used by SAST tools (e.g., RATS [17] and Flawfinder [39]). This technique tokenizes the program source code to identify a well-known set of tokens, such as library function calls that could cause buffer errors. The other three types (tainted data-flow, constraint-based, and annotation) involve more advanced analysis to understand the semantic of the source code [170]. In the tainted data-flow technique, untrusted inputs are marked as tainted, then tainted data propagation is calculated using a fixed point algorithm, such as constant propagation (e.g., Vulncheck [175]). On the other hand, the constraint-based technique produces constraints from the program code whose violations indicate vulnerabilities. Constraints are sometimes calculated and updated while traversing a program graph. The constraint-based technique may include integer range, symbolic value, or demand-driven analysis. The integer range analysis is based on analyzing the program statements that involve buffer declarations and operations. Further, each constraint is formulated in terms of a pair of integer ranges for the original buffer allocation size and the current size of a buffer for each buffer declared or accessed (e.g., Wagner et al., [197] and Ganapathy et al., [94]). In the symbolic value analysis approach, the constraints might include program variables and, whenever possible, their actual or symbolic values are assigned (e.g., ARCHER [204]). In the demand-driven method, programmers could formulate constraints to query vulnerable program locations. Additionally, the annotation-based inference method requires programmers to manually annotate a program code with pre- and post-conditions (e.g., Hackett et al., [102]). Annotations can be applied to both function declarations and program statements. An algorithm then evaluates if buffers can be used safely in functions/statements based on the annotated conditions.

- *Analysis sensitivity* refers to which pre-computed information based on the program code is used to perform the analysis, and it could be flow, path, context, point-to, or value range [170]. The flow-sensitive method means that the analysis runs based on the statement execution sequence with respect to a control flow graph (CFG) (e.g., Weber et al., [199]). Similarly, the path-sensitive analysis indicates that the analysis derives the execution paths from the CFG (e.g., Xie et al., [204]). The context-sensitive approach is able to distinguish multiple call sites of a function in regard to given arguments (e.g., Marple [126]). The point-to analysis method means that the method identifies memory variables that could be accessed by a given pointer variable (e.g., Dor et al., [78]). The value range analysis considers the lower and upper bounds of a variable (e.g., Vulncheck [175]).

- *Analysis granularity* indicates the granularity level of the program code at which an inference is performed [170]. The analysis granularity could be token, statement, intra-procedural, interprocedural, or system dependence graph. In the token level of granularity, the analysis method tokenizes the source code to determine keywords, variables, and functions (e.g., RATS [17] and Flawfinder [39]). In the statement level of granularity, the method analyzes all the program statements sequentially (e.g., Wagner et al., [197]). In the intra-procedural approach, the source code is analyzed based on the CFG or data-flow graph (DFG); however, the analysis is performed at the functional level (e.g., Dor et al., [78]). The inter-procedural approach, on the other hand, examines the whole program as one function (e.g., Xie et al., [204]). In the system dependence graph (SDG) level of granularity, the analysis method builds a node for each program point (e.g., statement), and an edge between two points representing dependency, which can be the control flow dependency or data-flow dependency (e.g., Ganapathy et al., [94] by using Codesurfer [184] and Weber et al., [199]).

2.1.4.2 Buffer Error Attributes in the Source Code

In this section, we discuss some elements of the source code that are associated with buffer errors. We present a taxonomy introduced by Kratkiewicz and Lippmann [120] and extended from Zitser et al., [210]. Zitser et al., [210] propose a taxonomy that contains thirteen attributes to describe C program buffer errors. The taxonomy was introduced to aid in developing test cases of a wide range of buffer errors and assessing static and dynamic analysis tools. Kratkiewicz and Lippmann have extended the taxonomy [120] and provided a comprehensive buffer error taxonomy that consists of twenty-two attributes [120], as shown in Table 2.1. This taxonomy is for C programs, but it could be extended to include C++ elements.

Table 2.1: Buffer error taxonomy attributes adopted from [120]

Attribute Number	Attribute Name	Attribute Number	Attribute Name
1	Write/Read	12	Alias of Buffer Index
2	Upper/Lower Bound	13	Local Control Flow
3	Data Type	14	Secondary Control Flow
4	Memory Location	15	Loop Structure
5	Scope	16	Loop Complexity
6	Container	17	Asynchrony
7	Pointer	18	Taint
8	Index Complexity	19	Run-time Environment Dependence
9	Address Complexity	20	Magnitude
10	Length/Limit Complexity	21	Continuous/Discrete
11	Alias of Buffer Address	22	Signed/Unsigned Mismatch

We note that some attributes are related to the actual buffer, and some attributes describe the operations on the buffer. Further, some attributes are associated with the buffer size, offset, or bounds, while others are related to the path to/from the buffer. We provide a detailed description of each attribute with some examples in Appendix A that are adapted from Kratkiewicz and Lippmann [120]. The *write/read* attribute describes the type of illegal memory access. An illegal write allows attackers to write more information into a smaller buffer leading to malicious code execution. In contrast, an illegal read may allow unauthorized access to information (e.g., Heartbleed) or be part of a multi-step exploit. Table A.1 in Appendix A shows the difference between illegal write/read memory access.

The *container* and *pointer* attributes describe the actual buffer holding data. The *container* attribute refers to the data structure that holds a buffer. Examples of such containers are struct and union, (which may be extended to other containers in C++, for example, a class). Buffer errors could occur within containers, such as when one array element in a struct overflows into the next variable, or beyond container boundaries. Moreover, the *pointer* attribute indicates whether a pointer dereference is used to access the buffer. The dereference could be used with or without an array index. Accordingly, we note that an array or a pointer would access a fine-grain buffer. Table A.2 in Appendix A shows some possible examples.

Some attributes related to numeral data are associated with the buffer; specifically, the buffer size, offset, or bounds. For instance, the *data type* attribute, which refers to the type of data stored in the buffer, could be related to the buffer size. This attribute may be a factor of a size miscalculation is some situations, such as casting. These primitive data types are shown in Table A.3 in Appendix A. Another related attribute to the buffer size is the *magnitude* attribute that describes the size in bytes of the buffer. The magnitude could be 1 byte, 8 bytes, or 4096 bytes; see Table A.4 in Appendix A, for example. Further, the *length/limit complexity* attribute is about the complexity of the length, or the limit of the buffer passed to a library function. The length/limit complexity may hold a constant, variable, linear expression, non-linear expression, function return value, or array contents, as shown in Table A.5 in Appendix A. Another vital attribute related to the buffer bounds is the *upper/lower bound* attribute that determines whether the upper or the lower buffer bound is overrun. While the most popular types of buffer errors exceed the upper bound, violating the lower bound is also possible. Table A.6 in Appendix A shows the difference between different types of illegal access of memory bounds.

We note that there is a subset of attributes related to the buffer offset that refers to accessing an individual buffer element. One instance is the *index complexity* attribute that denotes the complexity of the array index, and may possess a constant, variable, linear expression, non-linear expression, function return value, or array contents, as shown in Table A.7 in Appendix A. Further, the *address complexity* attribute indicates the complexity of the buffer address or the pointer computation. This might be a constant, variable, linear expression, non-linear expression, function return value, or array contents (Table A.8 in Appendix A depicts some examples). Moreover, the *alias of buffer address* attribute mentions if there is direct access to the buffer or through one or multiple levels of aliasing. One level of aliasing occurs when the original buffer address is assigned to a second variable; then, the second variable is used to access the buffer; Table A.9 in Appendix A includes some examples. This also occurs when passing the original buffer address to a second function. Further, the *alias of buffer index* attribute denotes whether the index is aliased or not. The index is aliased if it is a variable to which the second variable's value is assigned. Adding a variable assignment increases the level of aliasing; refer to Table A.10 in Appendix A for more clarification. Furthermore, the *continuous/discrete* attribute shows whether a buffer error accesses an arbitrary discrete buffer outside the allocated buffer or accesses continuous elements beyond the allocated buffer; see Table A.11 in Appendix A for more detail. The *signed/unsigned mismatch* attribute describes the situation where the buffer errors happen due to using a signed or unsigned value, where the opposite is expected. Usually, this happens when a signed value is used where an unsigned value is anticipated, and this is interpreted as a substantial unsigned or a positive value (see Table A.12 in Appendix A)

Some attributes focus on data flows to/from buffers; in particular, the *scope* attribute refers to different locations between the buffer allocation and the overrun. This can be a local, inter-procedural, global, inter-file/inter-procedural, or inter-file/global scope. In the local scope, the buffer is allocated and overrun within the same function. On the other hand, the inter-procedural and global scope occur within the same file. However, in the inter-procedural scope, the buffer is allocated in one function and overrun in another function, while in the global scope, the buffer is allocated as a global variable and is overrun in a function. The inter-file/inter-procedural and inter-file/global scopes are similar to the previous explanation, yet they happen across files instead of happening in the same file. As an illustration, when a C/C++ library function overruns the buffer, the overrun is inter-file/inter-procedural in scope and involves at least one alias of the buffer address. Furthermore, the *local control flow* attribute describes the control flow that immediately surrounds or affects the buffer, which could be either branching or jumping to another statement within the program, in particular *if, switch, cond, goto/label, setjmp/longjmp, function pointer, or recursion.* In the case of *if, switch, cond, function pointers, and recursion*, buffer errors happen within the conditional construct. Contrarily, in the *goto/label and setjmp/longjmp*, the buffer error is located at or after the target label or a longjmp address. Likewise, the *secondary control flow* attribute is similar to the local control flow attribute, and it describes if the control flaw precedes the buffer errors or includes a nested local control flow. Additionally, the *loop structure* attribute refers to the type of the loop construct that involves buffer errors, which might be *standard for, standard do- while, standard while, non-standard for, non-standard do-while, or non-standard while.* A *standard loop* is a loop construct that has typical components, such as initialization, a loop test, and an increment/decrement of a loop variable. A *non-standard loop* deviates from the standard loop in one or more of these areas. Besides, the *loop complexity* attribute indicates how many loop components, such as initialization, exit test, increment/decrements, are more complex than the standard baseline, particularly whether the initialization to a constant, testing against a constant, and incrementing/decrementing by one. TableA.13 in Appendix A shows some examples.

In addition to the above attributes related to the data flows to/from the buffer, the *asynchrony* attribute concerns whether buffer errors are involved in an asynchronous program construct, such as a thread, a forked process, or a signal handler. These constructs are implied by some OS-specific functions, for instance, thread functions, fork, wait, exit, and signal on Linux. Furthermore, the *taint* attribute shows whether buffer errors are affected externally, such as through argc/argv, environment variables, file read or stdin, socket, or process environment, as shown in Table A.14 in Appendix A. Finally, the *run-time environment dependence* attribute determines whether the buffer error relies on something determined at run-time, such as tainted data that depends on the run-time environment. Some attributes are related to how buffers are exploited, precisely the *memory location* attribute that involves where the buffer is allocated that could be stack, heap, data region, BSS, or shared memory (see Table A.15 in Appendix A)

2.1.5 Android Application Architecture and Buffer Errors

Android apps were used throughout the studies as an example of modern software. Android apps are unlike standard applications in two crucial ways. First, Android apps run in a security sandbox to manage application resources. 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 components in Android apps: activities are a single focused user interface, services run background tasks, content providers act as 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 [32].

Android apps are typically written in Java or Kotlin programming language. However, the Android Native Development Kit (NDK) provided by Google allows developers to implement Android apps using native programming languages, such as C and C++. Native code allows developers to use existing third-party libraries and allows hardware-specific optimization of performance-critical code. 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 also be 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. Java code is managed and executed by the Dalvik Virtual Machine (DVM)/Android run-time (ART)[2], while native code is not restricted to DVM/ART 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. The native components are also expected to interact with Java components carefully. If this interaction is not appropriately managed, the native components can cause errors that could crash the entire application [71].

2.2 Related Work

2.2.1 SAST Tool Warnings Assessment Using Test Cases

Evaluating SAST tools performance in terms of accuracy is a significant direction of research to understand the strengths and weaknesses of SAST tools and a foundation stone to build better tools. Multiple empirical studies have been performed to understand SAST performance [76] [120] [190] [210]; however, many of the studies used benchmarks or test cases (some with small-sized systems). Test cases are software projects with known vulnerabilities that have been curated and made available to everyone. Hence, the SAST tools could be tuned to detect the vulnerabilities in these test cases. For instance, Zitser et al., [210] have conducted a study on three open-source programs; BIND, WU-FTPD, and Sendmail contain 14 exploitable buffer overflow vulnerabilities to test five open-source SAST tools. While Torri et al., [190] provided a similar study, it was generalized for multiple vulnerability types present in five embedded systems. Kratkiew et al., [120] have evaluated five SAST tools using a corpus of 291 small C program test cases. Díaz and Bermejo [76] run nine SAST tools against two SAMATE reference dataset test suites for C language. All of the research mentioned above was conducted using test cases with real-world vulnerabilities known in advance to assess the false positives. The main idea of our work presented in Chapter 3, however, is to study the historical evolution captured in the repositories that are not part of any standard dataset to evaluate the false positives among the SAST-produced warnings on software projects in the wild.

[2]In recent Android versions, Java code is managed and executed by Android run-time (ART).

2.2.2 SAST Tools Warnings Against CVE Vulnerabilities

Another research direction is based on evaluating SAST tools warnings compared to real-world vulnerabilities. Edwards and Chen [81] examined historical releases of four large-scale projects; Sendmail, Postfix, the Apache httpd, and OpenSSL. The study's primary goal is to investigate warnings generated by three SAST tools: the HP Fortify Source Code Analyzer (SCA), IBM Rational AppScan, and Klocwork with some vulnerabilities published in the common vulnerabilities and exposures (CVE) dictionary. The authors examined the discovery rate of the vulnerabilities appearing in the CVE database for the four projects, and they correlated that to the output of SAST tools. The authors observed that the number of vulnerabilities does not always decrease with each new release. They also observed that the discovery rate of vulnerabilities begins to drop three to five years after the initial release. The study showed that the change in number and density of warnings generated by SAST tools is indicative of the change in the rate of discovery of vulnerabilities for new releases. The authors also demonstrated that SAST tools could be used to make some assessments of risk, even without a human review of the warnings. However, they considered all the warnings in aggregate and not tracing each warning to see if the warning ends up being a bug or even if it is removed, while our research in Chapter 3 traces SAST-produced warnings across historical versions of projects. Additionally, their project aims to see if the number of warnings correlates with actual vulnerabilities and not about studying each warning type's characteristics.

2.2.3 SAST Tools Warnings over Time

Another research direction has focused on tracing SAST-produced warnings over time since this approach can give more realistic results [75] [116] [176]. Spacco et al., [176] proposed a technique for tracking bugs across versions by using repository history and Findbugs. Kim and Ernst [116] introduced a bug warning prioritization method based on the software change history. They used bug warnings generated by three SAST tools for Java: FindBugs, Jlint, and PMD for three programs: Columba, Lucene, and Scarab. In that study, only bug warnings that are removed by fix-changes are considered indicating real bugs. They found that bugs that were fixed in fix-changes represent a tiny percentage of bug removals, and about 90% of the bug warnings either remain in the program or are fixed in non-fix changes considered being false positive warnings [116]. In Chapter 3, we follow a similar approach to understand the evolution of the SAST-produced warnings using historical information. However, Kim and Ernst neglected bug warnings that appear in non-fix changes; we consider all bug warnings in fix or non-fix changes. This is because researchers have emphasized that linking bugs to the comments in the commit logs may produce biased results [57][113]. For instance, Bachmann et al., [57] found that the bugs issued in a bug tracking system could be biased since not all bugs are reported to these systems. Another aspect is that sometimes it is difficult to isolate code changes due to bugs from other code changes due to different reasons. For instance, Murphy et al., [142] reported that 75% of surveyed developers had to remove features from software to fix bugs, and are therefore not tagged as bugs. Hence, we consider all commits and see if the warnings remain or disappear in subsequent versions.

Di Penta et al., [75] performed an empirical study to examine the evolution and decay of bugs in three networking systems detected by three freely available SAST tools. The primary focus of that study is to investigate warning categories and decay time. However, our approach in Chapter 3 is different from Di Penta et al., [75] approach in multiple aspects. One aspect is that we conduct our study on 116 open source projects that belong to various categories instead of only three networking systems. Also, we use six free open source and commercial tools to conduct our analysis. Finally, we adopt a well-known security bug classification to aggregate the results across tools and provide universal and comprehensive results.

2.2.4 Validating SAST Tools Warnings

Validating SAST tool warnings is an important step to conclude whether a given warning is false positive or true positive. This is usually performed by a human manual review. This method has been adopted in research, as few papers involve manual examination to evaluate SAST-produced warnings [55]. Ayewah et al., [55] run FindBugs against multiple large projects, such as the JDK, Glassfish J2EE server from Sun, and some portions of Google's Java codebase. The authors found that Findbugs reports real but trivial bugs. However, this approach can be impractical when it comes to evaluating a large number of warnings. Thus, Di Penta et al., [75] used the *ldiff* framework, tracing the source code changes across subsequent commits to extract needed information, and they made assumptions to decide whether a given warning is a false or true positive. In Chapter 3, we use a similar method by employing a novel tool to trace the line codes over time. We use a different tracing tool, namely *cregit*$^+$, that traces tokens across commits and allows it to detect line split and merge in subsequent versions and keeps track of the refactored files that cannot be done by the *ldiff* framework. This approach makes the analysis scalable to trace warnings in hundreds of software projects.

2.2.5 Evaluation SAST Tools for Buffer Errors in Android Apps

Although multiple studies have evaluated the use of static analysis tools for buffer error detection [210] [106] [190] [156], to the best of our knowledge, Chapter 4 introduces the first empirical study that evaluated these tools against known buffer errors in the Android apps domain. Our results expose the need for more advanced static analysis tools to detect buffer errors in Android apps considering Android app nature.

2.2.6 SAST Tools for Android

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 [167]. For instance, the SCanDroid tool [92] performs a data-flow analysis of installed Android apps to track inter-component communication through intents to detect the potential violation of permission through a coalition of applications. Similarly, Quire [77] tracks permissions through the IPC call to prevent privilege attacks among applications. Besides, several tools have been developed to detect private data leakage. TaintDroid [83] performs dynamic taint analysis, while FlowDroid [52], LeakMiner [206], and AndroidLeaks [96] are static analysis approaches to detect data leakage. VulHunter [157] 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 Chapter 4, our research looks at buffer error vulnerabilities in Android apps. However, 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. Indeed, in 2017, Li et al., [131] studied 124 published research papers to investigate state-of-the-art static analysis approaches in Android apps. Likewise, in 2017 Sadeghi et al., [167] conducted a systematic literature review by analyzing 336 published research papers to reveal state-of-the-art research in the Android security field. Both papers [131] [167] do not mention any tool or approach to detect buffer errors in Android apps.

Since Android apps might involve many programming languages, the security of Android apps that include native code has gained more attention [205] [131]. In [125] and [40], the lack of static analysis methods that understand both the Android code and the native code has been recently highlighted. DroidNative [41] uses a static analysis technique to detect Android malware to reduce the effect of obfuscations and provide automation. It was recognized that the native code in Android apps could reach all Android APIs and affect the whole application. The JN-SAF framework [200] performs an inter-language data-flow analysis that is a flow and context-sensitive approach that operates on the binary code.

However, statically detecting buffer errors in Android apps source code that may involve cross-languages has not been fully addressed. For that reason, Chapter 6 proposes a methodology to statically detect buffer errors that can work universally with any framework and analyze Android apps. The analysis is performed at the LLVM IR level and works on C/C++ code for now, and it considers all the cross languages link in such analysis. This can be extended into Java in the IR form as well. This can be achieved when using platforms (e.g., JLang [35]) that translate Java into LLVM IR. Currently, these platforms support Java but not Android code.

2.2.7 Android Vulnerabilities Pattern Analysis

Several studies have been conducted to understand the vulnerability patterns in Android apps. Huang et al., [109] studied the Android mobile vulnerability market to reveal the unique vulnerability patterns of mobile software generally. The study shows that the Android market vulnerabilities are more exploitable than in the entire market, and the exploitation impact is higher based on CVSS metrics. Our research in Chapter 4 took further steps to reveal the pattern of these vulnerabilities in Android apps.

Watanabe et al., analyzed 2M free Android apps and 30K paid Android apps to study Android app vulnerabilities associated with software libraries. They found that approximately 70%-50% of vulnerabilities of Android free/paid apps come from third-party libraries [198]. However, the analysis is based on DROID-V, which depends on five other static analyzers that are looking for limited types of vulnerabilities. Also, the used tool may include false positives. We studied real-world vulnerabilities published in public to reveal the trend in Android apps.

Jimenez et al., conducted a manual analysis of 42 Android vulnerabilities with associated patches published in the NVD for the period 2008 to 2014 [111]. They found that 70% of the vulnerabilities originate from the coding part. They also determined that incorrect implementation of a feature is the leading Android vulnerability, input validation is the second, and buffer errors are among the third most frequent vulnerabilities. However, their study includes Android apps and the Android system (except the kernel vulnerabilities). In contrast, our dataset includes 476 Android vulnerabilities that are only related to Android apps.

2.2.8 SAST Tools for Cross-language

Since multiple programming languages have been widely used to implement single software, there are multiple methods that support multi-language program understanding, factoring, and management [189] [154] [207] [155] [194] [73]. However, little attention has been given to address challenges in the field of static analysis approaches for multi-lingual projects [143]. Research indicates that multi-lingual projects are more defect prone [118]. Kochhar et al., [118] found specific languages, such as C++, Objective-C, and Java, are statistically significantly more defect prone across all bug categories when used in multi-language projects. Further, they mention that the effect is the strongest for memory and concurrency bugs. This is why research has focused on detecting software bugs in multi-lingual projects.

One research direction focuses on analyzing Foreign Function Interfaces (FFI). Kondoh and Onodera [119] introduced a bug-finding tool called BEAM that operates on the JNI code that detects bugs, such as mistakes of error checking, memory leaks, invalid uses of a local reference, and JNI function calls in critical regions. This work only focuses on bugs within the JNI code and does not consider cross communications information. Similarly, Li and Tan [133] proposed a static analysis framework to examine exception handling in JNI programs. Their framework can be applied to other foreign function interfaces, including the Python/C and the OCaml/C interfaces. Also, Lee et al., [127] discussed detecting FFI violations regarding JNI and Python/C. They provided bug detection tools for JNI and Python/C using the discussed approach.

Another research direction depends on isolating the FFI. Siefers et al., [172] presented Robusta, which is a framework that includes static analysis techniques to detect bugs in programs using JNI. Their analysis detects bugs, such as exception handling, memory leaks, and invalid local references. However, it aims at isolating native code into a sandbox to prevent unsafe system modifications and confidentiality violations. Similarly, Chisnall et al., [66] provided CHERI JNI, which is based on a sandbox approach that focuses on memory safety issues.

Detecting bugs in the FFI is an essential aspect of such an analysis. A fundamental problem happens when a static analysis tool misses information related to bugs scattered across languages. Tan and Croft [185] conducted an empirical security study on the native code portion of the Sun's JDK 1.6. They used ITS4, Flawfinder, and Splint to carry out the analyses. They mentioned that security-critical vulnerabilities like buffer errors are manifest in the native part of the program and go undiscovered. As a result, they mentioned that the reason is that SAST tools only analyze C code, without considering how Java code interacts with C; hence, they emphasize building an inter-language analysis tool to detect vulnerabilities in cross-languages settings. Furr and Foster [93] introduced an algorithm for checking type safety across a foreign function interface, in particular between OCaml and C. The system prevents foreign function calls in C from introducing type-related bugs into a safe language like OCaml. Thus, the inference system accepts a source code written in both OCaml and C and as an input, and then analyzes the types in the source code. The authors then [93] proposed a multi-lingual type inference system to detect type safety violations through JNI. Lee [128] proposed a JNI program analysis technique to analyze Java and the C code in JNI using analyzers for each language. The author proposed using a semantic summary for each C function callable from Java and based on that constructing a call graph. The author also showed using this technique to detect four bug types: missing exception handling, descriptor mismatching, a nondeterministic native function call, and use of garbagable Java objects. Wei et al., [200] introduced JN-SAF tool that analyzes Android binaries to capture the inter-language data-flow in Android apps. This tool operates on the binary code, and it is based on symbolic execution—nevertheless, the evaluation of the tool was conducted with small size code.

To support this aspect of research, in Chapter 5, we introduced a static analysis methodology that supports cross-language analysis. The methodology is based on lexical analysis to locate all sources and sinks across-language. It also uses the lexical analysis to extract the cross-language semantic to construct the FFI call graph. Then it uses the call graph as a coarser-grain abstraction to filter out the unlikely reachable source and sink. This technique supports solving security vulnerabilities that involve source and sink analysis. In Chapter 6, a methodology was proposed to find buffer errors that can cross from Java, which takes into consideration the FFI code.

2.2.9 Static Analysis Tools for Buffer Errors

As buffer errors are well-known problems, multiple tools and methods have already been proposed in academia and industry. We observe that the introduced SAST tools and approaches for buffer errors usually target homogeneous single source code applications. Table 4.3 shows some open-source SAST tools that detect buffer errors and the underlying analysis techniques. Among the popular tools that have been introduced in academia is the Secure Programming Lint (Splint) [87], which is an annotations-based tool built upon LCLint [86]. Splint detects buffer errors and other security vulnerabilities in C programs. Another tool is UNO [107], which is a constraint-based tool that detects uninitialized variables, dereferencing Null-pointers, and out-of-bound array indexing. Similarly, Buffer Overflow Detection (BOON) is a constraint-based tool that detects buffer error vulnerabilities by considering buffer manipulation as an integer range constraint problem [197].

Likewise, ARCHER [204] is a constraint-based tool to find buffer errors in arrays and pointer dereference using path-sensitive and inter-procedural symbolic value analysis.

Table 2.2: State-of-the-art SAST tools to detect buffer errors

Method Name	Type	Language	Inference Algorithm	Sensitivity	Granularity
Polyspace Bug Finder [210] [30] [82]	Commercial	C/C++	Constraint: abstract interpretation, symbolic value	Flow, point-to	Inter-procedural
Parasoft C/C++test	Commercial	C/C++	String pattern matching, Constraint: symbolic value	Flow	Inter-procedural
Klocwork [82]	Commercial	C/C++	Unpublished	Flow, path	Inter-procedural
Coverity [82]	Commercial	C/C++	Statistical analysis	Flow, path, context	Inter-procedural
Parfait [70] by Sun Microsystems	Property	C/C++	Constarin: symbolic value, demand-driven, taint	Flow , points-to	inter-procedural
PREfast/PREfix [146] by Microsoft	Property	C/C++	Annotation	Flow	intra-procedural, inter-procedural
PVS-Studio [30]	Commercial	C/C++	Constraint: symbolic value, annotations	Flow, path, value range	Inter-procedural
CodeSonar [30]	Commercial	C/C++	Constraint: symbolic value, taint data-flow	Flow, path	Inter-procedural
Flawfinder [190][106]	Open source	C/C++	String pattern matching	N/A	Token
RATS [190][106]	Open source	C/C++	String pattern matching	N/A	Token
Cppcheck [190][106]	Open source	C/C++	Constraint: integer range	Flow, context	Inter-procedural
Clang Static Analyzer [2]	Open source	C/C++	Constraint: symbolic value, annotation	Flow, path	Inter-procedural
Farma-C [30]	Open source	C/C++	Constraint: abstract interpretation, annotations	Flow, value range, point-to	Inter-procedural, System dependence graph
IKOS [62]	Open source	C/C++	Constraint: abstract interpretation	Flow, path, point-to	Inter-procedural
ASTREE [106][30]	Commercial	C	Constraint: abstract interpretation	Context	Inter-procedural
ARCHER [210]	Open source	C	Constraint: symbolic value	Flow, path, context, point-to	Inter-procedural
BOON [210][106]	Open source	C	Constraint: symbolic value	N/A	Inter-procedural
Splint [87] [210][190][106]	Open source	C	Annotation	Flow	Intra-procedural, inter-procedural
UNO [210][30][190]	Open source	C	Constraint: value ranges	Flow, path	Intra-procedural

However, all of these tools focus only on the C programming language. When looking at the proposed methods, a similar issue is noticed, as the methods target only C. For instance, Weber et al., (Mjolnir) [199], Ognawala et al., (MACKE) [151], Gjomemo et al., (MESCs) [97], Chess [65], Wagner et al., [197], Kim et al., [117], Hackett et al., [102], Dor et al., (CSSV) [170], Sotirov (Vulncheck) [175], Ganapathy et al., [94], Avots et al., [53], Larochelle and Evans [23], and Livshits and Lam [135] all introduced academic prototype approaches that only focus on C. Methods designed for C are not entirely applicable to C++.

There are some academic prototypes and tools that target both C/C++. RATS [17] and Flawfinder [39] are token-based tools that detect well-known vulnerable library function calls. These tools are simple and can scale to examine large programs; however, they suffer from a high false positive rate. Cppcheck [138] is a simple context-sensitive that uses the Abstract Syntax Tree (AST). Cppcheck primarily detects different vulnerabilities, such as out-of-bounds checking, memory leaks, and Null pointer dereference. Although this tool does not produce a high false alarm, it detects simple issues.

On the other hand, there are open source tools and methods that target C++ and perform in-depth analysis to reveal more complex issues and avoid the high rate of false positive. For instance, IKOS [62] is an open-source analyzer based on the abstract interpretation method, and it performs an inter-procedural buffer error analysis. IKOS uses the GCC/LLVM compiler framework that supports both C/C++ to understand the semantics of the software being analyzed. Similarly, Frama-Clang [24] is a plug-in based on Frama-C for parsing C++ source files, and it uses Clang/LLVM. The Clang Static Analyzer [2] employs a constraint-based symbolic value analysis that is performed at the compilation unit level. The issue with these tools that preform in-depth analysis, such as IKOS and Frama-Clang, and Clang Static Analyzer, is that sometimes they do not scale well.

Some academic prototypes were introduced to be applied to larger programs to provide scalability. Marple [126] is a static analysis method for buffer overflow that uses a demand-driven approach to be scalable. This method examines all possible paths backward from each buffer access and uses an external constraint solver to solve buffer error constraints. Li et al., [132] enhanced the scalability of Marple [126] by only solving linearly-related data dependencies and control dependencies when computing the symbolic values of a program variable. This methodology scales well to millions LOC, yet it only focuses on array access. The approach that we implemented in BEFinder was inspired by [126] and [132], as it does demand-driven analysis and constraint-based analysis. Unlike Marple [126], BEFinder in Chapter 6 uses sparse analysis to solve buffer errors problem. Moreover, BEFinder not only analyzes arrays but also covers errors involved in standard function calls. Furthermore, BEFinder considers cross-language communications from Java code.

2.2.10 Taint Analysis

Taint checking is a security approach that was originally introduced with the Perl language [15]. The purpose of the taint mode in Perl is to block malicious attacks from manipulating the program at run-time. Then, taint analysis has emerged as a beneficial program analysis approach to statically or dynamically detect security vulnerabilities [171] [90] [147]. This study concentrates on static taint analysis. Most of the static taint analysis approaches focus on one language at a time and provide an in-depth taint analysis approach based on a fine-grain level of program representation, such as the program dependence graph (PDG) and the exploded super-graph.

For instance, the Taint Analysis for Java (TAJ) is a static analysis technique for Java that performs a demand-driven traversal over a Hybrid System Dependence Graph (HSDG)[191]. Moreover, Livshits and Lam [135] introduced a static taint analysis technique for C using an augmented SSA form called IPSSA. Snelting et al., [173] proposed a static tainting analysis approach for C based on the Program Dependence Graph (PDG). FlowDroid [52] is a state-of-the-art highly-precise static taint analysis for Android based on the IFDS framework. The IFDS solves inter-procedural, finite, distributive subset problems using the exploded super-graph [161]. The super-graph considers every program statement as a node. Then, data is propagated along the super-graph to determine the reachability. However, this would be impractical with many source and sink pairs. SoS that we proposed does not perform the analysis based on a fine-grain representation of a program, but rather, it performs lightweight analysis and looks at the larger image and filter out all the unreachable pairs the call graph level. This way, sources and sinks that cross languages are supported. Generally, the purpose of SoS is to eliminate all the pairs that cannot be reached in the call graph and provide the pairs that "can" be reached. Hence, the advanced static analysis tools can operate on this subset to give the ultimate answer.

Recently, taint analysis for cross-language has gained increasing attention from researchers. Kreindl et al., [121] investigated the limitation of multi-language dynamic taint analysis to provide a language-agnostic framework. They also provide a platform based on the GraalVM, a multi-language virtual machine based on the Truffle framework. Truffle framework parses programs into a common abstract syntax tree representation, and hence this common AST was used for such analysis. Similarly, Azadmanesh et al., [56] introduced a language-independent information-flow analysis tool based on Truffle to support dynamic analysis. However, SoS in Chapter 5 supports static analysis rather than dynamic analysis.

The JN-SAF framework [200] performs a static cross-language data-flow analysis that operates on the Android binaries. SoS in Chapter 5 provides a general methodology to reduce the potential source and sink pairs and can support many languages. BEFinder that we introduced in Chapter 6 supports taint analysis and considering data that cross languages. One concern of most of the taint analysis approach is that there is little discussion about how many sources and the sinks can be discovered by such a tool and methodology. For that reason, most of the static taint analysis tools support a limited number of sources and sinks. Covering a vast number of sources and sinks can be very costly, and hence SoS in Chapter 5 can help in such an aspect.

Chapter 3

An Empirical Study of Security Warnings from Static Application Security Testing Tools

The Open Web Application Security Project (OWASP) defines Static Application Security Testing (SAST) tools as those that can help find security vulnerabilities in the source code or compiled code of the software. Such tools detect and classify the vulnerability warnings into one of many types (e.g., input validation and representation). It is well known that these tools produce high numbers of false positive warnings. However, what is not known is if specific types of warnings have a higher predisposition to be false positives or not. Therefore, this chapter's primary goal is to investigate the different types of SAST-produced warnings and their evolution over time to determine if one type of warning is more likely to have false positives than others.

Related Publications

The work described in this chapter has been previously published:

- Aloraini, B., Nagappan, M., German, D. M., Hayashi, S., & Higo, Y. (2019). An empirical study of security warnings from static application security testing tools. Journal of Systems and Software, 158, 110427.

3.1 Introduction

The SAST tools have not been well adopted, as they may produce a high rate of false positive alarms since most warnings do not indicate real vulnerabilities [68][112]. Hence, an understanding of the warnings generated by SAST tools, their characteristics, including their distribution across various types, and decay time, could help improve the quality of the SAST tools. For instance, if we find that many warnings of a specific type are indeed false positives, then the developers of SAST tools could build a better detection algorithm for that type of warning or reduce its severity score. Alternatively, based on our approach and potentially our results, developers could ignore the warnings that are more likely to be false positives more readily. Testing SAST tools and examining the characteristics of the warnings produced on software projects in the wild where one does not know where vulnerabilities exist, can be more useful. A study by the National Security Agency (NSA) [89] has emphasized the importance of evaluating SAST tools against real software projects to precisely predict the frequencies of warnings and detected vulnerabilities outside of controlled studies.

Thus, we extend the study by Di Penta et al., [75] and investigate the nature of software warnings as generated by SAST tools for C++ projects. Instead of studying only three networking systems, we apply the study on 116 real projects that belong to different categories, as well as four free and open-source SAST tools and two commercial closed source SAST tools. We use *cregit*$^+$, which is a more recent version of the state-of-the-art tool called cregit [95]. *cregit*$^+$takes full advantage of version control, which helps to track file renames. It also tracks the source code at a finer granularity (at the token level rather than line level), helping to detect line split and merge. Our working assumption (like that of Di Penta et al., [75]) is that if a warning was not removed and remains in the same piece of code across many versions for a long time (in our case, over five years), then the warning of that type may not be that important since the warning was not considered worth removing under any circumstance. Thus, that warning is more likely to be a false positive.

More specifically, the contributions of our study are as follows:

- We first examine the distribution of the various types of SAST-produced warnings through time.
- We investigate how many of the warnings of each type remain over time.
- We also examine the decay rate for warnings belonging to different categories and produced by various SAST tools. This is the time interval between the discovery and

removal of warnings, to see how long various types of warnings tend to stay in the project.

- Finally, we investigate the distribution of real-world vulnerability categories fixed by the project team in the studied projects to see if the results are consistent with the rest of the study.

3.2 Definitions

Below, we define some terms that we will use in the rest of the chapter, which is key to understanding our analysis. Note that our analysis examines the first available versions of the software projects in 2012 and 2017. We then see if the warnings that existed in 2012 remain in 2017 or not. Note that we do not know if the software developers used any of the SAST tools in this study. We are not claiming that. We want to see if warnings present in the 2012 version of the software were removed for any reason by the developers of the software in a five-year time period. This is similar to the context under which Di Penta et al., carried out their study [75].

Below, we present the various scenarios of what could happen to warnings from the 2012 version.

- Remained Warnings - When a line of code in the 2012 version also exists in the 2017 version (even if it has moved location within the file, or the line of code was modified (e.g., a variable name was changed)), and in both versions, the same warning exists. These warnings are likely to be false positives (FP); even though the tool identified a potential problem within the line, the warning has not been removed from the line, indicating that no problem has been found so far in it. Note that the typical time to fix a bug is around three years [63][75].

- Modified Warnings - When a line of code in the 2012 version also exists in the 2017 version, and the warning in the 2017 version is different from the warning in the 2012 version. These warnings are likely to be true positives (TP), as the project developers have made an effort to change the code that led to the disappearance of the original bug warning, yet they have introduced a new bug, which indicates incorrect fixes or buggy patches [208]. For instance, a bug warning might be raised because of the usage of strcpy function call. The strcpy function call copies a source buffer to a destination buffer, and it has two arguments: destination and source. So, if the source buffer is larger than the destination buffer, that might cause a buffer overflow. A developer might change the code by replacing the strcpy function call by strncpy function call with three arguments (destination, source, and the number of copied bytes from source to destination). However, strncpy function call has other problems. For instance, it does not supply null termination at the destination buffer. (This category shows two aspects: on the one hand: the tool was able to spot a real bug, on the other hand: the developer incorrectly fixed that warning)

- Disappeared Warnings - When a line of code in the 2012 version also exists in the 2017 version, but the warning present in the 2012 version disappears, and there is no new warning in the 2017 version. We consider that a Disappeared Warning indicates that the line was improved to remove the warning (and potentially do other work) and, thus, a true positive (TP).

- Removed Warnings - When a line of code in the 2012 version *does not* exist in the 2017 version, and hence the warning from 2012 also does not exist in the 2017 version. We cautiously decided to make no assertion about Removed Warnings (the line no longer exists). The reason is that we do not know if the line was removed due to a bug fix or because the feature is no longer necessary.

In Di Penta et al., work [75], they referred to warnings as 'Vulnerability.' We are not sure if they include *Modified Warnings* in Disappeared Vulnerabilities. In this chapter, we chose not to present or discuss the results of removed warnings because we do not know why the line was removed. However, the complete dataset from our study that we share [43] has the removed warnings classified, if the reader wants to examine them.

3.3 Research Questions

- **RQ1: How are the warnings from each SAST tool distributed across different warning types in 2012 and 2017?** This research question discusses the distribution of warnings detected by various SAST tools across the various warning types at different time periods: 2012 and 2017. This helps us understand the emergence of warnings to see whether the density of a particular warning type shows any positive or negative trend between 2012 and 2017.

- **RQ2: How are true positive and false positive warnings distributed across different warning types and across different SAST tools?** Our goal is to understand the distribution of true positive (TP) and false positive (FP) warnings that belong to different types as reported by SAST tools and have been removed later from the source code. In particular, we are interested in warnings that have been reported by SAST tools in 2012 and their status in 2017. As per our definitions in Section 3.2, we present the results of *Remained Warnings* (which are likely to be FP), *Modified Warnings*, and *Disappeared Warnings* (which are likely to be TP). Hence, we trace the lines of code across each project's history using *cregit*$^+$.

- **RQ3: How do different types of warnings evolve over time?** We are interested in warning decay, indicating the time interval between detection until removal. To clarify this, we provide this simple scenario: assume that a new project manager decides to run SAST tools in a particular period of time (in our case in 2012). We measure how long it took for the warning to disappear. By knowing this, we would be able to see if warnings of a particular type are removed sooner than others. In RQ2, we examine if the warning disappears by 2017, and now when between 2012 and 2017, it disappears. Hence, we focus on TP warnings (*Modified Warnings* and *Disappeared Warnings*) that were detected by SAST tools and likely fixed in the code. We only consider lines of code that include warnings and have been altered, as this gives us the ability to trace the changes in the lines of code. Whenever we find a warning that belongs to an altered line of code and disappeared in the recent version, we trace the line of code that contains the bug warning backward during the history to find the fixing time and flag that to be a *bug fix* version.

3.4 Experimental Design

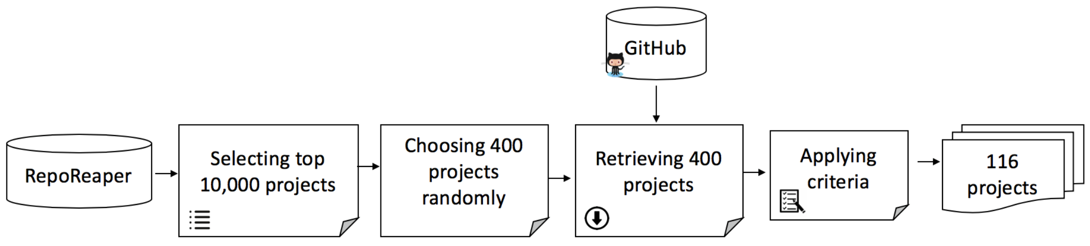


Figure 3.1: Case study project selection

3.4.1 Selecting Case Study Projects

When we gathered projects to conduct the study, we considered the following four criteria:

- **Real-world projects:** We aim at understanding SAST-produced warnings against large real-world projects. This allows for assessing software project warnings at scale and evaluate SAST tools performance.
- **Programming language:** We choose to analyze C++ projects, since it is ranked as the fourth most popular programming language based on the TIOBE Index as of August 2020 [25]. Additionally, C++ is generally more bug-prone as well [159].
- **Active:** We mine projects with adequately long and rich historical data, as those projects are more likely to be more mature and active. Thus, we focus on projects with high popularity and projects that have long and continuous development histories for at least five consecutive years, from 2012 to 2017.
- **Automation:** As we need to run SAST tools on the projects, and some of the SAST tools need to run on top of a build system, we limit retrieved projects to those that use automated build systems. We choose to use projects that adopt the *cmake* build systems.

As shown in Figure 3.1, we use projects hosted on GitHub because we want to analyze *real-world projects*. To compose the list of repositories, we make use of the metadata provided by the RepoReaper dataset [145], since it eliminates the noise (e.g., repositories that are not engineered software projects). Since we want to focus on the *C++ programming language*, we filter the RepoReapers dataset to obtain only projects that were written in C++. We sort the RepoReapers project names by the highest number of stars; then, we select the top 15,000 projects on the list. After that, we randomly choose and clone 400 projects from the 15,000 list. To ensure the *activeness* of the projects, we exclude those with fewer than 120 commits.

Moreover, we exclude projects that do not have an active development history for at least five consecutive years, from 2012 to 2017. To fulfill the *automation* criterion, we eliminate all projects that do not adopt the *cmake* build system and could not be built smoothly to run all SAST tools (e.g., due to dependency issues). Finally, we have 116 projects that fulfill all the above criteria.

3.4.2 Selecting SAST Tools

When selecting the SAST tools that we want to study in this chapter (as shown in Figure 3.2), we consider their diversity in terms of availability and underlying inference algorithms to provide various types of security warnings. Consequently, we gather the following eight SAST tools that analyze C++ source code: Parasoft C/C++test [11], PVS-Studio [16], Clang Static Analyzer [2], Cppcheck [138], Flawfinder [39], RATS [17], Polyspace Bug Finder [139], and Coverity Static Analysis [184] (academic version). However, we find that some of the SAST tools have restrictions on the number of analyzed lines of code (LOC), such as the Coverity Static Analysis tool (academic version). Also, some of the SAST tools do not generate warnings in an efficient format to analyze. For instance, the Polyspace Bug Finder only shows the warnings using a graphical user interface, which is highly ineffective in analyzing thousands of warnings. Hence, we exclude them and study only six SAST tools. Table 3.1 summarizes all SAST tools used and related information. Table 3.1 illustrates different SAST tools with their underlying inference algorithms to infer the presence of security bugs.

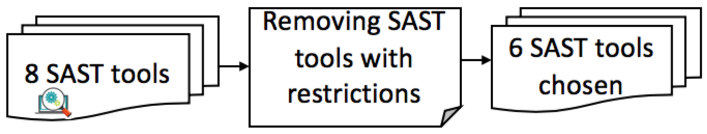


Figure 3.2: Selecting static application security testing tools

3.4.3 Extracting Security Warnings

After choosing the SAST tools and projects, we need to extract the warnings that each SAST tool finds in each of the projects. The approach we use for this step is shown in Figure 3.3. One of our goals is to see how the warnings evolve over time. Therefore, for each project, we check out two releases of their source code: the first release from 2012 and the first release from 2017. We choose 2012 and 2017, so we can be reasonably sure that warnings from 2012 that are not removed in 2017 do not affect the reliability of the product. We then run each of the SAST tools in 2012 and 2017 releases of every one of the 116 projects and collect the warnings.

We use all available checkers and analysis modules in SAST tools (e.g., when using the Clang Static Analyzer, we enable all available checkers in the tool). Next, we revise all

Table 3.1: Studied static application security testing tools for C++

Tool Name	Version	Availability	Inference Algorithm
RATS	2.4	Open source	Pattern-matching
Flawfinder	1.31	Open source	Pattern-matching
Cppcheck	1.76.1	Open source	Integer range analysis & data-flow analysis
PVS-Studio	6.13	Commercial*	Pattern-matching & symbolic execution & data-flow analysis annotation-based (automatic)
Parasoft C/C++test	9.6.1	Commercial	Pattern-matching & abstract interpretation & data-flow analysis
Clang Static Analyzer	3.8	open source	Symbolic execution & annotation-based & data-flow analysis

* PVS-Studio provides a free academic license.

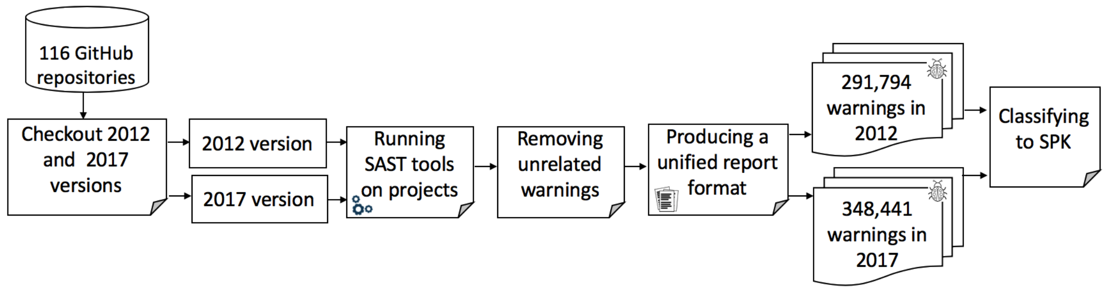

116 GitHub repositories
Checkout 2012 and 2017 versions
2012 version
2017 version
Running SAST tools on projects
Removing unrelated warnings
Producing a unified report format
291,794 warnings in 2012
348,441 warnings in 2017
Classifying to SPK

Figure 3.3: Extracting security warnings

the generated reports to ensure that they are relevant to the actual source code, as we are only interested in source code files that represent the program code, and not source code files for test purposes. Therefore, we exclude all warnings in test files. We infer that from the file path, which usually includes the keyword "test."

Furthermore, we only consider warnings that happened in C and C++ source and header files for our analysis. As a result, we find 291,794 warnings in 2012 and 348,441 warnings in 2017 when considering all the core projects and the linked sub-modules in our dataset, for all the six SAST tools. Finally, since each SAST tool has its warning format, we produce a consistent and unified format among SAST tools. Therefore, we save every warning from all SAST tools in one format in one file to process later.

3.4.4 Classifying Security Warnings

One of the challenges that we face is that the security warning classifications used by the SAST tools are different among the various SAST tools, making it difficult to analyze them in aggregate. Hence, we need to have one consistent classification across the SAST tools to simplify the analysis. We map the generated warnings by different SAST tools to the Seven Pernicious Kingdoms (SPK) classification [192]. The SPK classification provides a taxonomy of common types of security coding errors that might lead to vulnerabilities. This taxonomy is mainly based on the cause of a security vulnerability, but not necessarily the effect, and it focuses on implementation issues.

Each SAST tool has a list of warning types that they can detect in the documentation, along with a description for each of them. We analyze these lists and descriptions of warning types and manually classify each warning type into one of the seven types described by the SPK classification (see Table B in Appendix B for more detail). For each of the six SAST tools, we determine which tool can identify which type of vulnerability from the SPK classification. The results of that are presented in Table 3.2.

Table 3.2: Seven pernicious kingdom categories detected by each SAST tool

Tool	IVR	API	SF	TS	ERR	CQ	ENC	ENV
RATS	✓	✓	✓	✓		✓		
Flawfinder	✓	✓	✓	✓		✓		
Cppcheck	✓	✓		✓	✓	✓		
PVS-Studio	✓	✓		✓	✓	✓	✓	✓
Parasoft C/C++test	✓	✓	✓	✓	✓	✓	✓	✓
Clang Static Analyzer	✓	✓	✓	✓		✓		

3.4.5 Tracing Evolution of Warnings

Now that we have the warnings from each of the SAST tools for the 2012 and 2017 releases of the 116 case study projects, we can trace their evolution. We track each filename/line of code from the 2012 release into the 2017 release. We use the version history and *cregit*$^+$, to

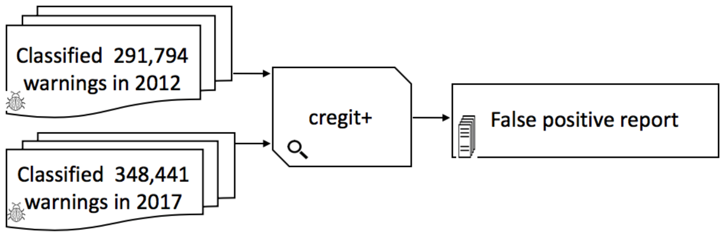

Classified 291,794 warnings in 2012
Classified 348,441 warnings in 2017
cregit+
False positive report

Figure 3.4: Tracing evolution of warnings using *cregit*$^+$

track the lines of code across the subsequent commits to 2017. When the 2012 line existed in 2017, we can identify if the warning is still present, has changed, or is a different one.

Therefore, by using *cregit*$^+$we can determine the exact line of code in the 2017 version of the project's source code that corresponds to the line of code under consideration from 2012 (as shown in Figure 3.4). We then check the warnings from 2017 to see if the determined line of code has the same warning, a different warning, or no warning. Note that *cregit*$^+$can come up with the result that the specific line of code from 2012 does not exist anymore (i.e., the line has been deleted). In this case, too, there will be no warning since the line of code does not exist. We classify each of these into the 4 cases, as discussed in Section 3.2.

Note that since we are tracing the source code across the various commits in GitHub, we only include warnings from the core project and not any third-party. Hence, we trace 213,627 warnings from 2012. All of our empirical data are available for download [43].

3.4.6 Extracting Warning Decay

In this step, we want to calculate the time it takes for a warning to disappear. We first include only warnings that disappear, but the lines of code have not been altered drastically, which may provide more concrete results. Hence, we compare the actual line of code in 2012 and the line of code in 2017 using *Levenshtein distance* and threshold of 0.8. Then, for each warning line, we extract the list of commits that touched that particular line. We do this using *git-blame*; for each line, we find the last commit that touched it and repeated this process recursively.

3.5 Empirical Study Results

3.5.1 RQ1: How are the warnings from each SAST tool distributed across different warning types in 2012 and 2017?

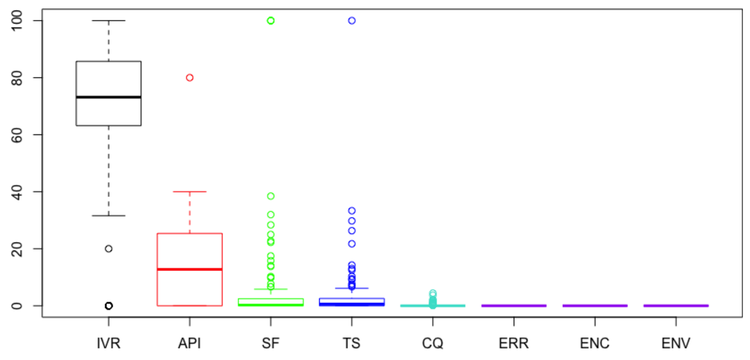
Approach: In this RQ, we analyzed 291,794 warnings from 2012 and 348,441 warnings from 2017 in 116 projects. For comparison, we represent warning distribution using a box-and-whisker plot, with each data point representing a project. We present the plots from 2012 and 2017 side-by-side to see if the trends are changing. We present two sets of plots in this RQ. In one set, we compare the warning types and their trends between 2012 and 2017 for each SAST tool. This helps us understand if the distribution of warning types in the same tool changes between 2012 and 2017. Then, we compare the warnings across SAST tools for each warning type between 2012 and 2017. In this analysis, we can see which tool finds how many warnings of a particular type and if that changes between 2012 and 2017. Thus, we are able to compare different SAST tools for the same warning type.

We also combine the box-and-whisker plot with the Scott-Knott Effect Size Difference (ESD) test to cluster different categories of warnings into statistically distinct ranks in terms of mean importance [186]. The Scott-Knott ESD test performs well, even when the dataset involves the overlapping problem (i.e., the probability of one or more warnings to be categorized in more than one group). This test performs multiple mean comparisons, as it separates means into statistically distinct groups with non-negligible differences.

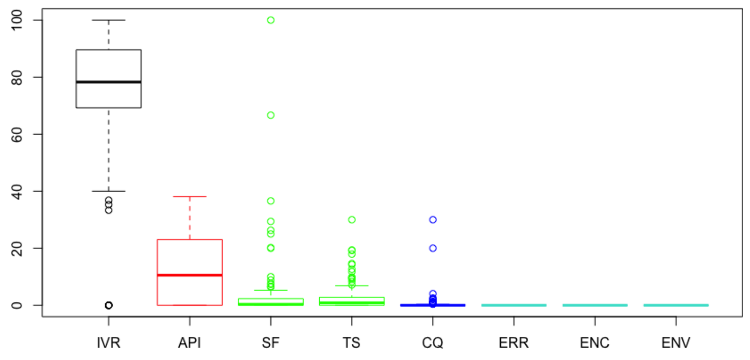
Results: The following are the results of the analysis.

1. *Warnings within SAST tools:* Figure 3.5a and 3.5b show the warnings distribution when using the RATS tool in 2012 and 2017, respectively. We include the results for the other SAST tools in Figure C.1 and Figure C.2 in Appendix C for readability purposes. The plots include the box-and-whisker plot and the Scott-Knott ESD test among warning types ordered from left to right by their decreasing mean values. In these plots, different colors represent distinct groups that differ significantly from the other, as determined by the Scott-Knott ESD test.

   From the plots, we can see that the distribution of warning among different categories is almost the same in 2012 and 2017 for the RATS tool. We observe the same phenomenon in all other tools as well (check Figure C.1 and Figure C.2 in Appendix C). Thus, the warning distributions are stable even after five years. We can infer from these plots that the quality of the projects from the perspective of the SAST tools is consistent. However, the different SAST tools have different vulnerability types that they detect.



(a) Bug warning categories in RATS at 2012



(b) Bug warning categories in RATS at 2017

Figure 3.5: Box-and-whisker plot and Scott-Knott ESD test of the number of warnings detected by RATS. In this figure, the y-axis represents the normalized number of bugs(bugs/LOC), and the x-axis represents different classes of SPK bugs. Each data point represents a project.

RATS (Figure 3.5a and 3.5b) and Flawfinder (Figure C.1c and C.1d) detect more *Input Validation and Representation* (IVR) warnings than any other type of warning in both 2012 and 2017. In Cppcheck (Figure C.1e and C.1f), the Clang Static Analyzer (Figure C.2a and C.2b), PVS-Studio (Figure C.2c and C.2d), and the Parasoft C/C++test (Figure C.2e and C.2f), we observe that *Indicator of Poor Code Quality* (CQ) warnings is the most significant type of warning that is being reported in both 2012 and 2017. Although, in all the latter tools IVR and *API Abuse* (API) warnings are the second and third most significant types of warnings being reported in both 2012 and 2017 (similar to RATS and Flawfinder).

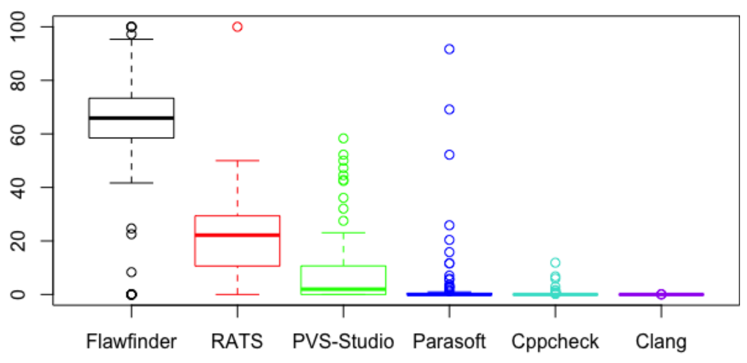
We can infer from the above result that the trend of reported bug warnings for each tool does not change over time. This might indicate that the quality of projects does not change over time, which is consistent with other research findings [81]. Another observation is that most of the studied SAST tools produce a high number of IVR, CQ, and API warnings in both 2012 and 2017.

Takeaway 1: The trend of reported bug warnings for each tool does not change over time. This might indicate that the quality of projects does not change over time.

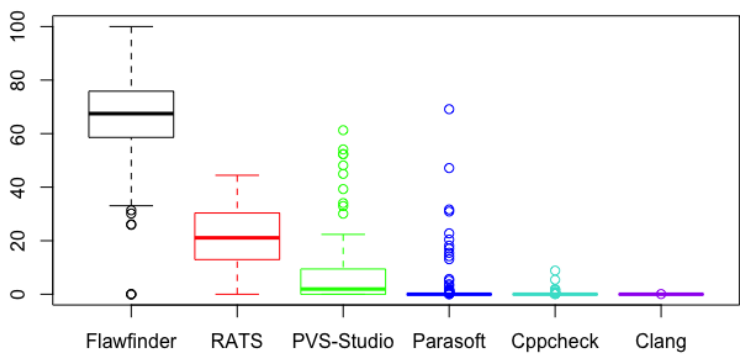
2. *Security warnings grouped by Seven Pernicious Kingdoms (SPK) classification:* In 3.6a and 3.6b we show the percentage of IVR warnings detected by the six different SAST tools in 2012 and 2017, respectively. We include the results for the other SPKs in Figure D.1 and Figure D.2 located in Appendix D. We notice that Flawfinder produces the highest number of IVR (Figure 3.6a and 3.6b), API (Figure D.1c and D.1d), and *Security Features* (SF) (Figure D.1e and D.1f) warnings and they have the highest median among projects in both 2012 and 2017.

   We also observe that *Time and State* (TS) results (Figure D.1g and D.1h) is less stable between 2012 and 2017. RATS produces TS more than other tools in 2102, while Flawfinder became the most significant tool that generates TS in 2017. We also note that PVS-Studio has generated the most warnings of CQ (Figure D.2a and D.2b), *Error Handling* (ERR) (Figure D.2c and D.2d), and *Environment* (ENV) (Figure D.2g and D.2h) for both 2012 and 2017. The Parasoft C/C++test is the only tool that produced *Insufficient Encapsulation* (ENC) type of warnings (Figure D.2e and D.2f), and it is negligible number warnings in both 2012 and 2017. The results denote that IVR, API, TS, and SF are more generated by tools that only use pattern-matching method, such as Flawfinder and RATS, while CQ is produced more by tools that use more advanced semantic analysis.

Takeaway 2: IVR, API, SF, and TS are more spotted by Flawfinder and RATS tools that use pattern-matching techniques, while CQ, ERR, ENC, and ENV are more reported by more advanced SAST tools.



(a) IVR warnings among SAST tools in 2012



(b) IVR warnings among SAST tools in 2017

Figure 3.6: Box-and-whisker plot and Scott-Knott ESD test of the number of warnings classified as Input Validation and Representation (IVR). In this figure, the y-axis represents the normalized number of bugs(bugs/LOC), and the x-axis represents different SAST tools. Each data point represents a project.

3.5.2 RQ2: How are true positive and false positive warnings distributed across different warning types and across different SAST tools?

Approach: We conduct our study on 213,627 warnings from the 116 projects and their versions in 2012 (which is less than the 291,794 warnings discussed in RQ1 because we only consider the core projects here and not include the dependencies like in RQ1). To answer this RQ, we conducted the following analysis:

- We show the *Remained Warnings*, *Modified Warnings*, and *Disappeared Warnings* distributions across warning types for each SAST tool and across SAST tools for each warning type, using both box-and-whisker plot and the Scott-Knott ESD test.

- We measure the likelihood of a warning that belongs to a specific warning type to be a real warning for different SAST tools using the Odds Ratio (OR) similar to Di Penta et al., [75]. The OR is the ratio of the odds of an event occurring in one set (in this research, the eliminated warning subset (TP including *Modified Warnings* and *Disappeared Warnings*) to the odds of it occurring in another set (in this research, the alive warning subset FP including *Remained Warnings*). An odds ratio of 1 means that the detected warnings could belong to FP or TP equally likely. An odds ratio higher than one means that the detected warnings are more likely to be TP, and an odds ratio less than one means that detected warnings are more likely to be FP.

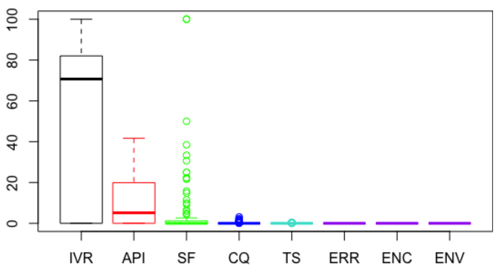
- We apply a proportion test similar to Di Penta et al., [75] that shows whether the proportion of eliminated warnings differ across warning types (H0: there is no difference among proportions of eliminated warnings). This could provide a better understanding of whether some types are receiving more attention than others.

Note that, in our analysis, we only consider *Modified Warnings*, *Disappeared Warnings*, and *Remained Warnings*. We ignore *Removed Warnings* (one where the entire line of code from 2012 no longer exists in 2017), as this type of warning should be considered with caution since it is hard to assume that the removal of lines of code that include warnings indicates a bug fix. The line removal could be due to other reasons, such as code refactoring or deleting features.

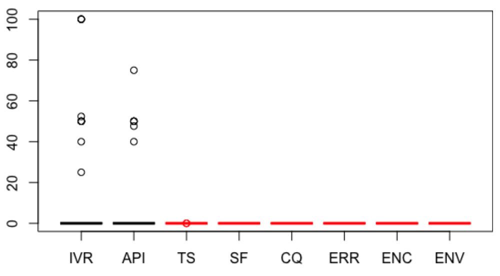
Results: The following are the results of the analysis.

1. *Analysis of Warnings among SAST tools:* This analysis is to compare the FP and TP rates per warning types among SAST tools. Figure 3.7a, 3.7b, and 3.7c depict RATS to show *Remained Warnings*, *Modified Warnings*, and *Disappeared Warnings* comparisons (other plots for other tools are located in Figure E.1 and Figure E.2 in Appendix E). We observe that the *Remained Warnings* plots have a similar pattern to the *Disappeared Warnings* plot for each tool. RATS (Figure 3.7a, 3.7b, and 3.7c) and Flawfinder (Figure E.1d, E.1e, and E.1f) show that the most significant groups of *Remained Warnings* and *Disappeared Warnings* are IVR, API, and SF, respectively. In Cppcheck (Figure E.1g, E.1h, and E.1i), PVS-Studio (Figure E.2d, E.2e, and E.2f), the Clang Static Analyzer (Figure E.2a, E.2b, and E.2c), and the Parasoft C/C++test(Figure E.2g, E.2h, and E.2i), we note that CQ is the most types of

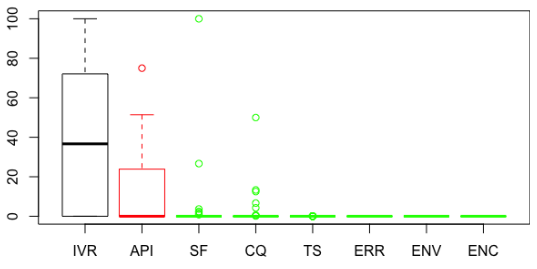
warnings that appear as *Remained Warnings* and *Disappeared Warnings*. It is remarkable to see that in RATS, Flawfinder, and the Parasoft C/C++test have some warnings in *Modified Warnings* plots, which refers to the warnings that likely have been resolved but there were other warnings have been introduced. An example of that is when developers try to replace possibly insecure function *strcpy* by another more secure but still vulnerable function *strncpy*. Hence, we infer from the above results that there is a strong correlation between the number of FP and TP for each type of warning in each tool. For instance, the most frequent type of false positive bug warnings produced by RATS tool is IVR, and also IVR is the most frequent type of bug warnings that disappeared. This means that the number of false positive could be used as an indication of the potential real bugs could be detected by each tool.



(a) *Remained Warnings* in RATS



(b) *Modified Warnings* in RATS



(c) *Disappeared Warnings* in RATS

Figure 3.7: Box-and-whisker plot and Scott-Knott ESD test of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different warning types by RATS. In this figure, the y-axis represents normalized number of bugs(bugs/LOC), and the x-axis represents different classes of SPK bugs. Each data point represents a project.

Takeaway 3: For all the SAST tools used in this study, for each type of warning there is a strong correlation between the number of true positive and the number of false positive warnings found.

2. *Analysis of Warnings among SPKs:* Another analysis we performed is to compare each type of warnings among different SAST tools. Figure 3.8a 3.8b, and 3.8c display IVR warnings to show *Remained Warnings*, *Modified Warnings*, and *Disappeared Warnings* rate comparisons (while other plots for other tools are located in Figure F.1 and Figure F.2 in Appendix F). In IVR (Figure 3.8a 3.8b, and 3.8c), API (Figure F.1d, F.1e, and F.1f), and SF (Figure F.1g, F.1h, and F.1i) plots, we see that Flawfinder produces a significant amount of *Remained Warnings*; in addition, it has the most significant number of warnings detected correctly in *Disappeared Warnings*. RATS is the most tool that produces TS as both *Remained Warnings* and *Disappeared Warnings* (Figure F.1j, F.1k, and F.1l), while PVS-Studio is the most tools that produces CQ (Figure F.2a, F.2b, and F.2c) and ERR (Figure F.2d, F.2e, and F.2f) as *Remained Warnings* and *Disappeared Warnings*. We note that ENC (Figure F.2g, F.2h, and F.2i) are produced only by the Parasoft C/C++test, but it is likely to be *Remained Warnings* that are FP warnings. Similarly, the results indicate that a tool that generates the highest number of false positive of a specific warning type compared to other tools, it would be the tool that the most disappeared warnings of the same type belong to. For instance, Flawfinder generates the most false positive IVR bug warnings compared to other tools, also the disappeared warnings of the IVR mostly belong to Flawfinder. This also could be used as an assessment to measure the potential real bugs of a certain type could be detected by such SAST tool.

Takeaway 4: Pattern-matching based tools, such as RATS and Flawfinder, produce a larger number of warnings (both FP and TP) of IVR, API, SF, and TS, while more advance tools, such as PVS-Studio and Parasoft C/C++test, generate a larger number of CQ warnings (both FP and TP).

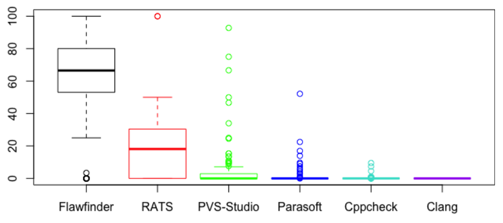
3. *Odds ratio analysis:* Table 3.3 reports all SAST-produced warnings in terms of:

   - the total number of warnings reported by each tool split by warning type in 2012.
   - the number of warnings that exist in the same line of code in both the 2012 and 2017 versions (*Remained Warnings*).

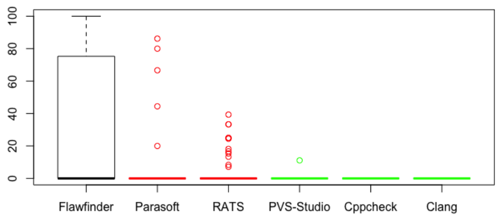
- the number of warnings from 2012, which disappear in 2017, but have a different warning in the same line of code as compared to 2012 (*Modified Warnings*).
- the number of warnings from 2012 that no longer exist in the same line of code, but the line of code still exists, albeit modified (*Disappeared Warnings*).
- the odds ratio of a tool finding a warning in 2012.

RATS, Flawfinder, Cppcheck, and Clang Static Analyzer have Odds Ratio (OR) less than one for all warning types, indicating that these types of warnings are likely to be FP.

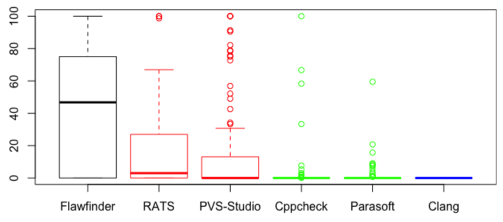
Conversely, PVS-Studio results denote that most reported warnings in 2012 disappeared in 2017; hence, they are more likely to be real warnings. Only ENV type of warnings is the most likely to be FP. Parasoft C/C++test is likely to be accurate with SF warnings. We note that it has (OR=16), which is a very high odds ratio to be a real warning compared to other types of warnings. Similarly, the ENV type of warnings show (OR=3.062) indicates a higher probability of having real warnings



(a) *Remained Warnings* among IVR



(b) *Modified Warnings* among IVR



(c) *Disappeared Warnings* among IVR

Figure 3.8: Box-and-whisker plot and Scott-Knott ESD test of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different warning types among Input Validation and Representation (IVR). In this figure, the y-axis represents the normalized number of bug(bugs/LOC), and the x-axis represents different SAST tools. Each data point represents a project.

reported. That is because Parasoft C/C++test enforces secure coding standards, such as MISRA, CERT, and ISO. Also, Parasoft C/C++test detects Environment errors with a high precision rate because it targets cross-compiler platforms.

From the result above, we note that tools that use pattern-matching along with other advanced analysis methods, such as Parasoft C/C++test and PVS-Studio, outperform other tools in terms of the likelihood of producing real security bugs.

Takeaway 5: SAST tools in this study that use the pattern-matching method along with semantic analysis are more likely to produce warnings that may be real warnings.

4. *Proportion test analysis:* For warnings detected with RATS and Flawfinder, the proportion test of likely resolved warnings significantly varies across types (p-value < 0.0001), meaning that some types are receiving more attention and likely being resolved than others. A small p-value (typically ≤ 0.05) indicates strong evidence against the null hypothesis, so we could reject the null hypothesis, which states that there is no difference among proportions. For warning detected with Cppcheck, the proportion of likely fixed warnings slightly varies across types (p-value $= 0.0025$), which denotes that there is a difference in the attention paid to different warning types detected by this tool. When looking at PVS-Studio results, we observe that the proportion test of likely fixed warnings significantly varies across types (p-value $=$ 4.008e-10). Finally, when glancing at the Parasoft C/C++test, we find that the proportion test of likely fixed warnings (TP) varies slightly across types (p-value $=$ 0.0002). This denotes that there is a difference in the attention paid to different warning types detected by this tool. Thus, we conclude that for each studied SAST tool, there are some types of warnings that receive more attention from the project teams to be eliminated, while other types of warnings are likely ignored, suggesting that they are not critical bugs.

Takeaway 6: For every SAST tool, some types of warnings were more likely to be eliminated than other types. Hence, it is very likely that some types of software security flaws receive more attention from developers.

Table 3.3: Numbers of all detected FP (which includes *Remained Warnings*), TP (which includes *Modified Warnings* and *Disappeared Warnings*), and Odds Ratio of with respect to FP.

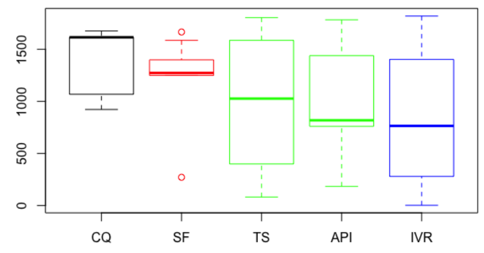
Tool	Type	Total	Remained Warning	Modified Warning	Disappeared Warning	TP	TP+FP	Removed Warning	OR
RATS	IVR	37,776	26,438(69.99%)	84(0.22%)	3,681(9.74%)	3,765	30,203	7,573	0.020
	API	10,979	6,770(61.66%)	80(0.73%)	1,772(16.14%)	1,852	8,622	2,357	0.074
	SF	592	409(69.09%)	0	46(7.77%)	46	455	137	0.012
	CQ	157	104(66.24%)	0	25(15.92%)	25	129	28	0.057
	TS	1,246	827 (66.37%)	3(0.24%)	140(11.24%)	143	970	276	0.029
Flawfinder	IVR	92,998	68,126(73.26%)	320(0.34%)	5,992(6.44%)	6,312	74,438	18,560	0.008
	API	35,314	24,491(69.35%)	234(0.66%)	2,731(7.73%)	2,965	27,456	7,858	0.014
	SF	1,157	882(76.23%)	4(0.35%)	48(4.15%)	52	934	223	0.003
	CQ	3,250	2,801(86.18%)	8(0.25%)	115(3.54%)	123	2,924	326	0.001
	TS	2,168	1,315(60.65%)	5(0.23%)	144(6.64%)	149	1,464	704	0.012
Cppcheck	IVR	274	144(52.55%)	0	41(14.96%)	41	185	89	0.081
	API	207	115(55.56%)	0	30(14.49%)	30	145	62	0.068
	CQ	1,563	850(54.38%)	1	385(24.63%)	386	1,236	327	0.206
Clang Static Analyzer	IVR	1	0	0	0	0	0	1	-
	CQ	31	14(45.16%)	0	5(16.13%)	5	19	12	0.127
PVS-Studio	IVR	7,154	2,408(33.66%)	1(0.01%)	3,461(48.38%)	3,462	5,870	1,284	2.067
	API	1,656	635(38.35%)	0	679(41.00%)	679	1,314	342	1.143
	CQ	16,808	6,295(37.45%)	9(0.05%)	7,464(44.41%)	7,473	13,768	3,040	1.409
	ERR	208	50(24.04%)	0	83(39.90%)	83	133	75	2.755
	ENV	29	16(55.17%)	0	11(37.93%)	11	27	2	0.472
Parasoft C/C++test	IVR	1,835	961(52.4%)	152(8.3%)	658(35.9%)	810	1,771	64	0.710
	API	1,792	1,010(56.4%)	158(8.8%)	554(30.9%)	712	1,722	70	0.496
	SF	5	1(20.0%)	0	4(80.0%)	4	5	0	16
	CQ	78,322	42,541(54.3%)	1,790(2.3%)	32,764(41.8%)	34,554	77,095	1,227	0.659
	TS	156	101(64.7%)	18(12.8%)	32(19.2%)	50	151	5	0.245
	ENC	2	2(100%)	0	0	0	2	0	-
	ENV	39	12(30.8%)	0	21(53.8%)	21	33	6	3.062

3.5.3 RQ3: How do different types of warnings evolve over time?

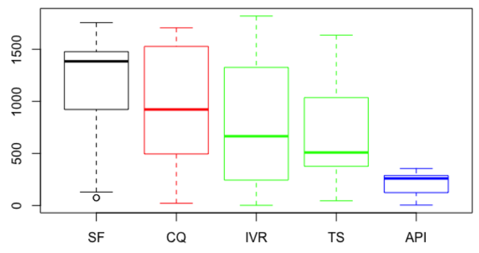
Approach: In this RQ, we focus on TP warnings (*Modified Warnings* and *Disappeared Warnings*) that were detected by SAST tools and eliminated in the code. We only consider lines of code that include warnings and have been altered, as this gives us the ability to trace the changes in the lines of code. Whenever we find a warning that was eliminated and belong to a line of code that was altered in the recent version, we trace the line of code that contains the bug warning backward during the history to find the fix time and flag that to be a *bug fix* version. To answer this question, we use *git blame*. This study involves

10,478 warnings that have been eliminated by altering the lines of code. We present the results using box-and-whisker plots. In these plots, each data point is how long it took for a specific warning in a specific line of code from one of the 116 projects to disappear.

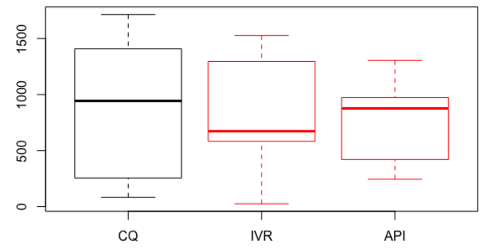
Results: The following are the results of the analysis.



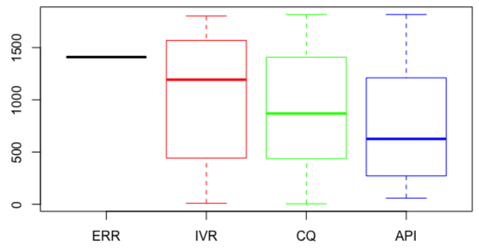
(a) Decay in RATS



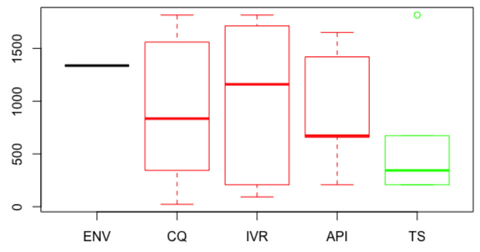
(b) Decay in Flawfinder



(c) Decay in Cppcheck



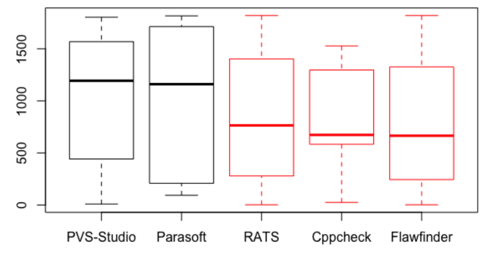
(d) Decay in PVS-Studio



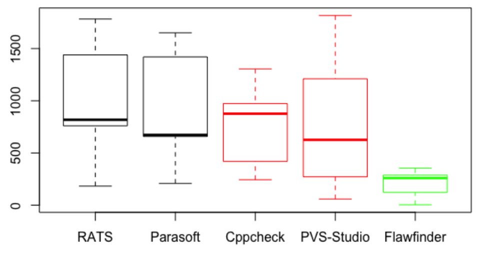
(e) Decay in the Parasoft C/C++test

Figure 3.9: Box-and-whisker plot of decays for various warning types in different SAST tools. In this figure, the y-axis represents decays in days, and the x-axis represents different classes of SPK warnings. Each data point represents a resolved bug warning time.

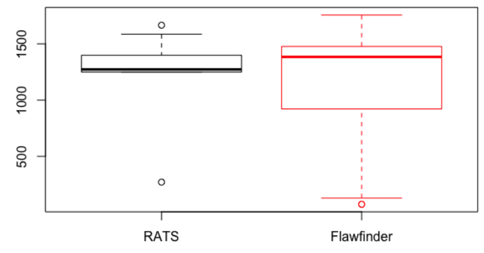
1. *Warnings decay among SAST tools:* We analyze the set of *Disappeared Warnings* and *Modified Warnings* that present in 2012, and the lines of code that were after



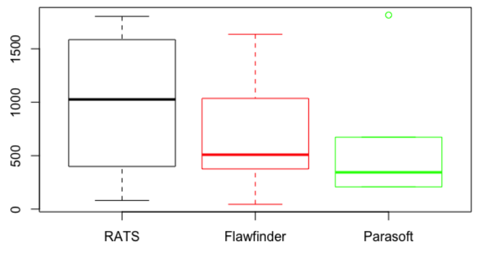
(a) Decay of IVR



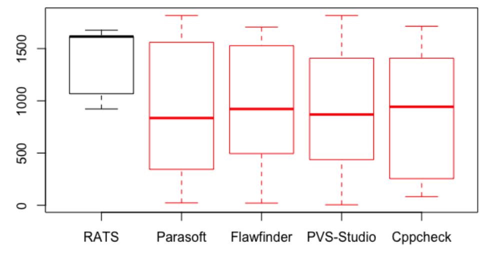
(b) Decay of API



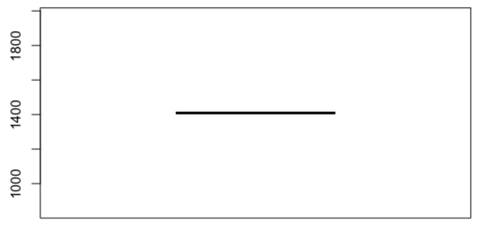
(c) Decay of SF



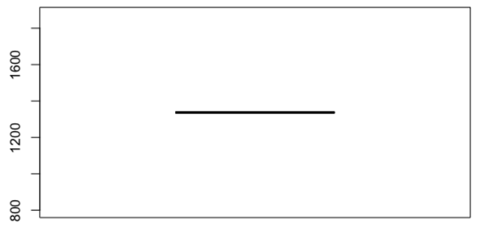
(d) Decay of TS



(e) Decay of CQ



(f) Decay of ERR



(g) Decay of ENV

Figure 3.10: Box-and-whisker plot of decays for various warning types in different warning types. In this figure, the y-axis represents decays in days, and the x-axis represents different SAST tools. Each data point represents a resolved bug warning time.

being altered, the warning disappeared. This helps us to measure the decay. Figure 3.9 shows the decays (expressed in days) for each warning type detected in the projects with different SAST tools. When looking at the plots, we observe that most of the warnings that belong to different types are eliminated approximately within 2-3 years. We also observe that in the Flawfinder plot, the median of API decay is significantly lower than the other warning types (approximately less than one year), suggesting that there is a tendency to eliminate this type of warnings faster than others. Generally, we observe that API warnings disappeared relatively faster than other types among tools. Additionally, the Parasoft C/C++test plot shows that TS warnings are eliminated relatively fast, indicating that this type of warnings has less removal time, which may indicate that this type of warnings is more critical than other types.

Takeaway 7: Most of the bug warnings that belong to different types disappeared approximately within 2-3 years. Generally, API warnings disappeared relatively faster than other types among tools.

2. *Warnings decay time among SPKs:*

   Here, we are interested in knowing if a particular type of bug warning could be eliminated faster if detected by a particular SAST tool. This allows us to know if the underlying analysis method influences the removal time of a bug warning of a specified category. Note that tools that only use the pattern-matching method refer to an issue that is seen clearly within the same line of code, while advanced tools could comprehend errors that are scattered on multiple lines and thus may be harder to understand. Figure 3.10 shows the results of such an analysis. The results imply that for IVR and CQ, the removal time is almost the same among all tools. It is interesting to note that IVR removal time for tools that only use the pattern-matching analysis is slightly less than other tools that use more advanced techniques. This could be because these tools generated trivial bug warnings that are easier to understand and resolve. We also perceive that generated API warnings by Flawfinder decay less than other API warnings generated by other tools (in less than one year). Furthermore, we see that TS generated by the Parasoft C/C++test decay in approximately one year. Another remarkable observation is that SF type of warning usually takes over three years to be eliminated, which is longer than any other warning type detected by most analysis tools. We note that there is no connection between decay time and different SAST tools that have different analysis methods. Hence, the nature of the bug warning may not be a factor that influences removal time.

Takeaway 8: There is no correlation between decay time and different SAST tools that have different analysis methods.

3.6 Discussion

3.6.1 SAST Warning Patterns and Real-world Vulnerabilities

When observing the results from our analysis, we notice that SAST-produced warnings have stable patterns through time (check RQ1 in Section 3.5). This may indicate that the quality of the project's development does not change over time. Further, we find that most of the SAST tools focus on IVR, API, and CQ types of warnings. This may denote that these types of warnings are critical and could be the types of security vulnerabilities in the real-world. Real-world vulnerabilities are exploitable software bugs discovered and published in recognized vulnerability databases with unique identification numbers. To verify this, we study the distribution of real-world vulnerabilities and associated vulnerability classifications in the studied projects that the project developers fixed to see if this aligned with our findings. So, we conduct the following analysis:

3.6.1.1 Identifying Incidence of Real-World Vulnerabilities

We choose to study vulnerabilities with a Common Vulnerability and Exposures Identification Number (CVE-ID) in the 116 projects in our dataset. The CVE-ID are unique numbers assigned to publicly known security vulnerabilities. In this step, we extract all commit messages along with the description of each commit, looking for the keyword "CVE-xxxx-xxxx" between 2012 and 2017. We find that of the 1174 times a CVE-ID was mentioned in 18 distinct projects, 395 have a unique ID. This helps us to identify how real-world vulnerabilities are distributed across various types of vulnerabilities.

3.6.1.2 Results of Real-World Vulnerabilities Distribution

Figure 3.11 displays the results of real-world vulnerabilities that have been fixed in the projects under the study. We can see that the IVR vulnerabilities are the predominant

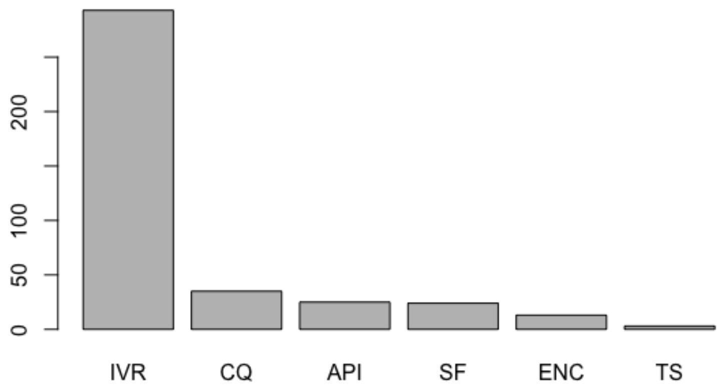


Figure 3.11: CVE vulnerabilities distribution based on the SPK taxonomy. In this figure, the y-axis represents the number of unique vulnerabilities and the x-axis represents different classes of SPK classification of bugs.

type of vulnerability that has been detected and fixed in real-world scenarios. The number of IVR vulnerabilities is 293. The other types of vulnerabilities that are given importance after IVR are CQ, API, and SF, respectively.

The results show that IVR is the most significant type of warnings being addressed in real-world vulnerabilities. Most of the studied tools produce a high number of IVR warnings. This may refer to the importance of this type of warning and the importance of enhancing the static analysis methods to detect IVR accurately. Also, CQ and API issues were being flagged in the real-world, similarly to the output of SAST tools. Generally, we conclude that SAST tool warnings follow a similar distribution of the discovered real-world vulnerabilities in the studied projects, which is a finding that is consistent with other research [81].

3.6.2 Robustness of Static Analysis Method

The results of RQ2 in Section 3.5 highlight the odds of each type of warning of being true or false alarms for each tool. The results indicate that PVS-Studio and the Parasoft C/C++test produce warnings that are likely to be real warnings. PVS-Studio has a higher chance of producing real IVR, API, CQ, and ERR, while the Parasoft C/C++test has higher chances to produce real SF and ENV. These two tools are the only tools that combine pattern-matching with more advanced semantic analysis (see Table 3.1 for more details). Pattern-matching by it is own is a powerful method that can detect many real bugs. That is notable since Flawfinder and RATS outperform PVS-Studio in finding more IVR warnings, and Flawfinder outperforms PVS-studio in finding more API warnings in terms of the frequency/quantity. This denotes the value of using the pattern-matching method since pattern-matching tools (Flawfinder and RATS) could detect more real bugs (quantitatively not qualitatively). For example, Flawfinder detected 5992 of Input validation and representation bugs while PVS-Studio 3461 bugs of the same type. Nevertheless, Flawfinder has around 6.5% accuracy while the PVS-Studio rate is 48%, which is better qualitatively. However, when combining pattern-matching with other analysis techniques, such as symbolic execution or abstract interpretation, the warning's accuracy increases. Hence, we think that pattern-matching is a robust technique, but it needs to be combined with a more advanced analysis method to filter out possible false positives. Also, ENV warnings are likely to be accurately flagged by the Parasoft C/C++test than PVS-Studio. This is because that the Parasoft C/C++test is highly focused on highlighting software environmental compliance.

3.6.3 SPK Warnings Decay Time

In RQ3 Section 3.5, we observe that of those warnings that are removed, the decay time is around 2-3 years. This result corresponds to other research findings, such as [152]. Ozment and Schechter [152] study the vulnerabilities of the OpenBSD operating system to investigate whether the software vulnerabilities are increasing over time. The authors measured the rate of reported vulnerabilities over a period of seven and a half years that span 15 releases. The study showed that vulnerabilities seem to be persistent even for a period of 2.6 years. Also, among studied tools, we find that generally, API has decay somehow faster compared to other types of warnings. This may imply the significance of such a warning that may have serious consequences. This could help give recommendations to developers about which bugs should be fixed first. Another possible reason is that this is because these types of bugs are easier to understand and fix since the API issue is likely to manifest in the same line of code compared to other types of bugs that could be caused by a different line of code. For instance, a buffer overflow warning that belongs to the IVR class of bug could happen when a condition does not validate the size of the buffer located in a different line of code. However, this is less likely since we have a limitation in RQ3 that we only examine warnings that disappear because the line of code has been modified.

3.7 Lessons Learned

In this section, we provide insights about how the findings can improve state-of-the-art and the state of the practice for developers, security testers, SAST tool designers, and researchers.

- **Software developers:** Building secure software in the first place is key to prevent security bugs from ever occurring. This requires solid knowledge about secure coding practice and building practical skills that can be learned and built from real-world problems. Although developers are expected to write secure code, around 70% of developers said they get little guidance or assistance according to a survey [188]. This study helps developers to be aware of common security bugs that are regularly addressed. The study shows that IVR issues are the most types of vulnerabilities that have been fixed in real-world projects. This work also demonstrates that most of the SAST tools focus on IVR, CQ, and API types of issues. This may indicate that these types of warnings are critical. Hence, it would be advisable for developers to build related secure coding practices and be aware of that during coding and take the time to ensure the code is free from these types of bugs.

  Additionally, SAST tools could generate many warnings that could overwhelm the most experienced software developers. This study shows that some types of warnings are not likely to be removed; hence, they tend to be false positives. A code that seems to be reliable and working well contains warnings, hence aiming to remove all warnings might not be cost effective.

  Furthermore, one of the implications of this study is that developers need to pay closer attention to the SAST tool's appropriate underlying design to detect a specific type of bug. For instance, for developers who develop web applications and interested in detecting IVR type of bug, it would be better to use a tool that uses both pattern-matching and symbolic execution, which was shown in this work to have a higher chance that generated warnings that are likely to be a real vulnerability.

- **Software security testers:** Software testers are responsible for investigating and evaluating the security and quality of software components. Running SAST tools may produce many warnings for a software component, and developers do not care to remove them all. Therefore, it is essential for software testers to be able to prioritize bug warning categories into importance, so when software testers scan the code, they should look for those that are more likely to create bugs. Further, software testers usually need to provide early estimates and an objective view of the software reliability or fault-proneness to help code inspections and testing and understanding the risks of software implementation. The empirical evidence in this research (RQ2) shows that using SAST tools might be a suitable method to measure the project's quality despite the false positive rates. Therefore, SAST tool warnings can predict the security risk of the products even without a manual review of the warnings. Hence, false warnings are not completely unbeneficial since we observe that the more false positive warnings of a particular type by a SAST tool, the more real bugs being fixed (see RQ2 in Section 3.5). This study also demonstrates that security bugs seem to be persistent even for a period of 2.6 years, which is consistent with previous research studies [152]. This means that when assessing a bug report, it should be taken into consideration that some critical bug's lifetime can be long and not to neglect it for that reason.

- **SAST tool designers:** The take-home message for SAST tool designers is that not all types of warnings appear to be critical to developers. This thesis shows that some types of warnings are never removed, then the challenge is to identify or rank warnings according to the likelihood that they will be a bug. While this is ideal, the work herein provides an easier-to-implement alternative: rank the importance of warnings based on whether they will be removed in the future. While this is imperfect, this might be better than no ranking at all. Moreover, this study finds in RQ2 that pattern-matching is a robust technique, but it needs to be combined with a more advanced analysis method to filter out any possible false positives. We also noticed that abstract interpretation could be an optimal approach to design tools that focus on Security Features (FS) security bugs, such as privacy violations.

  Another point to be highlighted is that by observing that API type of bugs decays faster than other types and underlying analysis techniques does not influence that. We think that these types of bugs were fixed faster because once identified, it was easier to understand and hence fixed. Previous studies have shown that the difficulty of understanding a bug warning may lead to neglecting it [112]. Unlike API bugs that could manifest on one line of code, other types of bug warnings can include issues scattered across multiple lines of code. Hence, bug warnings need to be represented well to developers, for instance, instead of showing a line of code that includes a bug and explaining that abstractly. Tool designers need to illustrate the set of lines of code that involve the problem.

- **Researchers:** This study illustrates that some types of warnings are more likely to disappear. Hence, research is needed into trying to understand if this is true because either: developers concentrate on these types of errors (they are low hanging fruit during code reviews and other when they edit the code around) or because they indeed create bugs and the warning indeed show the cause of the problem. Another research direction worthy of investigation is comprehending the reasons for the short-lifetime bug warnings and incorrect fixes, and buggy patches. Also, researchers could benefit from this research as they can continue this work and try to improve it by tracking more data/better data, surveys, more systems.

3.8 Threats to Validity

Threats to construct validity: In this study, we are not assuming that developers have run the SAST tools in this study. We only measure the issue with the source code that could be flagged by SAST tools and measure the FP and TP rate by observing the change in the line of code. We make the following assumptions: (a) If a warning exists in the released version of the project in 2012 and 2017, then that warning is a false positive warning. This is a fairly reasonable assumption since any warnings present in the released code for five years are likely not to affect the quality of the product. Hence, they are FP. (b) If a warning no longer exists in the 2017 version, we assume that it is a TP. We do not know why the developer resolved the warning, but just that the warning no longer exists. Irrespective of the reason, since the developers chose to take some action that resulted in the warning to go away, we assume that this is important. However, the developers may not remove the warnings consciously. Therefore, this could be a problem. One action we took to address this is that we do not discuss *Removed Warnings* (when a line is deleted) in this study. We believe that removing lines could be because of removing a feature. While editing a line is most likely to fix a bug or for maintenance purposes. Hence, we only discuss *Modified Warnings* and *Disappeared Warnings*.

Additionally, we manually reviewed some bug warnings to mitigate the threats to validity of our results regarding 1) the false positives and true positives rates of the same warning types that vary across tools, 2) decay time of bug removal of the same warning types that differ across tools. Different tools produce different outputs in terms of false positive and true positive rates for the same warning type. This is because different tools have different underlying analysis techniques (as discussed in Section 2.2.9). Hence, IVR warnings generated by RATS, which is based on pattern-matching, differ from IVR warnings generated by PVS-Studio, which is based on pattern-matching and symbolic execution. However, our results show that the underlying analysis method does not appear to influence how fast the bug is being fixed. Hence, to ensure that our results are concrete, we manually looked at a few cases of Modified Warnings and Disappeared Warnings to see if the developers indeed removed the warnings. We manually analyzed 40 warnings by reviewing the commit notes and source code, and we found that 28 cases of the fixed commits indicate bug fixing tasks. 11 commits out of 28 indicate that the bugs were detected by SAST tools, such as PVS-Studio and Cppchek, among other tools. For instance, a warning that belongs to InsightSoftwareConsortium/ITK project has disappeared; the fixing commit a3793658d2 (i.e., the commit that removed the warnings as identified in RQ3) shows this commit message "COMP: Fix all the valid Cppcheck warnings in ITK (last patch)." At the same time, other commits do not have enough information about bugs being fixed. In this case, we notice that the source code includes many more critical changes in those versions, such as adding features that are project dependent. This issue was observed in Kim and Ernst work [116] (check Section 2.2.3) when the researchers found that bugs that were fixed in fix-changes represent a tiny percentage of bug removals, and about 90% of the bug warnings either remain in the program or are fixed in non-fix changes.

Threats to internal validity concern factors that could have affected our findings. We ensure that the selected projects do not introduce any biased results to our study. We study 116 popular C++ projects, yet they vary in terms of complexity, size, and activeness. When we mined the chosen repositories from GitHub, we enforce some criteria to ensure our dataset's robustness. For instance, we built a script to ensure that retrieved projects are C++ projects and not falsely tagged in GitHub as C++. Another threat is that we chose an analysis time frame of five years (2012-2017). From past research, we know that the typical time to fix a bug is around three years [63][75]. Hence, we considered a five-year time frame so that we can be sure that if a warning existed in both versions, then it has a life span that is at least five years. Finally, we provide all our tools and data for replication [43], and the methodology of this study is described in detail in Section 3.4.

Threats to external validity This study was conducted on 116 different open source projects and six different SAST tools. Nevertheless, analyzing further projects, different programming languages, and other SAST that have different analysis techniques are desirable.

3.9 Conclusion

This work provides an empirical study to the warnings generated by SAST tools and investigates the history of these buggy lines of code, using 116 large C++ repositories. We find that the patterns of the warnings are stable through time for all studied SAST tools. Also, our results show that most of the tools produce on IVR, API, and CQ warnings. These types of warnings are detected in the real world with comparable patterns, but IVR is detected significantly in the real world compared to others. Also, we observe that there is a correlation between the number of bugs that belong to different bug types generated by such a tool and the likely real bugs that were resolved. This indicates that SAST tools could be used as an assessment tool to measure the quality of a product and the potential risks without manually review the warnings. Also, this study shows the power of using the pattern-matching algorithm along with other advanced analysis methods. The outcome of this research may have multiple possible future directions, such as bug warnings prioritizing. Knowing the relation between static analysis methods and false alarm could help design better SAST tools to minimize false alarm rates. Moreover, it provides an insight into which static analysis algorithm is appropriate for different classes of warnings.

Chapter 4

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

Modern mobile apps incorporate rich and complex features, opening the doors for different security concerns. Android is the dominant platform in mobile app markets, and enhancing the security of its apps is an essential 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, this chapter aims at assessing SAST tools that could be used by Android developers. The main goal of this chapter is to identify whether state-of-the-art SAST tools can find real-world vulnerabilities.

Related Publications

The work described in this chapter has been previously published:

- Aloraini, Bushra, and Meiyappan Nagappan. "Evaluating state-of-the-art free and open-source SAST tools against buffer errors in android apps." 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017.

4.1 Introduction

Android dominates the mobile market as it has gained tremendous popularity recently [110]. The primary Android market, Google Play Store, included three million apps and at least 65 billion downloads as of June 2017 [177]. Users assume that app markets guarantee the security of offered apps [144]. Nevertheless, app markets typically do not adequately ensure the security of the apps they offer [153]. 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 malware in disguise [183]. However, 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 the availability of systems and data, causing substantial economic loss. Another report [33] showed that nearly 75% of tested mobile apps showed at least one critical or high-severity security vulnerability.

Studying Android vulnerabilities is, therefore, a critical area of research. SAST tools are a common proactive method to find security vulnerabilities in source code early during the coding phase [146]. This leads to cost savings, which is a key benefit of SAST tools, as the earlier a vulnerability is detected, the cheaper it is to fix [174]. Applying a static analysis approach to Android apps to identify vulnerabilities could potentially ensure the quality and reliability of the apps and hence the app market. Thus, SAST tools are standard in both the maintenance and evolution of software.

However, we do not know how effective SAST tools are in detecting vulnerabilities in Android apps. This chapter, thus, aims to study state-of-the-art SAST tools that can be used to detect Android vulnerabilities during the coding process.

The contributions of this chapter are as follows:

- We first measure the frequency of the software vulnerabilities in Android apps, and we determine the most dominant type of vulnerabilities in such apps as buffer errors.
- We evaluate state-of-the-art SAST tools to detect buffer errors regarding the real-world vulnerabilities happening in Android.
- We describe the characteristic of buffer error vulnerabilities observed in Android apps.
- We describe the required features of such SAST to detect such a pattern of vulnerability.

4.2 Research Questions

In this chapter, we ask the following RQs.

Motivational RQ: What are the most common vulnerabilities in Android apps? From data in the National Vulnerability Database (NVD) [10], we find that buffer errors are the most frequent type of vulnerability in Android apps. They also have the highest risk that compromises the integrity, confidentiality, and availability.

Case study RQ: Are state-of-the-art SAST tools for buffer errors able to detect vulnerabilities reported in the wild for Android apps? In this study, we investigate 17 SAST tools that could discover buffer errors. Out of the 17, we test six free and open-source SAST tools on nine real-world buffer errors, and we find 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 [44].

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

4.3.1 Motivation

Since examining SAST tools' efficiency on all types of vulnerabilities is beyond the scope of a single study, we want to find the most frequent type of vulnerability 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, CERT [1], and Bugtraq (BID) [19]. These databases utilize the Common Vulnerabilities and Exposures Identifier (CVE-ID) [3], which is a unique number for a publicly recognized security vulnerability, as an identifier of a vulnerability record. The NVD database is synchronized with CVE-ID, and its records are mainly based on CVE information. Thus, the NVD database is preferred by researchers as it is the most comprehensive database. Therefore, we select the NVD database to construct the dataset for this RQ.

4.3.2 Methodology

4.3.2.1 Data Gathering

In this phase, we extract Android's relevant vulnerability records from 2008 (when Android was first released) to 2015 from NVD using an automated web-scraping tool. We use the "Android" keyword 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. We extract 2,089 records in the initial data gathering.

4.3.2.2 Removing Irrelevant Data

We clean the dataset in order to get more accurate results. We manually examine all the 2,089 entries to check whether they are accurately related to Android; if not, the record is excluded from the dataset. We find four records unrelated to Android: CVE-2015-3906, CVE-2015-3815, CVE-2012-1344, and CVE-2011-1001.

4.3.2.3 Data Processing

In this phase, we trace each vulnerability record manually to collect more information, such as whether the vulnerability was confirmed or patched, and how it was discovered. We obtain this kind of information from the software vendor websites and other resources, such as BID, CERT, and Japan Vulnerability Note (JVN) [8]. Based on the obtained information, we further sanitize the dataset. We classify 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 follows:

- **Confirmed and patched:** An advisory has been published by the vendor that explains the vulnerability and provides patching information. We find that 556 vulnerability records fall into this category.
- **Reported and patched:** The vulnerability has been reported to the vendor, and patches were released. We find 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 find five vulnerability records belong to this category.
- **Confirmed but not patched:** The vulnerability has been confirmed by the vendor; however, no patches were provided. We find five 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 find 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. We exclude 33 records as they do not include enough information.
- **Large-scale experiment:** a large-scale automated experiment has been conducted by Dormann [79] 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 ten 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. We exclude those records in order to produce more balanced and meaningful results. 1389 vulnerability records belong to this category.

4.3.2.4 Data Classification

After cleaning the dataset, 663 records remained. Next, we manually examined the vulnerability records to categorize them into two groups: Android platform vulnerabilities and Android app vulnerabilities. Android app vulnerabilities are any vulnerability that resides 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.

4.3.3 Results

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

4.3.3.1 Vulnerability Type Trend

We aim to tackle vulnerabilities at the app level, but not at the OS level. We want to focus our study at the app level because we want to understand how SAST tools perform with modern apps considering their architecture. Thus, an essential 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, we utilize the NVD classification of vulnerability type based on the Common Weakness Enumeration (CWE) [5], since it is the primary classification in the NVD.

CWE is a list of software flaw types maintained by the MITRE Corporation and used by security organizations and researchers. There are 24 different types of flaws in the Android platform in the studied dataset, and 20 types of them occurred in Android apps. The percentage of each vulnerability type has been calculated to uncover the trending vulnerability in Android apps. As shown in Table 4.1, *Buffer Errors* 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 a 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 [167], rarely are buffer error vulnerabilities related to Android apps being discussed (See Section 2.2.6 for more details). Since buffer error is the most common vulnerability that occurred historically in Android apps, further analysis is needed to recognize how severe and dangerous buffer error vulnerabilities are.

4.3.3.2 Buffer Error Severity Trend

To analyze how severe discovered buffer error vulnerabilities are, we use the Common Vulnerability Scoring System (CVSS) [4]. CVSS is an open standard used to assess the

Table 4.1: Vulnerability types distribution in Android apps

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%

severity of security vulnerabilities, and it is platform and technology independent. We use the CVSS Base score version 2 standards.

We find during our analysis that 97% of the buffer error vulnerabilities have high risk. These results show that buffer errors have severe implications for Android apps' security and end users. Also, 99.26% of buffer errors in Android apps are remotely exploitable, and no authentication is required. Thus, it could be concluded that the buffer errors in Android apps are easy to exploit. Finally, we find that 95.6% of buffer error vulnerabilities ultimately impacted the target app in terms of confidentiality, integrity, and availability.

Takeaway 1: Buffer errors are the most common vulnerability in Android apps. They also have the highest risk that compromises the 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 study's focus. 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.

4.4 Case Study

4.4.1 Selection Criteria for Buffer Error Vulnerabilities

In our case chapter, we want to test state-of-the-art SAST tools against buffer error vulnerabilities in Android apps. While we could write our toy Android apps and inject buffer error vulnerabilities, we feel that it would be a biased experiment. Hence, we mine the vulnerability records from NVD to identify example vulnerabilities in open-source Android apps that we can use as case study subjects. Having open-source Android apps was necessary so that we could run SAST tools to analyze the source code.

We find nine buffer error vulnerability records in three open-source Android apps or have open-source components (Google Chrome, Android Browser, and Mozilla Firefox). Thus, examining the SAST tools against existing popular Android apps with real-world vulnerabilities makes our experiments, results, and conclusions stronger. In this section, we describe each of the vulnerabilities categorized within corresponding reason in Table 4.2 and follow up with a discussion on some of the common attributes among all of them.

4.4.1.1 Buffer Size Miscalculation

- **CVE-2008-0985**

  A remote attacker could cause a heap-based buffer overflow by persuading a victim to visit a malicious website that contains GIF components. The vulnerability occurs at the GIF library in the WebKit framework in the 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 using malloc. However, the buffer is filled in with bytes based on the GIF image's real width and height. The buffer copy operation is performed within a loop.

Table 4.2: Studied buffer error vulnerabilities in Android apps

CVE ID	Reason	C/C++	Affected App	Source (input)	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	Within the 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

A remote attacker could cause a heap-based buffer overflow in Google Chrome by persuading a victim to visit a malicious website containing crafted image components. The overflow happens during image processing in Skia [1] that miscalculates the buffer size of the image.

4.4.1.2 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 website that includes a crafted bitmap. The Google Chrome rendering engine Blink fails to render that particular size causing a buffer error. From the source code point of view, the bitmap size that is tainted is used to allocate a new buffer in the heap without validating the size, leading to integer overflow, and then it is used as an index to access an array buffer that leads to a buffer overflow.

- **CVE-2014-3201**

[1] https://skia.googlesource.com/skia/

A remote attacker could cause a buffer overflow by persuading a victim to visit a malicious website 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. In the source code, the issue happens because the allocation of the buffer does not consider the variable size of the webkit-scrollbar.

- **CVE-2014-1705**

  A heap-based buffer overflow vulnerability was found in the Google V8 JavaScript engine, an open-source JavaScript engine written in C++. It exists within the 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. A boundary check has been added to fix the issue..

- **CVE-2014-1710**

  In this vulnerability, Google Chrome does not validate whether a specified location is within a shared memory segment's bounds. This allows remote attackers to cause GPU command-buffer memory corruption and a denial of service. The GPU command-buffer is how 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 the GPU process to crash. In this case, the user should open a carefully malicious a crafted page that has scripts to dynamically modify the web page's structure after load time. To fix the vulnerability, a boundary check was added such that the operation on the buffer does not go beyond the boundary of the buffer.

4.4.1.3 NULL Pointer Dereferences

- **CVE-2012-4190**

  This vulnerability was reported by a user who complained 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, 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 the Cairo library to use Mozilla in-tree setlcdfilter of FreeType. Thus, calling a system function instead of using a local function led to a NULL pointer dereference.

4.4.1.4 Incorrectly Applied Rules

- **CVE-2016-5200**
  A heap-based buffer overflow vulnerability was found in the Google V8 JavaScript engine. A remote attacker could exploit 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-bounds read/write. The patch includes fixing the typing rule for Math.sign [7].

4.4.1.5 Off by One Error

- **CVE-2016-5199**
  A remote attacker could cause a heap corruption 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, which results in corrupting a number of pointers.

4.4.2 Common Attributes of Buffer Errors in Android Apps

All the studied buffer error vulnerabilities are in client-side apps, such as web browsers. Also, seven out of nine of the studied buffer error vulnerabilities are C++ based, and they have some common characteristics. For instance, the input is read from untrusted sources, untrusted input is inadequately validated, or lacks boundary checking. Also, most of the studied vulnerabilities involve pointer indirection. Another observation is that buffer errors in our case study are inter-file/inter-procedural, as the buffer is allocated in one function in one file and overflows in another function in a different file. Sometimes, the inter-file/inter-procedural communication occurs in a cross-language manner, meaning that buffer errors occur in the native context, while the untrusted input comes from other contexts, such as Java.

4.4.2.1 Tested Static Analysis Tools for Buffer Errors

In this chapter, we only focus on SAST tools that support source code analysis. We study 17 popular SAST tools to detect buffer errors. Table 4.3 shows the studied SAST tools classified based on [170]. We gather some tools by reviewing research work that evaluated methods that detect buffer errors [210] [106] [190] [170] [196]. We also study some other SAST tools popular in the wild, such as Clang Static Analyzer [69] and Frama-C [30]. These tools are general tools for multiple platforms. Table 4.3 shows the studied buffer error SAST tools classified based on [170]. Also, we categorize the static analysis methods into two categories: Open source and Commercial.

We determine which of state-of-the-art SAST tools could detect the nine Android buffer error vulnerabilities described in Table 4.2. As our study reveals, most Android buffer errors occurred in C++ language; thus, all SAST tools that target only C language could not detect these vulnerabilities. So we exclude five out of 17 SAST tools that target the C language, such as Splint, BOON, ARCHER, and UNO. We find that 12 tools could analyze C++ and could potentially detect studied vulnerabilities. Six out of the twelve SAST tools are commercial tools that we did not test. However, we gather some commercial tools using an academic license, but we could not test them as they have a limit on the tested line of code (LOC), such as CodeSonar by Coverity, which only allows testing one million LOC, but the total lines of code in all the apps in our dataset exceeds one million LOC. Hence, we only test free and open-source SAST tools.

We investigate six SAST tools designed to detect buffer errors to study nine vulnerabilities in open source components of the tested apps. The study shows that the free and open-source state-of-the-art SAST tools do not efficiently discover studied buffer error vulnerabilities in Android apps. We analyze the tools carefully to see why they could not discover buffer errors. The next section discusses that in detail.

4.5 RQ: Are state-of-the-art SAST tools for buffer errors able to detect vulnerabilities reported in the wild for Android apps?

SAST 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 SAST tools. An efficient tool in this case study is the one that can detect the studied vulnerabilities as a true positive.

Table 4.3: State-of-the-art SAST tools to detect buffer errors

Method Name	Type	Language	Inference Algorithm	Sensitivity	Granularity
Polyspace Bug Finder [210] [30] [82]	Commercial	C/C++	Constraint: abstract interpretation	Flow	Inter-procedural
Parasoft C/C++test	Commercial	C/C++	String pattern matching, Constraint: symbolic execution	Flow	Inter-procedural
Klocwork [82]	Commercial	C/C++	Unpublished	Flow, path	Inter-procedural
Coverity [82]	Commercial	C/C++	Unpublished	Flow, path, context	Inter-procedural
PVS-Studio [30]	Commercial	C/C++	Constraint: symbolic execution, annotations	Flow, path, value range	Inter-procedural
CodeSonar [30]	Commercial	C/C++	Constraint: symbolic execution, taint data flow	Flow, path	Inter-procedural
ASTREE [106][30]	Commercial	C	Constraint: abstract interpretation	Context	Inter-procedural
ARCHER [210]	Open source	C	Constraint: symbolic execution	Flow, path, context, alias	Inter-procedural
BOON [210][106]	Open source	C	Constraint: integer range	N/A	Inter-procedural
Splint [210][190][106]	Open source	C	Annotation	Flow	Intra-procedural, inter-procedural
UNO [210][30][190]	Open source	C	Annotations	Flow, path	Inter-procedural
Flawfinder [190][106]	Open source	C/C++	String pattern matching	N/A	Token
RATS [190][106]	Open source	C/C++	String pattern matching	N/A	Token
Cppcheck [190][106]	Open source	C/C++	Constraint: integer range	Flow, context	Inter-procedural
Clang Static Analyzer [69]	Open source	C/C++	Constraint: symbolic execution, annotation	Flow, path	Inter-procedural
Farma-C [30]	Open source	C/C++	Constraint: abstract interpretation, annotations	Flow, value range, point-to	Inter-procedural, System dependence graph
IKOS [62]	Open source	C/C++	Constraint: abstract interpretation	Flow, path, point-to	Inter-procedural

4.5.1 Methodology

1) Collecting open-source SAST tools that detect buffer error:

We gather the six open source SAST tools that discover buffer error. Our study tests the following tools that support C++: IKOS [62], Frama-C [91], Clang Static Analyzer [69], Cppcheck [190], Flawfinder [190], and RATS [190]. Table 4.4 summarizes all tested tools and their versions.

2) Collecting the source code of the vulnerable apps: We then gather the source

Table 4.4: The studied free and open source SAST tools that detect buffer errors in C++

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

code of open-source Android apps that have buffer error vulnerabilities in our study.

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 SAST tools against the source code of the vulnerable apps: We test the source code of each app through the static analyzers. We run each tool several times using different options and flags. Then, we build automated scripts to run the tests automatically and save the results to files.

4) Analyzing the results: in this step, we analyze the results that were stored in files. Since most of the tools could report errors with the source code file path, error type, and error line number, we open the source code files and trace and analyze all reported errors. All collected source code and scripts could be found here [44]

4.5.2 Results

We find that Flawfinder, RATS, and Cppcheck SAST tools are easy to use. They could be executed through a command-line interface, and they accept a list of files or project directories to test with a set of options as parameters. Also, these tools show their results by default in the system's command line. The results contain the files path, the lines of code suspected of having 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. The source files in our case study need to be preprocessed first to analyze source files with Frama-C; 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 into LLVM bitcode first, which could be analyzed by the tool then. Like Frama-C, the files need many preprocessing settings when analyzing source files in complex apps. However, manually setting preprocessing for source files is hard to achieve with large and complex apps such as Mozilla Firefox and the open-source components of Google Chrome.

Our analysis reveals that none of the tools were able to detect any of the studied vulnerabilities. Table 4.5 shows the results of our analysis. In Table 4.5, we include a number to refer to why a tool cannot determine the specific vulnerability.

4.5.3 Discussion

In this subsection, we want to discuss possible reasons why the studied free and open-source SAST tools could not detect the nine vulnerabilities. To do that, we look at the characteristics of the techniques underlying the tools (in the context of buffer error vulnerabilities). In total, we present six such characteristics (the numbers in the enumeration below corresponds to the numbers in Table 4.5):

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 another language (except CVE-2012-4190), while the SAST tools only examine the C/C++ native code. This creates a dangerous blind spot in these tools as they lack comprehending modern frameworks since they only understand and work on a single language at time [136]. None of the six tools examined can determine tainted data that comes from code written other languages, such as in Java. This applies to other tools studied in Table 4.3.

Table 4.5: Tested SAST tools results: The numbers indicate the reason in the enumerated list below, which explains why the tool could not find the vulnerability.

CVE ID	RATS	Flawfinder	Cppcheck	Clang Static Analyzer	IKOS	Frama-Clang
CVE-2008-0985	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2012-4190	2,3	2,3	2,4	2,5	6	6
CVE-2014-1705	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2014-1710	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2014-3201	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2016-5182	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2016-5199	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2016-5200	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6
CVE-2017-5014	1,2,3	1,2,3	1,2,4	1,2,5	1,6	1,6

2. Unable to keep track of pointer operations: The tool is unable to keep track of data when a buffer is manipulated using nontrivial pointer operations (CVE-2017-5014), or when a null-pointer is dereferenced (CVE-2012-4190). Almost all of the studied nine vulnerabilities involve some sort of pointer operation. We know from past research that pointer indirection could be hard to discover by most commercial tools [123]. For instance, the Heartbleed is a pointer indirection vulnerability in OpenSSL [29] that was not discovered by commercial tools, such as CodeAdvisor, CodeSonar by Coverity, Klocwork, and Veracode [123].

3. Simple lexical analysis only instead of semantic analysis: RATS and Flawfinder only perform simple pattern matching, basically tokenizing the source code and looking for tokens that are well-known to cause buffer errors. They mostly focus on tokens that could potentially be a source or sink to buffer errors, such as dangerous functions like the **strcpy** function. However, the sinks in our dataset have 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. This type of issue may not be detected by a simple lexical analysis tool. In general, Flawfinder and RATS have a 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 the 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 pointers dereference issues. They are passed as parameters 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 discovered in CVE-2012-4190 and reported to the NVD.

5. Limited scope of analysis: Clang Static Analyzer performs a high-level AST analysis. However, since it uses the AST generated by Clang, it 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. Almost all of the vulnerabilities in our case study are inter-file/inter-procedural.

6. Tied to a specific compiler: IKOS and Frama-C directly invoke the Clang/LLVM compiler framework that supports both C and C++. Therefore, they cannot analyze Android apps that use specific toolchains. Besides, these tools are difficult to be integrated into build systems. However, these tools perform sophisticated semantic, data flow, control flow, pointer operation, and inter-procedural analysis in a robust fashion.

From the reasoning above, we can see that an ideal static analysis tool to detect Android apps' buffer errors will:

- performs cross-language analysis to be able to understand both Java and the native code contexts.
- utilizes tainted data flow as an inference algorithm.
- employs in-depth semantic, data flow, control flow, and pointer operation analysis
- employs Inter-procedural/Inter-file analysis

4.6 Threats to Validity

One possible threat to external validity is that we did not test many commercial tools that use their 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 us to analyze one million lines of code. However, the analyzed apps in our study include several million lines of codes.

Also, 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 commercial tools [123]. For instance, the Heartbleed vulnerability involves pointer indirection in the OpenSSL [29] that was not discovered by commercial tools, such as Coverity Code Advisor, CodeSonar, Klocwork, and Veracode [123]. Heartbleed vulnerability was hard to be found due to using multiple levels of pointer indirection and the complexity of the execution path from the buffer allocation to buffer misuse, which is similar to this case study. Additionally, we qualify that our experiments are done only on free and open-source tools, by clearly stating "free and open source" in the study to avoid any misunderstanding.

Another threat to validity is that some apps, such as Google Chrome, contain some closed source parts that might include the vulnerability. However, in our study, we ensured that all vulnerabilities reside in the open-source component (e.g., V8 and webkit). We browsed the source code as we know where the vulnerabilities exist from the NVD website. 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 the NVD to ensure that they related to the Android ecosystem and removed all biased records. Also, we manually reviewed the SAST tool results to guarantee the correctness. To address any threat to internal validity (i.e., mistakes we could have made) and for the ability of anyone to replicate our experiments, we provide all the data in our experiments [44].

Another limitation of our study is that the sample size is small, yet we have tested all available vulnerabilities that occurred in open source apps. Another threat is that there is a possibility that the tested vulnerabilities may not be representative of other buffer error in Android apps in general. However, when observing the descriptions of other buffer error vulnerabilities in our dataset that occurred in closed source apps, we found some similarities with our sample, such as apps are written in C++ and buffer errors happened due to input that was read from untrusted sources.

4.7 Conclusions

In this chapter, it was found that buffer errors were the most frequent type of vulnerability in Android apps and easy to exploit. Therefore, we studied the effectiveness of state-of-the-art free and open-source SAST tools for detecting buffer error vulnerabilities in Android apps. Also, our goal was to understand how buffer errors happen in the Android context. Our findings indicate a lack of SAST tools that target Android to detect buffer errors. Thus, general SAST tools for native code may be usually used by Android developers.

However, currently free and open-source SAST tools for C++ could not detect real-world buffer error vulnerabilities in Android apps.

Also, our study found some patterns of characteristics for buffer errors that occurred in Android apps, such as occur within C++, due to pointers indirection, untrusted input travels from other contexts to C++ where the buffer is misused. Thus, by utilizing these patterns, SAST tools could be built to work more efficiently. The experimental results show that such an evaluation brings a vital 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 affect other modern apps in other platforms that include native C/C++ as a library), (3) has a 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/inter-file analysis to analyze data travel cross procedures.

Chapter 5

SoS: Source Sink Filter for Modern Software

Static taint analysis is a useful analysis technique to detect a range of security vulnerability types. Usually, this type of analysis involves analyzing data from sources to sinks. However, modern software is more complex, making it challenging to applying such an analysis method to detect security vulnerabilities. This chapter proposes a methodology to filter out the source and sink pairs that do not have feasible paths. The goal is to introduce a practical and generally low-cost extensible and configurable static analysis methodology to remove many likely not reachable pairs.

5.1 Introduction

A static taint analysis approach has been primarily adopted to detect different types of critical security vulnerabilities. Static taint analysis is an information-flow analysis that concerns about values from untrusted input (i.e., sources) that may spread into security-sensitive operations (i.e., sinks) without being validated by a sanitizer [171]. This type of analysis applies to *Input Validation and Representation* (IVR) class of vulnerabilities, such as command injection, SQL injection, and buffer overflow. Static taint analysis also concerns about information that is confidential (i.e., sources) may flow to places where it becomes publicly available information (i.e., sinks). This type of analysis applies to *Information Leakage* class of vulnerabilities [195]. However, detecting these critical classes of security vulnerabilities in modern programs is more challenging than ever before. This is because modern programs provide a different level of complexity. Thus, directly using taint analysis with modern programs may not be effective.

The first challenge is that modern programs may integrate multiple sub-components of different languages (i.e., polyglot software), such as using libraries that belong to different programming languages, to keep up with the current technologies. Generally, software that is using just one language is becoming much less common [140]. In practice, software developers use different languages in one program to achieve specific ends. The reason is that some languages are appropriate for one task but not another; hence using multiple programming languages allows developers to use the right tool for the task. For instance, an Android app is usually written in Java/Kotlin. However, if a component needs to be performance efficient, C/C++ could be used. Also, developers may choose to use existent libraries or open-source components written in different languages. This complexity may lead to more vulnerable software, as data travels across languages, and possibly a problem that may be scattered across these different languages gets unnoticed [44]. Usually, static analysis methods look into one language at a time, creating the so-called blind spot.

Besides, in modern apps, such as Android apps, there are many API calls that read-/write from/to shared resources, and the number of API calls increases over time. For instance, in Android 4.2 Jelly Bean (API level 17), the number of sources is 18,044, and the number of sinks is 8,278 as produced by SuSi [158]. While, in Android 10 (API level 29), the number of sources is 38,504, and the number of sinks is 8,704 as produced by SuSi [158]. SuSi is a machine-learning tool to define an exhaustive predefined source/sink pattern in Android directly from Android's source code. It was stated that "no matter how good the tool, it can only provide security guarantees if its list of sources and sinks is complete" [158]. However, a complete list can be a burden on such in-depth taint analysis tools. This means that a single program can have a large number of detected sources and sinks. Then, every source and sink pair needs to be analyzed using the taint analysis, making traditional taint analysis techniques deemed to be ineffective. A large number of API calls could be even more problematic with large-scale software. For that reason, multiple taint analysis tools, such as LeakMiner [206], Amandroid[201], and DroidSafe [99], limit the number of the predefined source/sink list to be examined. Other tools, such as FlowDroid [52], IccTA [130], and DIALDroid [61], use *some* of the predefined source/sink list identified by SuSi [158]

Considering the above challenges, we examine a static analysis methodology that aims to run on large programs that may span multi-languages and can cover all the source/sink APIs/patterns preidentified. The main goal is not to perform the taint analysis itself, but rather to filter out all the source and sink pairs that are possibly not reachable, which may help taint analysis tools run on all the possible risky pairs. This chapter investigates the usability of using the call graph to determine the reachability between sources and sinks. Call graphs are universal among programming languages and can be inexpensively acquired. The main idea of this methodology is first to reveal possible vulnerabilities that span into multi-languages. It also reduces the search space and can help the code reviewers focus on likely problematic issues, or it could also be used with static analysis tools as a preprocessing phase to minimize the search space and help SAST tools to apply in-depth analysis on the possibly reachable pairs.

The main contributions of this study are as follows:

- We highlight the necessity for a general methodology to quickly examine the reachability between all possible source and sink pairs, including pairs that cross-languages,
- We use the call graph construct to filter out the likely unreachable source and sink pairs,
- We illustrate two approaches based on a call graph to measure the reachability,
- We implement SoS and evaluate the two described approaches.

5.2 Static Analysis Approach

The proposed static analysis approach is based on lexical analysis and the call graph construct. The approach first uses lexical analysis to extract all the sources and the sinks existing in the source code in different languages. If the number of sources or the number of sinks is zero, the approach can safely conclude that there is no taint issue there; thus, no further analysis is required. Otherwise, the approach then needs to filter the source and sink pairs. To this end, the proposed static approach uses a call graph that spans multi-languages to determine which pairs of sources and sinks can be reachable and eliminate all the pairs that are likely not reachable by a feasible path.

The methodology first proposes a method to construct the *Complete Call Graph* (CCG) for a program, which includes all call graphs of sub-component written in a different language and the call graph of the cross-language communication. To determine the reachability between sources and sinks in the CCG, we introduce two approaches. The first approach investigates whether a source and sink pair belongs to one connected component sub-graph in the undirected version of the call graph (the *Undirected Complete Call Graph* (UCCG)), and if so, it measures the distance (i.e., call edges) between the source and the sink pair. The assumption here is that if the distance between the source and sink is small, they are likely reachable; otherwise, the pair can be discarded from further analysis. The second approach assumes that a source and sink pair are reachable if they have a common ancestor, and this approach uses the CCG. To explain the methodology, we use the buffer error problem that can be detected by taint analysis that occurs in the Android app as an example of polyglot software. The next section explains the methodology in detail.

5.2.1 Definition of Sources and Sinks

Generally speaking, in the tainted data-flow analysis field, the source indicates where data comes from, while the sink refers to where data ends. As mentioned in section 5.1, sources and sinks differ based on the vulnerability type. For the IVR class of vulnerabilities, data is tainted if it comes from an untrusted source, such as a file, socket, network, environment variables, or end-user. Also, data that is altered by any tainted data is considered tainted. In the tainted data-flow analysis, when tainted data travels to a vulnerable point in the program, the so-called sink implies a potential problem. Nevertheless, the sink differs based on the analysis problem. In the information leakage problem, a sink is sensitive data that leaves the program. One example of the information leakage problem is log forging vulnerability, where sensitive data is printed into a log file. In the SQL injection problem, a sink is a sensitive function that executes SQL statements against a database, while in the command injection problem, a sink is a statement that executes arbitrary commands. In the buffer error problem, sinks are the program statements that might perform unsafe buffer operations. Hence, it is essential to identify the specific vulnerability problem in the context. In this chapter, the buffer error problem, a subclass of the IVR vulnerability class, was chosen to describe the methodology. Hence, the definition of the sources and the sinks are based on this specific problem. Besides, we select Android apps to represent modern software that is composed of multi-languages. We only consider the communication between C/C++ and Java in this approach.

Definition 1. *(Sources): A set of program statements that accept untrusted data, such as methods that read files or inputs from the user, are untrusted sources.*

In the buffer error problem in Android apps, untrusted sources can belong to any language. Hence, in Android apps, sources might belong to Java or C/C++. Hence, there is a necessity for every programming language to identify all possible sources that accept untrusted input. Many API calls receive untrusted input on the Android side, such as sockets, files, and user input. As a result, manual identification of sources can be infeasible. Hence, we use SuSi [158] to extract all the possible sources in Java, which is a practical best-effort solution that identifies sources to a large extent. All the methods that are generated by SuSi and have return types as objects have been used. The number of sources used from SuSi is 20,820 sources that belong to 5,903 classes. We manually construct a list to define the sources that accept tainted data in C/C++ inspired by the technical specification ISO/IEC TS 17961 [34] in traditional C/C++. The number of sources that we compile is 149. For instance, the traditional C/C++ function *read* attempts to read a number of bytes from the associated file descriptor.

Similarly, in the C/C++ Android context, we manually gather Android NDK API. We retrieve the full list of the Android NDK API from the Android website [37]. The constructing of the list is based on the methodology described in SuSi [158], such as APIs that include the keyword *get*, *read*, and *receive*, which indicate a source. Also, we conduct a manual investigation of the APIs, and we only include the API calls that refer to reading from untrusted sources. The number of sources that we compile is 277. For instance, the NDK C/C++ function *AAudioStream_read* attempt to read data from the audio stream. Further identifying API calls that belong to third-party libraries or the new language standard is desirable, but it is out of the scope of this study.

Definition 2. *(Sinks): are a set of program statements where the actual problem takes place.*

In the buffer error context, sinks are the set of statements that perform read/write operations on a buffer. In the buffer error problem, sinks only occur in the C/C++ language. From the literature, a read/write operation usually achieved through known unsafe function, such as *strcpy* and *memcpy*. A buffer error can happen in an array or pointer arithmetic while writing and reading within loops. When looking at C/C++ unsafe standard libraries internally, one can see that they involve a loop and buffer read/write operation with the lack of performing boundary checks. C/C++ are strictly performance-oriented languages; hence boundary checking would be against the philosophy. Therefore, it is up to the end developer to decide when a boundary check is necessary. In this chapter, these program statements are called sinks. To define sinks in C/C++ language, we follow a similar approach to defining C/C++ sources. We use a set of functions that perform unsafe buffer operations that are well defined and discussed in the literature, such as memcpy, strcpy, and strncpy. Another type of sink that we consider is when the read/write operation of a buffer occurs within the loop, such as accessing an array or pointer arithmetic. For example, the following code is a pattern that we are including in our analysis, which includes accessing a buffer through pointer arithmetic:

```
 while(*src_ptr != ’\0’)
   *dest_ptr++ = *src_ptr++;
*dest_ptr = ’\0’;
```

The number of sink APIs gathered is *88*. However, in the future, a better approach is desirable. Finally, all the lists of the sources and the sinks for each language are constructed as XML files. This way, it can be more fixable to add more sources and sinks as discovered. The full list of sources and sinks in Java and C/C++ can be found here [49].

5.2.2 Finding Sources and Sinks in Android Apps

Once all possible sources and sinks are defined, the lexical analysis could be performed to find the sources and sinks in the source code. Lexical analysis is a lightweight analysis method that can be very fast to extract tokens of interest in the source code. The intuition here is that only software that has sources and sinks may be vulnerable. Then, if the source code file has sinks or sources, the file parse tree is built to recognize which methods encapsulate the sources and the sinks. In this study, the methods that are defined by the software team that has a call to functions/APIs that read data from untrusted sources are called *enclosing source*. The methods that are defined by the software team and comprise potentially unsafe operations on a buffer, using functions/APIs calls or a pattern, are called *enclosing sink*. The parsing phase gathers the related information of the sources and the sinks, such as the line number, the parameters, and the enclosing method parameters. If sources and sinks are being located, then more analysis is necessary. Otherwise, the analysis can terminate with the conclusion that the software is safe.

5.2.3 Reachability Through Call Graph

The call graph is a valuable representation of the control flow information. A call graph is a directed graph that depicts calling relationships between methods in a program source code [166]. Each node depicts a method that may be internal methods implemented by the software team or external methods that could be system and third-party library calls. Each directed edge from method A to method B denotes one or more invocations from method A to method B.

Definition 3. *(Call Graph): A call graph is a directed graph G(V, E) with node set V=V(G), representing the methods in the program, and edge set E=E(G), where* $E(G) \subseteq V(G) \times V(G)$*, representing the method calls.*

This chapter suggests using the call graph construct to infer the reachability between sources and sinks. However, a fundamental question to ask is: can an analysis using the call graph give useful results? Furthermore, how can we use such representation? Thus, we introduce two approaches. Before describing the proposed approaches and the reasoning behind them, lets first introduce some call graph constructs used in this chapter.

Definition 4. *(Complete Call Graph): the Complete Call Graph (CCG) is a call graph G(V, E) that connects two or more call graphs that might span multiple languages.*

This graph connects multiple call graphs by understanding how different components belong to various languages communicates. Section 5.2.4 explains how to construct such a graph.

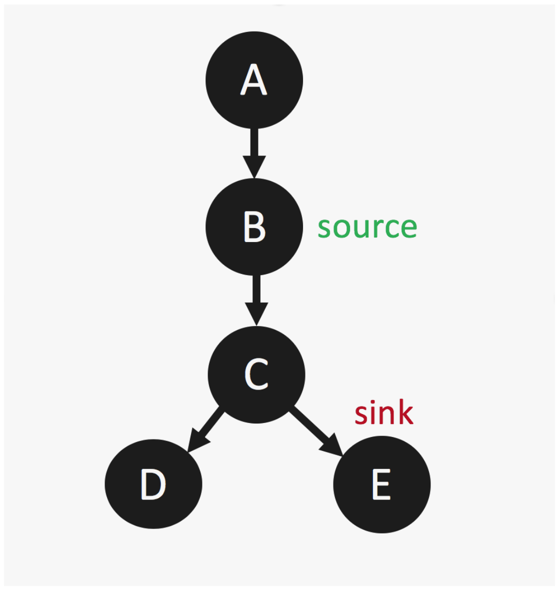
Definition 5. *(Undirected Complete Call Graph): the Undirected Complete Call Graph (UCCG) is derived from CCG, but it operates on the undirected edges.*

One approach uses the UCCG construct to be applied to any program either with one programming language or multi-languages to get useful results. The other approach uses the CCG construct, and it is only applied to programs that include multi-languages. Nevertheless, in this chapter, we describe the problem in the multi-language program setting. Now let us explain the approaches and how do they use the above call graph constructs.

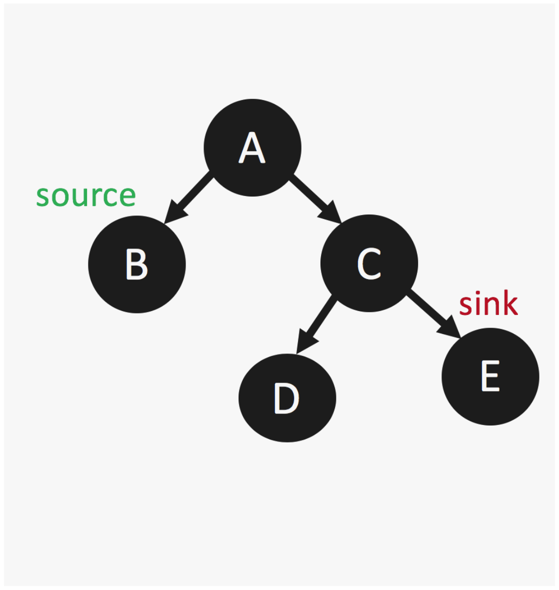
5.2.3.1 Reachability Through UCCG

This approach can be applied to any program either with one programming language or multi-languages. It uses the UCCG construct. The intuition behind using the call graph with undirected edges is that a source and a sink are reachable if there is a path between the source and the sink. To be more precise, a sink can be reachable by a source if there are control and data-flow from the source to the sink. However, how can such information be extracted from the regular call graph? The control flow information might be explicit in such a graph. When method A calls method B in the call graph, a directed edge from A to B represents the control flow information. Yet, how about the data-flow information?

The directed edges from the source to the sink can only provide control information but not data-flow information. Figure 5.1 demonstrates the problem. Figure 5.1a shows a simple call graph that includes five methods. In that graph, method A calls method B, which reads input from an untrusted source. Method B then calls method C that calls two methods: D and E. Method E includes a sink, which is a sensitive buffer operation.



(a) Direct flow from source to sink



(b) Indirect flow from source to sink

Figure 5.1: Source and sink in a call graph.

Hence, the directed edges can infer both the control and data-flow as it suggests that the sink in E is possibly reachable from B that read an untrusted source. Thus, one can say the control flow from method A to method B implies data-flow from method A to method B.

When looking at Figure 5.1b, if such analysis based on the call graph construct starts at the source, the directed edges tell us that there is no direct control flow from the source to the sink; hence there is no perceived data-flow. However, this might not be accurate. Assume that method A calls method B that reads an untrusted input. Then, method B returns the data acquired from an untrusted source to A. Then, method A sends the untrusted data to method C, which in turn, sends the untrusted input to method E with a sensitive sink. Consequently, an edge between methods that include source and sink despite its direction might refer to a possible issue. Therefore, we assume that the same edge with both directions can represent possible data-flow since the called function can return a value or manipulate a reference or field object rendering back data changes to the caller.

Thus, this would assume that in the UCCG graph, a source and a sink are reachable if they reside in the same connected component. However, this assumption has its issues.

Lets first review the connected component definition:

Definition 6. *(Connected Component): A connected component of an undirected graph G is a sub-graph in which there exists a path between every pair of nodes.*

For example, the graph shown in the Figure 5.2 has three connected component sub-graphs. A node that has no incoming or outgoing edge is itself a component. The software may include multiple connected components, and hence a pair of reachable source and sink would be possible only if both the source and the sink belong to the same connected components. Ideally, this may be adequate if the analyzed source code represents a library. However, for software with a single entry point, there are usually large connected component sub-graphs and source and sink pairs are likely to belong to the same connected component sub-graph. Thus, the distance between the source and the sink can give a hint. One can configure the static analysis tool to filter out the source and sink pairs that are reachable by visiting a specific number of nodes (i.e., threshold), representing functions in a call graph, as measured by the Breadth-First Search (BFS) or the Depth-First Search (DFS) algorithms.

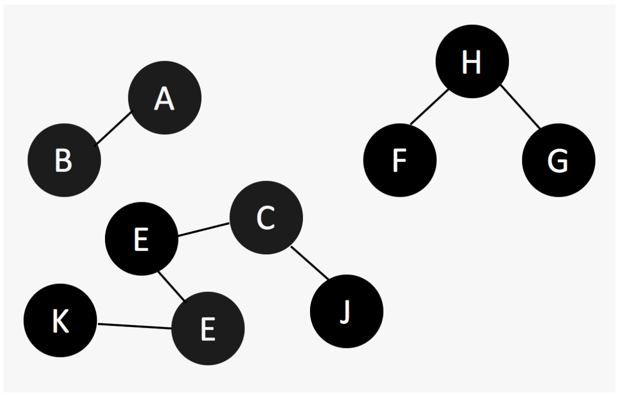


Figure 5.2: Connected component sub-graphs

5.2.3.2 Reachability Through CCG

This approach can be applied to the multi-languages programs or libraries, and it uses the CCG construct. This approach suggests that a source and a sink are reachable if they have a common ancestor in the CCG graph. This cannot be useful to single language programs with a single entry point. Since all the functions, and hence all the sources and sinks will have a common ancestor, the main method.

Definition 7. *(Lowest Common Ancestor): the lowest common ancestor of two nodes v and w in directed acyclic graph (DAG) G is the lowest node with both v and w as descendants, where each node is a descendant of itself.*

The assumption here is that when both the source and the sink have a common ancestor, this implies the sink can be reachable from the source. To clarify the approach, look at Figure 5.1; in both direct flow between a source and a sink 5.1b and indirect flow between the source and sink 5.1b, there is always a single common ancestor which is A. We adopt the naïve lowest common ancestor [59] to determine if a source and a sink have a common ancestor.

5.2.4 Constructing the CCG Graph

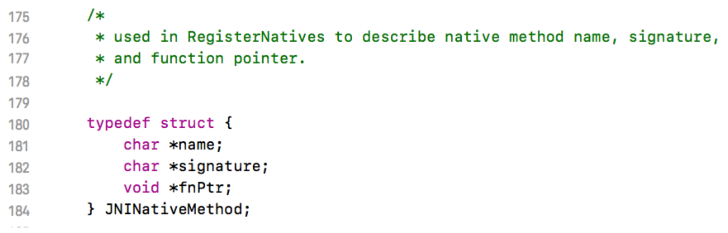
The CCG merges various call graphs that belong to different programming languages. In this setting, a call graph for each sub-component in each language is formed. Then, the cross call graph (*x-call graph*) is created, which a call graph that includes calls between functions/methods belonging to different languages. The call graphs for different languages can be constructed depending on existing tools. However, for the x-call graph, we provide the approach that we follow to build it.

5.2.4.1 Constructing the X-Call Graph

Different programming languages communicate using a Foreign Function Interface (FFI). Many programming languages have their own FFI. In Android apps, as an example that we use in this study, data might travel from the Java side into the C/C++ side and vice versa. This communication is accomplished using the Java Native Interface (JNI), the FFI for Java programming language.

Inferring calls from Java to C/C++ In the Java language, any method that has *native* modifier could eventually call C/C++ methods. In the C/C++ language and more particularly in the JNI layer, there are two ways to tell if an invocation from Java to C/C++ can happen. One way is by utilizing the naming convention used in the JNI specification. For instance, a method called *foo* that belongs to *MainActivity* class which belongs to *example* package on the Java side would be mapped into a C/C++ function named *Java_example_MainActivity_foo*.

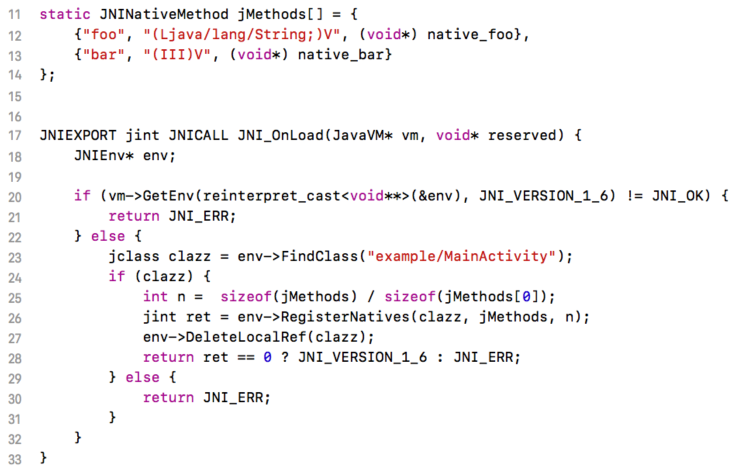
The second way is to dynamically register and map the Java method signatures to the native functions. In this approach, a data structure called JNINativeMethod defines



```
/*
 * used in RegisterNatives to describe native method name, signature,
 * and function pointer.
 */

typedef struct {
    char *name;
    char *signature;
    void *fnPtr;
} JNINativeMethod;
```

(a) JNINativeMethod struct in jni.h



```
static JNINativeMethod jMethods[] = {
    {"foo", "(Ljava/lang/String;)V", (void*) native_foo},
    {"bar", "(III)V", (void*) native_bar}
};


JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env;

    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK) {
        return JNI_ERR;
    } else {
        jclass clazz = env->FindClass("example/MainActivity");
        if (clazz) {
            int n =  sizeof(jMethods) / sizeof(jMethods[0]);
            jint ret = env->RegisterNatives(clazz, jMethods, n);
            env->DeleteLocalRef(clazz);
            return ret == 0 ? JNI_VERSION_1_6 : JNI_ERR;
        } else {
            return JNI_ERR;
        }
    }
}
```

(b) An example of dynamic registration

Figure 5.3: Dynamic function register resolution.

how Java methods are mapped into C/C++ methods. Figure 5.3a exhibits the JNINativeMethod structure that belongs to the JNI layer, where it includes the Java method name, the method signature, and the corresponding native function pointer. Figure 5.3b shows a code snippet that includes a definition of JNINativeMethod array jMethods to initiate a mapping for Java methods *foo* and *bar*. Also, the jMethods array defines the method signatures in the format of *(argument types)return type*. Table 5.1 shows the Java type resolution on the C/C++ side. When *RegisterNatives()* is invoked, it passes the jMethods structure to be dynamically registered. Then, the dynamic registration function mapping is accomplished by implementing the *JNI_ OnLoad()* method. Function calls from C/C++ to Java can occur as well.

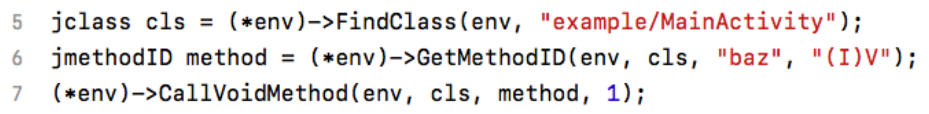
Inferring calls from C/C++ to Java C/C++ methods can invoke any Java method, which does not have to include the *native* modifier on the Java side. As shown in Figure 5.4, C/C++ calls Java method by preforming three steps: First the native code

Table 5.1: Java types in the JNI layer

Java Type	JNI Encoding
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
Java class	LJava class;
type[]	[type

tries to find the class, using class retrieving methods, such as *FindClass*, *GetSuperclass* or *GetObjectClass*, that the method resides in the *JNIEnv* variable to search for class. Then, it looks for the method in the given class using the method call. *GetStaticMethodID* or *GetMethodID*. Finally, the actual call is conducted using one calling method, such as *CallByteMethod()*. We use 90 methods in the JNI standard functions that call Java methods from C/C++ to extract such cross-language information (see Appendix G). The functions' parameters determine the Java called class, the method name, and the return type of method parameters type. It is worth noting that sometimes a Java/Android method can be deemed unreachable within Java/Android call graph; however, it can be called from the C/C++ side.

The cross-link *(xlink)* information between the two languages is extracted using lexical analysis and parsing. Once the xlink is recognized, all produced call graphs are merged into one graph representing the CCG.



```
jclass cls = (*env)->FindClass(env, "example/MainActivity");
jmethodID method = (*env)->GetMethodID(env, cls, "baz", "(I)V");
(*env)->CallVoidMethod(env, cls, method, 1);
```

Figure 5.4: JNI call from C/C++ to Java

5.3 Design

This section gives the design followed to build the **So**urce **S**ink (SoS) tool. The tool's principal purpose is to extract sources and sinks, including cross-language communications, to supply all possible reachable pairs of sources and sinks. The tool currently only handles buffer errors, but users could extend the tool to cover other classes of problems solved by taint analysis. Figure 5.5 illustrates the steps followed.

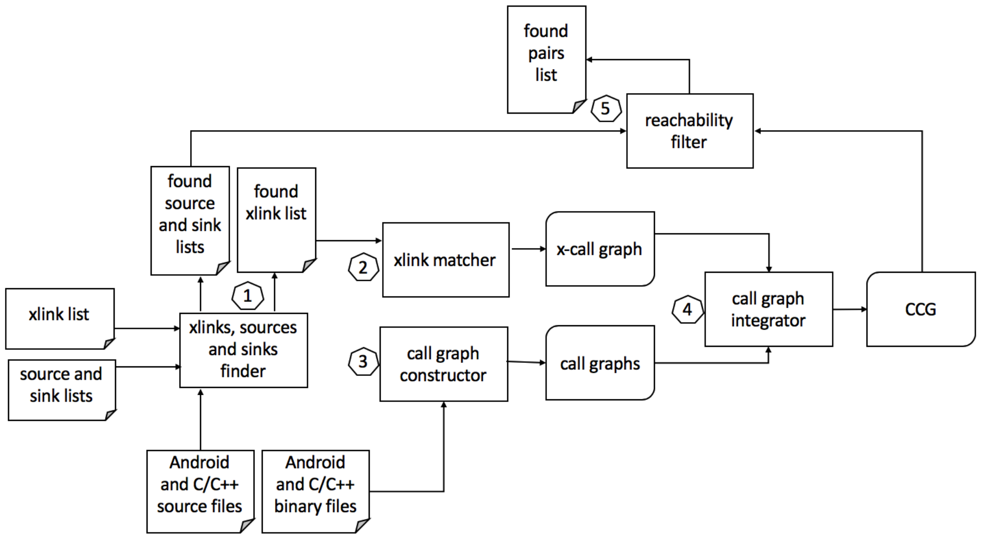


Figure 5.5: SoS design

1. The **xlinks, sources, sinks finder**, as exhibited in Figure 5.5, is a component that performs two primary tasks. The first task is to perform a combination of parsing and lexical analysis. Lexical analysis is a simple method to scan source files to produce a series of tokens that compose the source files. It matches the given tokens against the lists of tokens of interest (i.e., xlink, source, and sink related tokens). Accordingly, this component reads all the source files that compose the program as an input.

   The tool accepts lists of all the potential sources and sinks that belong to every language in the analysis. These lists can be in the XML/JSON file format. In this study, a separate list of sources and sinks for C/C++ and another list for Java/Android are injected as inputs. Moreover, a separate list of xlinks for C/C++ and another list for Java/Android are supplied as inputs. Note that methods that connect C/C++ and Java/Android could differ from the JNI API. Still, this was not considered in this study, yet the design of SoS is flexible. To incorporate other xlinks, sources, or sinks methods/tokens, one can add all the API to XML/JSON files, which the tool would recognize in such an analysis.

Once a token related to a source, sink, or xlink is found in a particular file, it parses the source file. So, the function name that includes the token of interest is gathered along with additional information. The information gathered by the parser comprises the source file path, the language, the line number, char position, the context of the line of code, enclosing method signature, source, and sink method name. Note that sometimes identifying a token needs a chain of statements to be resolved. This is particularly true for xlink, such as resolving calls from C/C++ to Java. For instance, when the analyzer finds the token *CallVoidMethod* in the following statement: *(env)->CallVoidMethod(env, cls, method, 1)*, it needs to resolve the class name and the method name (See Figure 5.4 for more details). Hence, the analyzer keeps track of all possible values of interest in a symbol table to fully resolve the call.

The other task is to obtain all the sinks in C/C++ that are not unsafe library calls but rather follow a pattern, such as having read/write operations inside a loop. Finally, this would produce a list of all found sources and sinks. If some sources and sinks were discovered, the tool moves to the following steps; otherwise, it terminates.

2. The **xlink matcher** reads the list of found xlinks among languages and matches functions that call each other. Then, when two methods that belong to two different languages are found by SoS match, two nodes and an edge are written into the x-call graph.

3. The **call graph constructor** generates a call graph for each programming language for each module. For instance, a program can incorporate more than one C/C++ subprogram, and so for every C/C++ subprogram, a call graph is generated. Besides, an Android call graph is generated. Every call graph would be produced as a separate dot file.

4. The **call graph integrator** reads all the call graphs that were constructed for every subprogram for each programming language and the x-call graph. It then merges all the call graphs, including the x-call graph, to obtain the CCG that shows how different languages are communicating.

5. The **reachability filter** traverses the UCCG and the CCG to determine the reachability. It has two modes. One mode traverses the UCCG to examine whether the source and sink pairs belong in the same connected component and separated by a

short distance based on both BFS and DFS. The UCCG graph is acquired by removing the edge direction from the CCG. The second mode traverses a CCG to find the least common ancestor of the source and the sink. Finally, it generates a file containing a list of all possible reachable pairs of sources and sinks.

5.4 Implementation

We implement SoS in Java, and the files that describe all possible sources, sinks, xlinks are processed as an XML format, and the output is generated as a JSON format. We implement the lexical and parser elements using ANTLR version 4. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator that uses LL(*) for parsing. The primary motivation for choosing the ANTLR parser generator is that it can parse diverse languages. ANTLR reads grammar that specifies a language and creates source code for a recognizer of that specific language that can build and walk parse trees.

The call graph generator has two options to build a call graph. The first method is using the ANTLR parser, which is straightforward that could be extended to any other language. The tool also accepts other means to generate a call graph using different tools to produce a more precise call graph. For instance, we use the Soot framework [193] to compose the Java call graph and FlowDroid [52] [129] to produce an accurate Android call graph. In addition, we adopt the SeaDsa framework [101] to generate C/C++ call graph that involves function pointers.

We process the call graphs using the JGraphT library, a graph library that includes multigraph algorithms to process graphs, such as transforming a directed graph into an undirected graph, finding the lowest common ancestor, and merging multiple graphs. For instance, the UCCG graph is created by JGraphT by providing a call graph, and then it would generate the undirected graph of it. JSON.simple and StaX were used to handle the input and output files. JSON.simple is a simple Java library to process JSON data. This library reads and writes JSON data following SON specification RFC4627. StaX (Streaming API for XML) is an API interface for reading and writing XML data. SoS data is available online [48].

5.5 Experiment Setup

We test SoS with five Android apps to examine the usability of the methodology. We select the Android apps based on the following three criteria: 1- Open-source apps since

Table 5.2: Tested Android apps versions

App Name	Version
AdAway	v4.3.2-15-g72e024b1
Orbot	16.1.2
Sipdroid	9067a3c
Signal	v4.50.3
ProxyDroid	fc94aec

Table 5.3: The line of code measure of different languages in the tested Android apps

LOC	AdAway	Orbot	Sipdroid	Signal	ProxyDroid
C	116,236	116,648	24,738	0	29,057
C++	0	0	3,367	32	1,540
C/C++ header	15,089	28,967	9,517	14	4,870
Java	24,684	46,533	25,125	157,778	17,663
Others	128,769	116,010	14,919	254,525	55,758
Total	284,778	308,158	77,666	412,349	108,888

the source code must be available to run the tool. 2- The size of the apps measured in thousands of lines of code (KLOC) to verify whether the tool can analyze large-scale software. 3- The apps incorporate both Java and C/C++ to check if the tool can determine the cross-language communications. We could not study the apps listed in Chapter 4 since most of the apps in that dataset are not fully open source. We choose the apps from a list of free and open-source Android apps [9]. When we review the list [9] to gather the apps that fulfill the above conditions, we find it challenging to obtain apps that meet all of the three conditions. Every single app on that list was examined to see if it satisfies the requirements, and thus only five apps were used. The apps are listed in Table 5.2. Table 5.3 shows the lines of code (LOC) for each programming language of interest. We use the *cloc* utility to measure the line of code metric.

Table 5.4: Number of sources and sinks detected in Android apps

	AdAway	Orbot	ProxyDroid	Signal	Sipdroid
source	862	1,495	326	8,926	1,142
sink	8,026	3,649	1,553	0	4,479
source/sink pairs	6,918,412	5,455,255	506,278	-	5,115,018
e-source	248	385	83	2,344	350
e-sink	494	241	196	-	239
e-source/e-sink pairs	122,512	92,785	16,268	-	83,650

5.6 Results and Discussion

5.6.1 RQ1: How many sources and sinks exist in the case study apps?

Table 5.4 depicts the information assembled regarding the source and sink pairs that occur in the apps. The *source* shows the number of methods/API calls that receive input from untrusted sources in C/C++ and Java in this study. The *sink* displays the number of sinks found in the app, and since this study focuses on buffer errors, the sinks would be in the C/C++ language. The *source/sink pairs* presents the number of all potential source and sink pairs; this was calculated by multiplying the number of sources by the number of sinks, considering every source can be reaching every sink.

The *e-source* (enclosing source) shows the number of methods that enclose or invoke methods that read sources from untrusted inputs. Similarly, the *e-sink* (enclosing sink) displays the number of methods that enclose or invoke those methods or patterns that can be sinks. The *e-source/e-sink pairs* exhibits the number of all possible paths from the enclosing sources to reach the enclosing sinks.

The results designate that usually, there are a vast number of source and sink pairs on average. For instance, the AdAway app comprises 862 occurrences of reading untrusted sources included across 248 enclosing methods; and 8,026 possible unsafe buffer operations across 494 enclosing methods. Thus, a taint analysis tool would have to perform 6,918,412 tainted tracing analysis to investigate whether data travels from every source to every sink. This number hints that checking every pair can be time and space consuming. The Signal app will be excluded from further analysis since it does not have sinks.

Table 5.5: The number of connected components and associated nodes in the UCCG in the tested Android apps.

App Name	Number of CC	Median	Min	Max	Total Nodes
AdAway	77	1	1	5,555	6,336
Orbot	67	1	1	5,175	5,247
ProxyDroid	91	1	1	7,751	7,860
Sipdroid	158	1	1	2,039	2,409

Table 5.6: The shortest path and the degree of all the nodes in the UCCG in the tested Android apps.

App Name	Average Shortest Path	Average Node Degree	Min Degree	Max Degree
AdAway	4	5	0	869
Orbot	3	6	0	1,009
ProxyDroid	5	6	0	721
Sipdroid	4	5	0	869

5.6.2 RQ2: Is the UCCG representation useful to determine the reachability between two nodes?

This RQ aims to study the UCCG graph characteristics and determine if it is useful to use such representation. To get such an understanding, we ask two questions. The first question is that to what extent call graphs consist of multiple connected components? If the graph has multiple significant connected components, this means a source and sink might belong to different connected components, and then it can be easily inferred that they are not reachable. We mean by a significant connected component that the connected component sub-graph has more than one node, since that a function/method that is not called and does not call any other function/method is considered a connected component. The source and sink pair that belongs to the same connected component is worth investigating. The second question we ask is that for one connected component, what is the average shortest path between two nodes (which may indicate the average shortest path between sources and sink).

Table 5.5 shows the number of connected components in each tested app. The *Num-*

ber of CC column shows the total number of connected component sub-graphs in the UCCG. The *Min* column indicates the number of nodes (functions/methods) in the smallest connected component sub-graph, while the *Max* column indicates the number of nodes (functions/methods) in the largest connected component sub-graph. SoS calculates the number of connected components in the UCCG using the BFS algorithm. The analysis indicates that the majority of the connected components are composed of only one function/method. The results also suggest that even call graphs produced by compilers might have many functions/methods that are not reachable or eliminated as dead codes. It may represent noise if the tested source code is an executable app; however, it can be reasonable if the examined source code represents a library. This means that inferring the reachability using only the connected components may not be practical.

Column *Average Shortest Path* in Table 5.6 shows the average shortest path between all the nodes in the CCG. It is clear that with the UCCG representation, when a source and a sink belonging to the same connected component, it would likely be separated by 3-5 nodes. Hence, using the shortest path measure between a source and a sink can be unuseful. The column *Average Node Degree* shows, on average, how many edges connected to one node (i.e., function call in CCG), which shows that there are 5-6 edges on average connected to one node. This means that a function/method has an average of 5-6 calls, either incoming or outgoing. Therefore, we suggest considering the degree of the nodes to filter out likely unreachable pairs. For that reason, we investigated using both the BFS and DFS and having a threshold to filter out the results.

5.6.3 RQ3: Is the reachable number of source and sink pairs identified in the connected components using the UCCG significantly reduced?

In this study, only the sources and sinks separated by at most 30 (as threshold) nodes (i.e., functions/methods) are considered. The distance between the sources and the sinks is measured using BFS and DFS. The threshold is configurable. We randomly select 30 nodes as a threshold. This was inspired by the value of the average node degree (See Table 5.6). However, further studies are desired to reveal the optimal threshold. Table 5.7, displays the number of reachable sources and sinks that belong to the same connected component when identified using the UCCG. The *reachable source/sink pair* displays how many pairs of sources and sinks are reachable using the UCCG, while the *reachable source/sink%* exhibits the percentage of reachable source and sink pairs compared to Table 5.4. The *reachable e-source/e-sink* illustrates the number of the enclosing source and the enclosing sinks that are reachable, and finally, the *reachable e-source/e-sink%* illustrates the reachability percentage between the enclosing source and the enclosing sink methods.

The outcomes in Table 5.7 demonstrate that reachable source and sink pairs designate a small percentage of all possible source-sink pairs. For instance, the AdAway app encompasses 6,918,412 possible source and sink pairs scattered across 122,512 enclosing source and sink methods. Nonetheless, the analysis insinuates that only 6,267 source and sink pairs are possibly reachable, around 0.09% of all the possible pairs. Also, a similar pattern applies to the other tested apps. The results signify that using a call graph-based method can quickly filter out many pairs that are likely not reachable by a feasible path.

Table 5.7: Source and sink reachability in the UCCG graph

	AdAway	Orbot	ProxyDroid	Sipdroid
reachable source/sink pair	6,267	17,755	3,276	1,520
reachable source/sink %	0.09%	0.3%	0.6%	0.02%
reachable e-source/e-sink pair	127	111	96	21
reachable e-source/e-sink %	0.10%	0.11%	0.59%	0.02%

5.6.4 RQ4: Is the number of reachable source and sink pairs identified using the lowest common ancestor and the CCG significantly reduced?

This RQ's assumption is that source and sink pair are reachable if they have at least one common ancestor. This RQ explores the usability of adopting the lowest common ancestor in the CCG to decide the reachability. Table 5.8 exhibits the number of enclosing sources and the enclosing sinks that have common ancestors. When looking at Table 5.7, AdAway has 6,267 possibly reachable pairs, while in this approach, AdAway has 7,867 possibly reachable pairs. However, both approaches show a promising reduction in the number of pairs that may have feasible paths. In both approaches, we might have false negatives, meaning pairs that are reachable but are not deemed as reachable by this approach. It is hard to do such analysis since we do not have a ground truth about all the reachable pairs.

Also, the frequency where there is a direct call from the source to the sink has been calculated. To do so, we used the Dijkstra Shortest Path algorithm to measure the distance between the source to the sink. If there is a distance found, this refers to a direct call.

Table 5.8: Source and sink reachability via the lowest common ancestor

	AdAway	Orbot	ProxyDroid	Sipdroid
reachable source/sink pair	7,867	238,455	4,551	106,796
reachable source/sink %	0.11%	4.3%	0.8%	2.08 %
reachable e-source/e-sink pair	197	202	111	38
reachable e-source/e-sink %	0.16%	0.21%	0.68%	0.04%

Table 5.9 reveals that among all the possibly reachable pairs, how many of them include a direct call from the source to the sink and how many of them include indirect call. *Direct call* shows the number of pairs that have a direct flow, as shown in Figure 5.1a. *Indirect call* shows the number of pairs that have an indirect flow, as shown in Figure 5.1b. The results refer to the fact that indirect calls can be more common.

Table 5.9: Direct call and indirect call between sources and sinks

App Name	Total	Direct Call	Indirect Call
AdAway	7,867	2,378	5,489
Orbot	238,455	80,098	158,357
ProxyDroid	4,551	2,974	1,577
Sipdroid	106,796	7,692	99,104

5.6.5 RQ5: How efficient is SoS is in terms of the running time?

In this RQ, we evaluate the performance of SoS in terms of the running time. SoS first performs a pattern matching for every single source code file. When a token or a pattern is found, the tool builds a parsing tree for the source code file to extract information, such as line number, line context, char position, enclosing method name, enclosing parameter, API call, the associated parameters. All the detected sources and sinks, xlinks along with the details are written into JSON files. Then, SoS builds a call graph for every language and attaches all the call graphs based on the detected cross-language links. All the call graphs are written separately into *.dot* files. It also traverses the UCCG/CCG graph to determine the reachability between sources and sinks, and finally, it writes the reachable pairs into a JSON file. We run SoS ten times for each app, and the average time was calculated to measure the performance. The time was calculated using the *time* command. Table 5.10 shows the running times for every tested project, as it is apparent that running time correlates with the KLOC. Note that our measured running time is coarse since the running time includes other tasks, such as parsing XML/JSON files and logging the results to files that were not excluded from the time measurement. We notice that creating a call graph for C/C++ takes a long time.

Table 5.10: SoS running time in seconds

App Name	Running Time	Standard Deviation
AdAway	312 sec	14.17
Orbot	474 sec	18.84
ProxyDroid	294 sec	11.56
Sipdroid	353 sec	14.05

5.7 Threats to Validity

Despite the best efforts carried in this study to decrease the impact of threats to validity, there are still inherent threats that need to be discussed. One potential threat to external validity is that the sample size of the Android apps is small. We only evaluate five Android apps, and one of them does not have sinks; consequently, it was excluded from further analysis. Additionally, the Sipdroid app is essentially a prototype and implements primary functionalities so that other complete apps can be built on top of it. Sipdroid is a free and open-source Session Initiation Protocol (SIP) app for Android, which has been considered the default Android SIP client by some vendors [31]. Sipdroid allows users to initiate free phone calls and provides multiple features, such as exchanging SMS messages and videos. Therefore, this small number of Android apps does not represent a statistically significant sample, which hints that the results may not be generalized to all Android apps. Nevertheless, due to the availability constraints of Android apps that are available and open-source and have both C/C++ and Java code, it has only been possible to consider this small number of Android apps. However, the selected apps have a range of different domains and real-world Android apps with reasonable and different sizes; and they are not tailored for the experiments. The results may also not be generalized to other applications other than Android apps, and applications written in languages other than Java and C/C++.

A threat to internal validity is related to the fact that the built call graphs maybe not accurately reflect the relationship between methods, hence the results may include false positives and false negatives. To mitigate this threat, state-of-the-art static analysis tools and methods that construct call graphs were used. For instance, on the Android Java side, FlowDroid has been used along with Spark to build a call graph for Android Java code. FlowDroid precisely models the Android components' lifecycle and callbacks. FlowDroid provides the Soot Pointer Analysis Research Kit (SPARK) to generate a call graph. However, there is a limitation of using FlowDroid, which is that FlowDroid does not fully support implicit or indirect control flow, such as intents or reflection. On the C/C++ side, the SeaDsa framework [101] that is based on the Data Structure Analysis (DSA) algorithm, has been adopted to construct a call graph that handles indirect calls via a function pointer. Moreover, some scenarios are not fully supported, such as multi-threading. Thus, further work and analysis are desirable.

5.8 Conclusion

This chapter introduced a general methodology to provide the source and sink pairs that are possibly reachable in polyglot programs using call graphs. We suggest two approaches to determine the reachability between sources and sinks. One approach applies the concept of the connected component using the UCCG graph, and the other approach utilizes the least common ancestor concept in the CCG graph. The methodology infers that using call graphs to determine the possibly reachable source and sink pairs can be beneficial, as it could decrease the number of all possible pairs dramatically, which helps to promptly render the results. SoS can be adopted by code reviewers to manually investigate possible reachable pairs. It also could be used by other static analysis tools, as described in Chapter 6 to apply more in-depth analysis techniques. We test the proposed methodology with Android apps, yet it could be used with different software and can catch vulnerabilities dispersed across multiple programming languages.

Chapter 6

BEFinder: Buffer Errors Finder for Modern Software

Buffer errors are still among the most frequent and dangerous types of exploitable vulnerabilities nowadays, despite all the efforts made during the past 30 years of studying and introducing solutions to solve such a problem. The technological advancement and software shifting make finding this type of vulnerability trickier in modern apps, such as Android apps, as discussed in Chapter 5. In this chapter, a methodology is introduced to meet the needs of modern software analysis. The introduced methodology is based on the reduced set of sources and sinks provided by SoS. This chapter aims to propose a methodology that performs taint analysis to examine the reachability between sources and sinks and looks for "sanitizer" or "validator" that validates the untrusted input. We implement the described methodology in a tool called *Buffer Error Finder* (BEFinder) and test it with Android apps.

6.1 Introduction

Buffer errors are among the dominant types of software security errors throughout history[36]. They continue to be the most frequent type of vulnerabilities in modern software, such as Android apps [45] [46]. Using static analysis security testing (SAST) tools helps proactively detect and eliminate buffer errors during the coding phase. However, the studied state-of-the-art SAST tools for buffer errors cannot effectively detect real-world vulnerabilities in modern apps, such as Android apps [45]. As discussed in Chapter 4, one of the reasons is that buffer errors detected in real-world Android apps are complex, but naïve SAST tools that perform lightweight analysis could not detect such type of issue [45]. Furthermore, more advanced state-of-the-art SAST tools that perform in-depth static analysis cannot analyze C/C++ code in Android apps because either the tools work with specific compilers or the tools do not scale well [45]. Besides, Android apps that include C/C++ code are likely to have Java/Kotlin code. However, SAST analysis tools that analyze buffer errors in C/C++ do not consider data that can cross from a different language in such analysis [45]. When looking at buffer errors in the wild, we find that the majority of the buffer errors accept input from untrusted sources [44]. Static taint analysis may be effective in detecting such a pattern of vulnerability. Thus, the buffer error detection problem is modeled as a reachability problem between the source (i.e., tainted input, including cross-language communications) and sink (i.e., buffer read/write operations). The analysis then examines the code to see whether the untrusted source has been *validated* before reaching the sink. Every sink type is associated with a set of constraints that need to be validated by the programmer. The methodology operates on the reduced source and sink pairs given by SoS. In this way, only sinks that can be possibly reachable by tainted input are considered for such analysis. This way, the methodology can work with software with multi-entry points; it can also consider data cross the Foreign Function Interface (FFI).

The proposed methodology is context, path, point-to, -sensitive and performs an interprocedural analysis. The approach is based on the concept of the Value Flow Graph (VFG) [178] that captures data dependencies in the source code, which we build on top of the Sparse Value Flow Graph (SVFG) [181]. This method is used to examine the reachability between sources and sink using a fine-grain representation of the source code. In a nutshell, the analysis starts at a source and sink pair that are reachable at the call graph level. The analysis then starts at the sink and propagate the analysis in a bottom-up fashion to look for "sanitizer" or "validator" points. Recall that in real-world buffer errors in Android apps, there is a lack of validation, meaning that the developer uses the untrusted data without validating the data, such as data size fits the allocated buffer. The validator is examined using a simple constraint-based analysis, which is performed to ensure the buffer operation safety. The analysis can be terminated at any time once the analysis found that the buffer is well managed, such that the developers ensure the safety of the buffer operation using a validator. A tool called BEFinder is implemented and tested to evaluate the methodology.

The contributions of this chapter are as follows:

- We model buffer errors as reachability problem between source and sink,

- We describe a taint analysis methodology that scales well and captures data and control dependencies using the SVFG,
- We propose solving buffer errors problem using a simple constraint-based approach,
- We implement BEFinder that works with any C/C++ source code, including Android apps.

The next section discusses the methodology we follow to detect buffer errors and build BEFinder in detail.

6.2 A Motivating Example

Figure 6.1 represents a toy example that shows a possible buffer over-write vulnerability that could be triggered by attackers. On the Java side, the function *decodeData* reads tainted data and the associated size from a file, and passes them as arguments to C++ code. The function *Java_ com_ example_ Decoder_ decodeData* is a function in C++ code that is called from Java code. This function gets *input* and *size* parameters that are tainted from the file. The function then passes those tainted parameters to *processData* function along with the calculated length of *input* (line 10). The *processData* function examines and compares *size* with *length* (line 14), if the tainted *size* value is greater than the *input* length, *size* value will be adjusted to be equal to *input* length (line 15). This way, the programmer ensures that *size* argument represents the length of *input*, since the size is tainted and may not represent the real size. The intent of the programmer is that s/he want to ensure that no more than the actual length of the input buffer will be read, hence preventing buffer over-read. Then, *input* and *size* are sent to the function *copyData*. Function *copyData* then allocates *buffer* (line 20) and copies the tainted data into *buffer* using memcpy function (line 21).

The *memcpy (void * destination, const void * source, size_ t num)* is a standard library function that copies a number of *num* bytes from the location pointed to by *source* to the buffer pointed to by *destination*.

In this example, two possible dangerous cases were neglected. (1) *size* can exceed the *BUFFER_ LEN*. Hence, buffer over-write can occur as the destination size may be less than the copied data. (2) *size* can be less than zero. This is particularly important since the *data* and the *size* arguments are tainted. In line 14, the check statement ensures that the size does not exceed the length of the *input*. If the size exceeds the length, the size will be adjusted to be equal to the length. This way, memory over-read would be prevented. However, this can happen when the size, which is tainted value, is a negative number, and hence this would pass the check in (line 14).

```
1   extern "C"
2   JNIEXPORT void JNICALL Java_com_example_Decoder_decodData
3   (JNIEnv *env, jobject obj, jobjectArray input, jint size) {
4       const char *rawInput ;
5       int len = env->GetArrayLength(input);
6       for (int i=0; i<len; i++) {
7           jstring string = (jstring) (env->GetObjectArrayElement(input, i));
8           rawInput = env->GetStringUTFChars(string, 0);
9       }
10      processData(rawInput, len , size);
11  }
12
13  void processData(const char *input, int len , int size) {
14      if (size > len)
15          size = len;
16      copyData(input, size);
17  }
18
19  void  copyData(const char *data, int size){
20      char  buffer [BUFFER_LEN];
21      memcpy (buffer, data, size );
22  }
```

Figure 6.1: An example of possible buffer over-write

6.3 Problem Statement

Definition 8. *Consider a program P that includes the following statement types: a source statement that accepts input from an untrusted source, and a sink statement that performs buffer read/write operation. A source-sink reachability problem for buffer error is the problem of checking that a value accepted at the source will flow into at least a sink without being validated properly.*

In this analysis framework, source statements can be any statement that reads tainted data, such as reading files, sockets, or other languages. While the sink statements read-/write data, such as standard library calls (e.g., memcpy and strcpy) or through accessing containers (e.g., array and struct). The analysis examines if the buffer operation at the sink statement does not trigger buffer error that leads to vulnerability. To decide if the sink triggers a buffer error, one needs to answer the following question: does the program include a path on which data flows from the source to the sink without going through a proper validation statement? A validation statement is a statement that validates if the input buffer is valid to perform safe buffer operation (e.g., valid as the size is appropriate).

The validation statement's goal is to ensure the safety of buffer operations since C/C++ lets the programmer manages the buffer operations. Nevertheless, every sink type might need a different type of validation. For instance, to access an array safely, the index should be less than the buffer size and more than or equal zero. While for standard library function call like strcpy, the programmer needs to validate that the destination size is large enough to contain the source data, including the terminating NULL character. The analysis model ensures that all the paths between a source and a sink are safe. If there is one path that is not safe, the analysis might conclude the presence of a vulnerability.

6.4 Analysis Framework Overview

Since the buffer error problem is reduced to the source-sink reachability problem, we adopt a demand-driven approach. A demand-driven analysis is used to minimize the number of analysis operations as much as possible to fit a larger program. In the demand-driven analysis for buffer errors, only a set of constraints of interest are investigated as to whether buffer access is safe. The analysis is conducted in a bottom-up fashion, where it starts at the sink statement and performs backward slicing for all the variables of interest in that statement. Once a validation statement is found that influences the sink statement, the analysis checks whether the validation statement ensures the safety of the buffer operation. The examination of validation statements is based on a set of constraints (Section 6.4.2 includes more details about the constraints). This way, the analysis can be halted if the developers control the buffer read/write operation (i.e., sink statement). Once that the paths from source to the sink are safe, the analysis concludes the safety of the operation; otherwise, it reports a problem or identifies the possible outcome of the buffer operations as unknown (when the required analysis is beyond the framework capabilities).

This chapter's static analysis approach uses the Intermediate Representation (IR) of the LLVM framework to perform the analysis. The IR is a low-level representation of a source code produced by compilers or virtual machines for optimization and translation purposes. The IR was chosen because it is independent of any programming language and can perform a static analysis that can be extended across different languages. Also, the IR of LLVM was used since it has many analysis capabilities. For instance, it encapsulates control and data dependence, which supports efficient and precise data-flow analysis. This section provides in-depth details about the analysis approach and the reasoning behind the chosen method.

6.4.1 Data and Control Dependencies

Data-flow analysis is required to trace a set of variables included in a sink statement. Data-flow analysis is an approach to collect information about the possible set of values of variables calculated at several points in a program [115] [148]. The classical data-flow approach captures information flows between different variables via propagating data-flow facts, which are the general properties of data at a program point, using the control-flow graph (CFG) [42]. This is conducted via a dense analysis where information is gathered at every program point, even if program points are irrelevant. However, because in buffer errors, there is only a subset of program statements is of interest, this methodology is inefficient, specifically when it comes to larger programs. One property of the IR of a program is the single static assignment (SSA) [165] [74], which encapsulates data-flow dependencies in such a way that facilitates fast and precise data-flow analysis. This makes detecting buffer errors to be more efficient compared to the classical data-flow analysis approach.

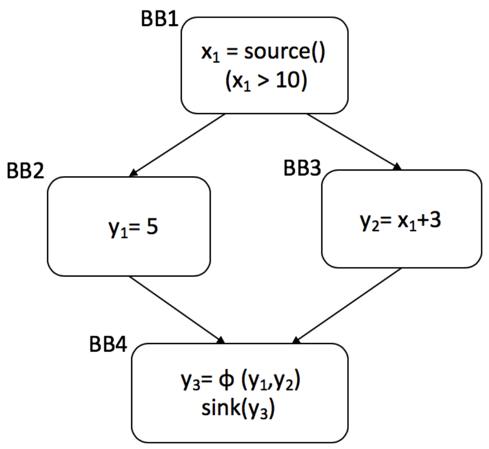
In the SSA form, the data-flow information is gathered using a sparse representation of the program (i.e., def-use and us-def chains [179]). In this sparse representation, data-flow information is captured directly from definition statements to uses. That is because data-flow information is associated with variable names, instead of storing a vector of data-flow facts for all variables for every program point. In the SSA form, every variable has a unique assignment (i.e., definition). In this way, variables are split into versions and every new assignment of variable results in a new version. The variable versions are referred to by the original name with a subscript that is a sequential number (see Figure 6.2). Therefore, the SSA form satisfies the dominance property where the variable definition dominates each use of a variable. In this way, each use of a variable version could be linked back to a unique definition.

In the SSA form, multiple control-flow paths can merge into a join node in the CFG. In consequence, different definitions of the same variable may reach the join node. The *phi* function (ϕ) merges different values of the same variable from different incoming paths of control flow. To illustrate this, consider the code example and its corresponding CFG in Figure 6.2a and Figure 6.2b. There is only one definition of x, while there are three definitions of y. Two definitions of y in each branch at BB2 and BB3, and one definition of y at the control flow merging point at BB4. The merging point (i.e., phi function) is a point of a program where there are two possible values of y depending on the result of the if conditional statement.

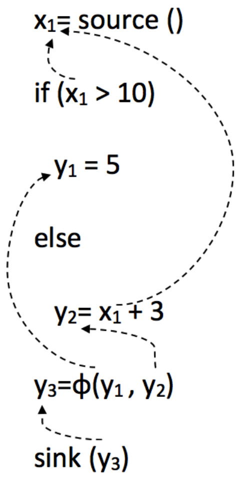
A use-def chain is a link from a variable use site to a variable definition site [179]. In contrast, a def-use chain is a link from a variable definition site to a use site for that

```
x= source ()
if (x > 10)
  y = 5
else
  y= x + 3
sink(y)
```

(a) Source code



(b) SSA in CFG



(c) Use-def chain

Figure 6.2: A toy example of a source code and the associated SSA form in CFG and the use-def links

variable. A definition site for a variable is a statement that assigns to the variable other variables. A use site is an operand of a statement. A use-def chain can be constructed between the use of a variable and a definition of the variable. For a use site, there is only one definition site that can be linked to. For instance, in Figure 6.2c both use sites of *x1* in [*if x1 > 10*] and [*y2= x1 +3*] are linked to one definition site which is [*x1= source()*]. Hence, data dependencies are the incoming operands of the instruction.

Control dependencies are the control predicates that need to be satisfied for a statement to be executed. We are only interested in control dependencies related to the value of interest, which can then be captured from the use-def chain. To clarify this, lets first review how the program is represented in the IR form. A program consists of a set of function definitions, and each function includes a set of basic blocks. A basic block is a sequence of instructions that ends with a terminator instruction that explicitly indicates the successor basic blocks (i.e., basic block to be executed after the current block is finished), such as a branch or function return instructions. The terminator instructions sometimes are associated with control predicates that indicate conditional expressions that alter the control flow. For instance, in Figure 6.2c, the control predicate [*if x1 > 10*] controls the value of [*y3*].

6.4.2 Safety Constraints

For every type of sink that may lead to buffer errors, there is a set of constraints to be examined that ensure the safety of a buffer operation. The analysis includes a simple constraint framework that contains the following elements:

6.4.2.1 Sink Set

Sink set is a finite set of sink types that may include potentially buffer read/write operations. For instance, copying data using standard library calls or directly assigning data via pointers arithmetic or array subscript within loops. Table 6.1 includes some examples of this component in column *Sink Type*. The full list of the safety constraints is given in Appendix H.

Generally, the sinks in this analysis are classified into three main categories as fallow:

- String operation function call
    1. String copy function call (e.g., strcpy, strlcpy)
    2. String concatenation function call (e.g., strcat, wcsncat)
    3. Formatted string output function call (e.g., sprintf, snprintf)
    4. Formatted string input function call (e.g., scanf, swscanf)
- Memory operation function call (e.g., memcpy, memmove)
- Container operation: write or read to/from a container, such as an array or struct.

Every sink type includes a number of **Variables**, which is a finite set of variables related to every sink where queries are initiated. This includes all the variables that need to be traced in such a sink statement. For instance, in *void * memcpy (void * destination, const void * source, size_t num)*, the variables that need to be traced are: *destination*, *source*, and *num*. Notice that the analysis distinguishes between two types of variables: a buffer and a number. In this example *destination* and *source* represent a buffer, while *num* is a number. The buffer type includes the number type, representing the buffer size and the buffer length that were traced and captured as well.

6.4.2.2 Constraints Set

This is a finite set of constraints for every sink. This includes the constraints that need to be checked to ensure the buffer operations safety. In Table 6.1, the *Constrain* column includes some examples. For instance, the analysis needs to know if the accessed index is between [0, size-1] for array access. Here is the set of the constraints implemented in BEFinder.

- **NULL checking**: because the analysis focuses on tainted data, tainted data can be anything, including NULL. So, when a program tries to copy tainted data that is NULL, a segmentation fault error can happen that might lead to a program crash. Hence, a constraint that checks against NULL looks for control predicates that check against NULL with the related data dependencies.
- **String source length compared to sink allocation size**: some function calls that perform string operations, such as *strcpy*, copy the source string into the sink buffer without doing any validation within the function or specifying the number of bytes to be copied from the source. Hence, the programmer needs to validate that the sink buffer (i.e., *Size(sink)*) can accommodate the source string by checking the length of the source string (i.e., *Len(source)*).
- **Length compared to sink allocation size**: this constraint applies to both string operations and memory operations function calls. Length here refers to the parameter given in a function call that specifies the number of bytes copied from the source buffer to the sink buffer. For example, *memcpy* has a parameter that represents the *Length*. If the length exceeds the sink allocation size, a buffer over-write occurs. To ensure that this constraint applies, the constraints solver checks to see if this argument is a constant; note that this constraint can be solved quickly because of the constant propagation optimization used in this analysis. If the *Length* is constant, the comparison can be done quickly by comparing it to the *Size(sink)*. If the *length* is a variable, then a validation point needs to be present. A validation point is a control predicate that influences both the execution and the values of the sink statement.
- **Length compared to source allocation size**: this constraint similar to the constraint *Length compared to sink allocation size*. However, when this violation happens, a buffer over-read occurs. Instead of comparing *Length* to the sink size, this constraint compare *Length* to the source size (i.e., *Size(source)*).

- **Length compared to source length**: Sometime a source can actually have a length (i.e., *Len(source)*). In this case, reading more than the supplied buffer's length can lead to a buffer over-read. Hence, this constraint checks to see if there is a validation point where the length is compared to the source length is adequate.

- **Source length and sink length compared to sink allocation size**: this constraint applies to string concatenation function calls, such as *strcat*, that concatenate the source string with the sink string. In the string concatenation function calls, the sink buffer allocation size *Size(sink)* should be larger than the source string length and destination string length combined.

- **Length and sink length compared to sink allocation size**: this constraint applies to string concatenation functions, such as *strncat* that provides the number of bytes to be copied. Similar to the above constraint, but this constraint checks the provided length rather than source length.

- **Length is positive**: if the Length argument is a variable based on tainted data. A possible error is that this variable might be less than zero, and hence a buffer error can happen. This is particularly true if the length has *int* type, and sometimes integer overflow in this argument can lead to buffer error.

- **Index within bound**: this constraint is associated with container operation, such as reading/writing from/to arrays. The index should be within the lower bound (zero in arrays) and the upper bound (the size of the buffer minus one in arrays) to have safe access to the buffer container.

- **Format argument is not tainted**: this constraint applies to format string function calls. Format string function calls provide a format string argument that instructs to read/write from/to memory. However, if this argument is tainted, it is possible that an arbitrary memory read or write can occur.

- **Format specifier number matches the number of provided arguments**: this constraint applies to format string function calls. The number of specifiers provided in the format argument should match the provided arguments. For instance, *printf ("The value of n: %d, and its address: %p", n);*, this function has two specifiers *%d* and *%p* , but only one argument is provided which is *n*. Hence, buffer over-read may happen as printf() will read data from the stack that do not belong to the function call.

- **Format buffer size is controlled**: this constraint applies to format string function calls. Some format string functions provide a parameter to control the number of bytes to be read or written, and other format string functions do not, but control the number of byte through the width field specifies. In either case, this constraint checks whether the function provides a size that does not exceed the buffer size.
- **String is NULL terminated**: this constraint ensures that tainted strings are NULL terminated.

6.4.2.3 Constraints Solver

The constraint solver is a custom solver to evaluate the buffer operations against the constraints. The constraints solver first provides a mapping between the set of sinks to the set of constraints. For every type of sink, there is a set of constraints. Different types of sinks might share some constraints. The solver depends on the sliced information to check if the constraint is met. The constraint solver looks for missing checks, yet, it does not solve the issue if there is a check but was wrongly performed, such that the buffer size was mistakenly miscalculated, which requires a more advanced solver. In this implementation, when the solver checks the constraint $n < Len(s)$, it examines the control predicts to see if one operand congruent to n and the other operand congruent to *Len(s)*. This relationship can be linear or nonlinear. The analysis concludes that the buffer operation is safe if it meets the given constraints. Once constraint violations are detected, the solver would report a potential memory error. If the analysis could not gather enough information to conclude whether Safe or Unsafe, the analysis would be resolved into Unknown.

6.4.3 The Demand-Driven Analysis Algorithm

The approach starts the analysis by performing slicing at the sink statement based on the SVFG. Program slicing is a technique for tracing only relevant statements of a program [202]. This method traces a set of statements that may directly or indirectly influence a value at a point of interest, referred to as a slicing criterion. A slicing criterion is a pair *<s, V>*, where s is a program statement, and V is a subset of program variables. In the motivating example shown in Figure 6.1, *memcpy (buffer, data, size);* is the program statement, while *buffer*, *data*, and *size* are a set of variables that need to be traced to solve the buffer error problem. The goal of performing such slicing is to inquire about the safety constraints at the sink that are propagated bottom-up along with the control flow.

Table 6.1: Buffer errors safety constraints examples

Sink Type	Variables	Constraints
memcpy(d, s, n)	d, s, n	n < (Size(s) & (Size(d)) n < (Len(s)) n >= 0 d != NULL s != NULL
strcpy(d, s)	d, s	Size(d) > Len(s) d != NULL s != NULL
array[i]	i	0 <= i < Size(array)

The analysis starts by analyzing the source and sink pair given in the JSON file. The main goal is to extract all the sources that can be reaching a specific sink. This way, the analysis would analyze a sink once, and it examines multiple sources at once. For example, a sink can be reachable by 16 sources, but the sink needs to be sliced only once. Once a slice is created, the analysis determines if any given sources appear in such a slice. If a source happens to be not in the slice, the source is discarded. If there are absolutely no sources that can reach the sink, the sink is discarded, and the next sink is examined. Only sinks that are reachable by at least one source are analyzed. The JSON file provides the flowing: the *source enclosing function name*, which is the function that the source resides in, the *source function name*, which is the function call that accepts input from untrusted sources, the *line number* and the *character position* that the source call belongs to. Similar information is provided for the sinks.

The analysis first extracts the IR instructions and the basic block associated with the source and the other with the sink. Then, the analysis extracts a slice that starts at the sink and goes all the way up to the source; the analysis reason if the source belongs to such a slice. Once a reachable pair is identified, the analysis solves the constraints. To do so, the initial basic block that is passed to be analyzed first is the one that has a sink and moves bottom-up to evaluate the control predicates using given constraints at each basic block. However, if the constraints could not confirm the buffer operation's safety, the analysis continues to perform inter-procedural analysis, analyzing data across functions.

The inter-procedural analysis is performed as follows: once a call site is detected in a basic block by the analyzer, the query is propagated to the called function; this happens only if the called function is related to the query, which is known from the slice. A work-list is used for both processed functions and basic blocks. This helps to handle recursion for processed functions and loop for processed basic blocks. While propagating the query against constraints, the relevant information is collected, such as the allocated buffer's size or if there is any buffer length calculation to resolve the query. Meanwhile, the query is propagated to all predecessors of the analyzed basic block once the validation is not found. Finally, the analysis terminates when all constraints have been met, or the analysis reaches the source of untrusted input that could be exploited.

When such information is retrieved and a violation is found, the analysis can be halted with unsafe buffer operation outcomes. This design was used since, upon observation, developers usually validate the read/write buffer operations before the read/write buffer operations occur. Hence, the bottom-up analysis and examining validation statements along the path can find any violation and terminate the constraint solving quickly as the desired information is likely to be gathered soon. The slicing can ensure that this unsafe operation is affected by untrusted input (i.e., taint analysis). The analysis can run in parallel to solve every query independently.

6.5 Implementation

We implement BEFinder to conduct static analysis using a combination of techniques. It was implemented in C++ using the LLVM framework version 8. BEFinder reads a JSON file that includes a set of sources and sinks to be tested. The sources and sinks include information such as the enclosing method names and the line number. Then the analysis starts at the sinks applying the algorithms and techniques discussed in this chapter. BEFinder builds the SVFG based on the SVFG by the SVF framework and the def-use chins by the LLVM. This provides a scalable and precise value-flow construction, including point-to analysis in inter-procedural fashion for C/C++ programs. Finally, the warnings are written into a JSON file, including the results that could be safe, unsafe, or unknown. A safe outcome indicates that no violation is detected, and the buffer operation is likely to be safe. Unsafe means there is at least one volition that was found. An unknown outcome refers to the fact that more complex analysis needs to be conducted to make a concrete conclusion. BEFinder data is available online [47].

6.6 Experiment Setup

To evaluate BEFinder, we run it on the set of Android apps that were discussed in the previous chapter (See Section 5.5 for more details). We test four Android apps: AdAway, Orbot, ProxyDroid, and Sipdroid. The goal here is to check if the tool can find buffer error vulnerabilities. The projects' sizes are ranging from 35 KLOC to 145 KLOC. The analysis was conducted on a MacBook Pro with a 2.2 GHz Intel Core i7 processor and 16 GB of memory. Every project includes a potential unsafe source-sink pair JSON file and the compiled C++/C code into LLVM IR, BEFinder examines every pair and produces the results into a JSON file. Once the pair were found to include a potential vulnerability, a warning is generated. Table 6.2 presents the tested apps in this experiment.

Table 6.2: Tested Android apps

App Name	Version	KLOC
AdAway	v4.3.2-15-g72e024b1	131K
Orbot	16.1.2	145K
Sipdroid	9067a3c	37K
ProxyDroid	fc94aec	35K

6.7 Result and Discussion

The evaluation of the methodology and BEFinder consists of reachability analysis accuracy between sources and sinks and buffer error detection rate. The first assessment aims to validate whether using the SVFG can help spot the reachable source and sink. We conduct the experiment by running BEFinder without solving any constraint, but only to determine the reachability. Then, we randomly select instances and manually inspect them to determine the correctness of the report.

6.7.1 Reachability Analysis

To evaluate the reachability between sources and sinks, we randomly select and manually inspect 20 source and sink pairs of each Android app that are flagged as reachable. The inspection involves studying each pair to ensure that a data accepted at the source travels to the sink. For every source, the tainted argument or the tainted return value were traced through the source code. If there is an executable path from the source statement to the sink statement and the tainted data that was accepted at the source statement propagated to the sink statement, then this means that the untrusted source is truly reachable to the sink. Finding the executable path between the source and the sink is simple when the source and the sink belong to the same function.

Similarly, checking the tainted data propagation from the source to the sink is straightforward when the source and the sink belong to the same function. The manual inspection takes more time and effort when the source and the sink belong to different functions or source code files. In this case, we investigate the full call graph that was generated by SoS to draw a partial call graph using *grep, cat, awk* and other Unix text pattern scanning tools to comprehend the inter-procedural relations between the functions in question. We then inspect the source code files to find an executable path from the source to the sink and track the tainted data propagation status to all the locations where the data is used. Tainted data may propagate directly or indirectly to the sink. Direct propagation occurs when the tainted data copied as is to the sink, while indirect propagation happens when the tainted data becomes a part of another variable or object that travels to the sink.

The analysis results presented in Table 6.3 demonstrates the reachability analysis precision when tracing value flow using the SVFG. *Pair* column shows the number of pairs that needed to be sliced. Note that each pair includes more than one slice criteria; for instance, the standard function call *memcpy* includes three slices, as discussed earlier. *Tested* column shows the number of pairs that have been found and located in the code. This happens because the initial pairs were acquired using an external parser with SoS using the source code, but in BEFinder, the analyzed code is the intermediate representation. Sometimes, the LLVM omits some function calls for optimization purposes. *Reachable* column presents the number of pairs that analysis renders as reachable in the SVFG. *TP* column shows the number of instances that are true positive, meaning that they are reachable. On the other hand, *FP* column shows the number of false positive instances, meaning that they were wrongly tagged as reachable since, after examining the instance, they were found that they are not reachable. Finally, the *Precision* column demonstrates the percentage of relevant reachability results. The precision is measured as the number of true positive results in the total number of warnings.

Reachability analysis results show that the precision is at least 65%. It was noticed through the manual inspection is that the precision increases when the function body is smaller and less complex. The main reason we have such precision is the used point-to analysis algorithm, Andersen's points-to analysis that only provides may-point-to information.

Table 6.3: Reachability analysis using the SVFG

App Name	Pairs	Tested	Reachable	Examined	TP	FP	Precision
AdAway	7,867	7,414	3,114	20	13	7	65%
Orbot	238,455	28,258	13,419	20	16	4	80%
ProxyDroid	4,551	4,397	2,042	20	20	0	100%
Sipdroid	2189	1,965	1,223	20	15	5	75%

For instance, in most of the cases of the false positive, the source and the sink are accessing the same memory location; however, it does refer to the fact that data flow from the source to the sink. To overcome such a problem, one should adopt a more precise point-to analysis. However, a more precise point-to analysis may not scale well. Nevertheless, the results are promising in applying such an approach with other similar problems, such as information leak or SQL injections [180]. It is worth noting that the results may have false negative as renderings such pairs are not reachable, but in fact, they are reachable. However, this needs a complete ground truth that we do not have.

6.7.2 Buffer Errors Detection

The second part of the analysis evaluates BEFinder detection rate of buffer errors. We conduct the experiment by running BEFinder and then evaluating all the *Unsafe* warnings by manually inspecting each warning. The manual inspection involves reachability analysis described in Section 6.7.1, and examines the generated constraint violations reported in the warning. For a particular violation, we perform a manual review to check if the violation leads to vulnerability, and the results are sound. This process involves comprehending the surrounding source code. To inspect the violation, we start the analysis at the sink, and all the program statements are examined until reaching the source to look for validation points. This is done in intra-procedural and inter-procedural fashion, depending on the source and sink pair if they are located in the same function or not.

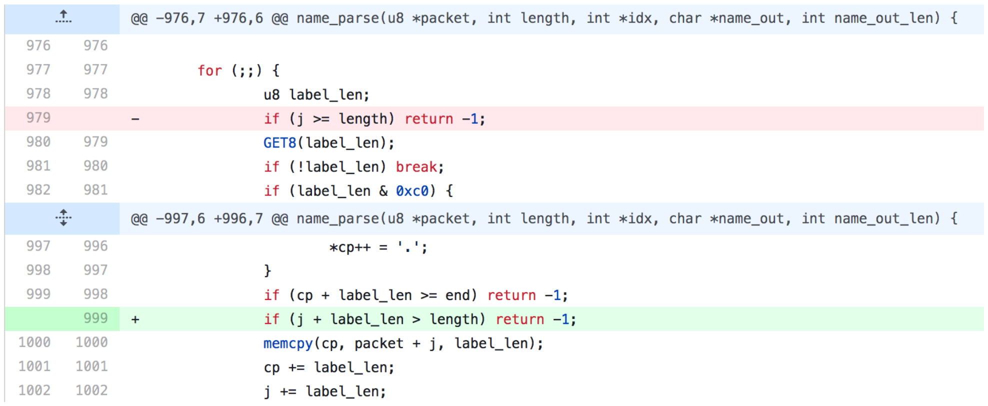
Table 6.4 illustrates the results. *Reachable* column refers to the total number of reachable pairs determined by the first experiment. *Unknown* column shows the number of pairs that the analysis cannot determine whether it is safe or unsafe; hence more analysis capabilities are needed to resolve those pairs. While *Safe* column refers to the number of source and sink pairs that are safe, meaning that the data that travel from the source to the sink has been validated well. On the other hand, *Unsafe* column shows a lack of validation between the source and sink. Then, *TP* column, *Likely TP* column, *FP* column, and *Precision* column are all related to the *Unsafe* column.

Table 6.4: Buffer error detection rate by BEFinder

App Name	Reachable	Unknown	Safe	Unsafe	TP	Likely TP	FP	Precision
AdAway	3,114	768	2,316	30	17	8	5	83%
Orbot	13,419	4,012	9,286	93	-	93	0	100%
ProxyDroid	2,042	190	1,840	12	10	-	2	83%
Sipdroid	1,223	530	693	0	-	-	-	-

The *TP* column shows the vulnerabilities that have CVE-ID, patched by the developers, or we could implement exploits to trigger the flaw. The *Likely TP* column shows the warnings that we inspect manually, and from the manual review, we find a possible execution path that can trigger the issue. Also, the issue was reported to the developers but not confirmed yet. The precision is measured as the number of true positive warnings that genuinely show the problem's presence to the total number of warnings. The analysis results indicate that precision is 84% or higher. This percentage was observed in the ProxyDroid project. Also, it was observed that the false positives are because the app uses wrapper functions to perform the validation without using the standard libraries, and here where the analysis fails. Missing functions that allocate buffer or measure the buffer length can lead to imprecise results. Thus, to further enhance the results, a helper tool might be needed, such as a tool that understands these function signatures.

Two generated warnings were detected in the past by independent individuals, and they have CVE-ID. For instance, the CVE-2016-10195 and CVE-2016-10196 are all previously discovered vulnerabilities in the libevent library that is integrated with ProxyDroid app are flagged by BEFinder. Figure 6.3 shows the CVE-2016-10195 vulnerability that occurs when copying data using *memcpy* function, which indicates that the developer did not validate whether the length of *label_ len* given does not exceed the source buffer *packet+j*, which may lead to a buffer over-read. BEFinder associates six constraints with *memcpy* (See Table 6.1). Yet, BEFinder reported one violation, which is *n >(Len(s)*. The recall indicates the percentage of total relevant results correctly flagged by the analysis algorithm. Since we do not have a ground truth about all the possible vulnerabilities in such apps, we cannot measure the recall. In this experiment BEFinder was not compared with other tools that detect buffer errors for the main reason, which is that BEFinder looks at the cross-language communications where there is no open-source tool that does that (See Chapter 4).



```
@@ -976,7 +976,6 @@ name_parse(u8 *packet, int length, int *idx, char *name_out, int name_out_len) {

        for (;;) {
                u8 label_len;
-               if (j >= length) return -1;
                GET8(label_len);
                if (!label_len) break;
                if (label_len & 0xc0) {
@@ -997,6 +996,7 @@ name_parse(u8 *packet, int length, int *idx, char *name_out, int name_out_len) {
                        *cp++ = '.';
                }
                if (cp + label_len >= end) return -1;
+               if (j + label_len > length) return -1;
                memcpy(cp, packet + j, label_len);
                cp += label_len;
                j += label_len;
```

Figure 6.3: Code snippet shows CVE-2016-10195

6.8 Conclusion

In this chapter, we introduce a demand-driven analysis method for buffer error detection in modern software. This technique's main goal is to analyze C/C++ code and understand communication with Java to reveal buffer errors in Android apps. We evaluate BEFinder, and the results demonstrate that the method is useful for buffer error detection in Android apps with a false positive rate of around 16%. In addition to the buffer errors, the analysis can be applied to detect other critical vulnerabilities related to the reachability problem, such as the IVR class of vulnerability. Nevertheless, some enhancements can be done to improve the tool performance to reduce the false positive rate of the reachability analysis by adopting a better point-to analysis and reducing the false positive rate of the generated warnings by adopting a more advanced constraints solver.

Chapter 7

Conclusion

SAST tools are one of the most effective ways to deal with software vulnerabilities. SAST tools help programmers discover software vulnerabilities during the coding phase, and hence, such an issue can be eliminated from the code, preventing any further damages. For that reason, studying static analysis tools and methods has been an important direction of research in cybersecurity. Nevertheless, the technologies' progression yields a shift in software, which contributed to rising software vulnerabilities. However, it is not known whether SAST tools are adapting to such changes. Hence, this thesis takes a step in the direction of identifying the features needed in the SAST to be able to address software vulnerabilities in modern software. The thesis depends on the empirical evidence acquired through research to perceive how state-of-the-art SAST tools operate on real-world projects and how software vulnerabilities manifest in the real world. We find that buffer errors that belong to the IVR class of vulnerability dominate modern software. We then introduce two methodologies and proof-of-concept tools that give some directions on how to detect the most significant issues in software vulnerabilities: buffer errors. The next section summarizes the insights of his thesis.

7.1 Summary and Contributions

In Chapter 3, we conduct an empirical study to investigate the precision of the state-of-the-art SAST tools. The research assumption is that the lines of code that indeed include software defects, as of the outcome of SAST tools, have been noticed and fixed by the project team. The empirical study aims to determine the SAST tools' precision using the warnings' observations over time.

More specifically, the contributions of our study are as follows:

- We first examine the distribution of the different types of SAST-produced warnings through time.
- We investigate the number of the warnings remain over time based on the warning type.
- We study the decay rate for warnings belonging to different categories and produced by various SAST tools.
- We adopt the *cregit*$^+$, a novel tool for version control tracking to measure warning evolving.
- Finally, we investigate the distribution of real-world vulnerability categories removed by the project team in the studied projects to measure our results consistency.

In Chapter 4, we carry out another empirical study to look at SAST tools from another perspective. In this study, we start the investigation from real-world vulnerabilities that occur in Android apps, as one example of modern software that has been discovered by individuals and reported to vulnerabilities databases. The main goal is to measure the recall of SAST tools when they work with modern software.

The contributions of this study are as follows:

- We first measure the frequency of the software vulnerabilities in Android apps, and we determine the most dominant type of vulnerabilities in such apps as buffer errors.
- We evaluate the state-of-the-art SAST tools to detect buffer errors regarding the real-world vulnerabilities happening in Android apps.
- We describe the characteristic of buffer error vulnerabilities observed in Android apps.
- We describe the required features of such SAST to detect such a pattern of vulnerability.

In Chapter 5, we propose a general methodology that can help static taint analysis tools. This methodology tackles two aspects of the complexity of modern software. One aspect is that one vulnerability can be scattered across different languages in a single program, making the analysis harder to achieve as it involves multi-languages understanding and a larger code. The second aspect is that the number of sources and sinks is significant and increasing over time, which can be challenging for taint analysis to cover such a large number of sources and sinks. Hence, we introduce SoS that filters out the source and sink pairs that likely do not have feasible paths.

The main contributions of this study are as follows:

- We highlight the necessity for a general methodology to quickly examine the reachability between all possible source and sink pairs, including pairs that cross-languages,
- we use the call graph construct to filter out the likely unreachable source and sink pairs,
- We illustrate two approaches based on a call graph to measure the reachability,
- We implement SoS and evaluate the two described approaches.

Chapter 6 focuses on discovering buffer errors that occur in modern software. The method performs taint analysis to examine the reachability between sources and sinks and look for "sanitizer" or "validator" that validates the untrusted input. We implement a static analysis security testing tool called BEFinder to carry out such analysis.

The contributions of this study are as follows:

- We model buffer errors as reachability problem between source and sink,
- We propose a slicing methodology that scales well and captures data and control dependencies precisely using SVFG,
- We solve the buffer errors problem using a simple constraint-based approach,
- We implement BEFinder that works with any C/C++ programs, including Android apps.

7.2 Future Work

The research findings discussed in this thesis offers a basis for future work. This section describes several potential research directions that involve studying both empirical and theoretical approaches.

7.2.1 Recall and Precision of SAST Tools

This thesis emphasizes the significance of studying real-world software and vulnerabilities outside any controlled environment to investigate SAST tools' performance in terms of recall and precision. The empirical evidence designates that we can use SAST tools as a quality measure. Since this is a significant indication, more empirical studies are needed; this may also refer to the fact that perceived false positive warnings might not necessarily be false positives. Hence, more empirical studies need to be conducted in this aspect to gain such understanding. Furthermore, understanding the false negatives by SAST tools is an important research direction that needs to be frequently investigated on a larger scale to define more limitations and continue improving existing tools.

7.2.2 SAST Approaches

This thesis finds that the IVR types of security vulnerabilities are common in modern software. Existing taint analysis tools improve the inference algorithms that infer taint information. However, a significant challenge we find is an inherited problem of the growing software that needs to be addressed before applying the taint analysis method: the exponentially growing number of sources and sinks in software. We describe some approaches that can be expanded to multiple-language software to cover the cross-language communications and many source and sink pairs. This is the first research study that describes the problem and provides heuristic approaches to the best of our knowledge. Further studies of the call graph properties may help build a more robust approach, avoid false positives and negatives. This can be acquired by studying call graph properties to extract facts about how data-flow information usually travels among functions/methods.

In terms of the buffer error detection, our study shows promising results for the described approaches. However, our practical experience with the SAST tools highlights another aspect of the problem and open more research questions. For instance, what are the practical approaches to validate the generated warnings? Should this be up to the developers to decide? The developers may have a various understanding of how vulnerabilities occur, and one security vulnerability may not be obvious for some developers. Since buffer errors happen under the umbrella of IVR, we need to think of other techniques that may be used to validate the SAST warnings—for example, integrating dynamic and fuzzing methods to validate the produced warnings. Fuzzing methods can be used to inject different inputs, and once the program execution reaches the vulnerable point, dynamic analysis can test the concrete values against the safety constraints.

References

[1] CERT . Web page. http://www.cert.org/.

[2] Clang Static Analyzer. Web page. https://clang-analyzer.llvm.org/.

[3] CVE - Common Vulnerabilities and Exposures . Web page. https://cve.mitre.org/.

[4] CVSS - Common Vulnerability Scoring System . Web page. https://www.first.org/cvss/v2/guide,.

[5] CWE - Common Weakness Enumeration: A community-developed list of software hardware weakness types. Web page. https://cwe.mitre.org/.

[6] Dr.memory: Memory debugger for Windows, Linux, and Mac. Web page. https://drmemory.org/.

[7] Fix typing rule for Math.sign . Web page. https://github.com/v8/v8/commit/f9ee4f19b4d0c35088e557952e8e842f31307a7b#.

[8] JVN - Japan Vulnerability Note. Web page. https://jvn.jp/en/.

[9] List of free and open-source Android applications. Web page. https://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications.

[10] NVD - National Vulnerability Database . Web page. https://nvd.nist.gov/.

[11] Parasoft C/C++test. Web page. https://www.parasoft.com/products/ctest.

[12] Parasoft Insure++: Runtime memory leak detection and memory debugging for C and C++ applications. Web page. https://www.parasoft.com/products/insure.

[13] Peach fuzzing platform. Web page. https://www.peach.tech/.

[14] Perl: Buffer overflow — GLSA 200711-28. Web page. https://security.gentoo.org/glsa/200711-28.

[15] perlsec: Perl security. Web page. https://perldoc.perl.org/perlsec.html.

[16] PVS-Studio Analyzer. Web page. https://www.viva64.com/en/pvs-studio/.

[17] Rough Auditing Tool for Security (RATS). Web page. https://security.Web.cern.ch/security/recommendations/en/codetools/rats.shtml/.

[18] Security enhancements in Android 1.5 through 4.1. Web page. https://source.android.com/security/enhancements/enhancements41.

[19] SecurityFocus. Web page. http://www.securityfocus.com/.

[20] Smashing C++ VPTRS. Web page. http://phrack.org/issues/56/8.html.

[21] Software assurance. https://us-cert.cisa.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf.

[22] SPIKE fuzzing platform. Web page. https://resources.infosecinstitute.com/fuzzer-automation-with-spike/.

[23] Splint- annotation-assisted lightweight static checking. Web page. http://www.splint.org/.

[24] The Frama-Clang plugin. Web page. http://frama-c.com/frama-clang.html.

[25] TIOBE index. Web page. https://www.tiobe.com/tiobe-index.

[26] Valgrind. Web page. http://valgrind.org/.

[27] Verifying app behavior on the Android runtime (ART). Web page. https://developer.android.com/guide/practices/verifying-apps-art.

[28] Pax address space layout randomization (ASLR). 2003. https://pax.grsecurity.net/docs/aslr.txt.

[29] The Heartbleed bug. Web page, 2014. http://heartbleed.com/.

[30] Static source code analysis tools for C. Web page, 2014. https://spinroot.com/static/.

[31] Sipdroid, 2015. voipvoip.com/sipdroid.

[32] Application fundamentals. Web page, 2016. https://developer.android.com/guide/components/fundamentals.html.

[33] HPE security research cyber risk report 2016. Web page, 2016. https://www.thehaguesecuritydelta.com/media/com_hsd/report/57/document/4aa6-3786enw.pdf.

[34] Programming languages, their environments and system software interfaces — C secure coding rules. Web page, 2016. https://www.iso.org/standard/61134.html.

[35] JLang: Ahead-of-time compilation of Java programs to LLVM. Web page, 2018. https://github.com/polyglot-compiler/JLang.

[36] 2019 CWE top 25 most dangerous software errors. Web page, 2019. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[37] Android NDK native APIs. Web page, 2019. https://developer.android.com/ndk/guides/stable_apis.

[38] 2020 open source security and risk analysis report. Web page, 2020. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf.

[39] David A. Wheeler. Flawfinder. Web page. http://www.dwheeler.com/flawfinder/.

[40] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupe, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni 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.

[41] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Automating and optimizing detection of Android native code malware variants. *Computers & security*, 65:230–246, 2017.

[42] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[43] Bushra Aloraini. An empirical study of security warnings from static analysis tools. Web page. https://github.com/BushraAloraini/SATs.

[44] Bushra Aloraini. Evaluating state-of-the-art free and open source static analysis tools against buffer errors in Android apps. Web page, 2017. https://github.com/BushraAloraini/Android-Vulnerabilities.

[45] Bushra Aloraini and Meiyappan Nagappan. Evaluating state-of-the-art free and open source static analysis tools against buffer errors in Android apps. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 295–306. IEEE, 2017.

[46] Bushra Aloraini, Meiyappan Nagappan, Daniel M German, Shinpei Hayashi, and Yoshiki Higo. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software*, 158:110427, 2019.

[47] Bushra Alorini. Befinder. Web page, 2020. https://github.com/BushraAloraini/BEFinder.

[48] Bushra Alorini. SoS. Web page, 2020. https://github.com/BushraAloraini/SoS.

[49] Bushra Alorini. Sources and sinks for buffer errors. Web page, 2020. https://github.com/BushraAloraini/Sources-Sinks-for-Buffer-Errors.

[50] James P Anderson. Computer security technology planning study. Technical report, Anderson (James P) and CO., 1972.

[51] Patroklos Argyroudis and Chariton Karamitas. Exploiting the jemalloc memory allocator: Owning firefox's heap. *Blackhat USA*, 2012.

[52] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM Sigplan Notices*, 49(6):259–269, June 2014.

[53] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 332–341, May 2005.

[54] Itzhak Avraham. Popping shell on A(ndroid)RM devices. *Blackhat USA*, 2011.

[55] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[56] Mohammad R Azadmanesh, Michael L Van De Vanter, and Matthias Hauswirth. Language-independent information flow tracking engine for program comprehension tools. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 346–355. IEEE, 2017.

[57] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[58] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[59] Michael A Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[60] Larry Boettger. The Morris worm: how it affected computer security and lessons learnd by it. Web page, 2000. http://people.cs.vt.edu/~kafura/cs6204/Readings/Context-Problems/MorrisWorm.htm.

[61] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Android collusive data leaks with flow-sensitive DIALDroid dataset. https://github.com/dialdroid-android/DIALDroid.

[62] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. IKOS : A framework for static analysis based on abstract interpretation. In *International Conference on Software Engineering and Formal Methods*, pages 271–277. Springer, 2014.

[63] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1), 2009.

[64] Raymond Chen. *The Old New Thing: Practical Development Throughout the Evolution of Windows*. Addison-Wesley Professional, 2006.

[65] Brian V Chess. Improving computer security using extended static checking. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 160–173. IEEE, 2002.

[66] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A Theodore Markettos, J Edward Maste, Robert Norton, Stacey Son, et al. CHERI JNI: Sinking the Java security model into the C. *ACM SIGARCH Computer Architecture News*, 45(1):569–583, 2017.

[67] Tzicker Chiueh and FuHau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417. IEEE, 2001.

[68] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 332–343. IEEE, 2016.

[69] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. BegBunch: Benchmarking for C bug detection tools. In *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems*, DEFECTS '09, pages 16–20, 2009.

[70] Cristina Cifuentes and Bernhard Scholz. Parfait: designing a scalable bug checker. In *Proceedings of the 2008 workshop on Static analysis*, pages 4–11. ACM, 2008.

[71] Onur Cinar and Grant Allen. *Pro Android C++ with the NDK*. Springer, 2012.

[72] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[73] Di Cui, Ting Liu, Yuanfang Cai, Qinghua Zheng, Qiong Feng, Wuxia Jin, Jiaqi Guo, and Yu Qu. Investigating the impact of multiple dependency structures on software defects. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 584–595, Piscataway, NJ, USA, 2019. IEEE Press.

[74] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control

dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[75] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology*, 51(10):1469–1484, 2009.

[76] Gabriel Díaz and Juan Ramón Bermejo. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and Software Technology*, 55(8):1462–1476, 2013.

[77] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX security symposium*, volume 31, page 3. San Francisco, CA;, 2011.

[78] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, 2003.

[79] Will Dormann. Finding Android SSL vulnerabilities with CERT Tapioca. Web page, Sept 2014. https://insights.sei.cmu.edu/cert/2014/09/-finding-android-ssl-vulnerabilities-with-cert-tapioca.html.

[80] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, 2014.

[81] Nigel Edwards and Liqun Chen. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 183–194. ACM, 2012.

[82] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, July 2008.

[83] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[84] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[85] Hiroaki Etoh. ProPolice: GCC extension for protecting applications from stack-smashing attacks. *IBM*.

[86] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes*, 19(5):87–96, December 1994.

[87] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.

[88] Justin N Ferguson. Understanding the heap by breaking it. *Black Hat USA*, pages 1–39, 2007.

[89] National Security Agency Center for Assured Software. On analyzing static analysis tools, 2011. https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf.

[90] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002.

[91] Frama-C Software Analyzer. Web page. http://frama-c.com/index.html.

[92] Adam P Fuchs, Avik Chaudhuri, and Jeffrey Foster. SCanDroid : Automated security certification of Android applications. *Technical report, University of Maryland*, 10(November):328, 2009.

[93] Michael Furr and Jeffrey S Foster. Polymorphic type inference for the JNI. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 309–324, Berlin, Heidelberg, 2006. Springer-Verlag.

[94] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 345–354, New York, NY, USA, 2003. ACM.

[95] Daniel M German. cregit. Web page. https://github.com/cregit/cregit,.

[96] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao 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*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[97] Rigel Gjomemo, Phu H Phung, Edmund Ballou, Kedar S Namjoshi, V N Venkatakrishnan, and Lenore Zuck. Leveraging static analysis tools for improving usability of memory error sanitization compilers. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 323–334, Aug 2016.

[98] Guang Gong. Exploiting heap corruption due to integer overflow in Android libcutils. *Black Hat USA*, 2015.

[99] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS*, volume 15, page 110, 2015.

[100] Jordan Gruskovnjak and Aaron Portnoy. Stagefright: Mission accomplished?, 2015. `http://blog.exodusintel.com/2015/08/13/stagefright-mission-accomplished/`.

[101] Arie Gurfinkel and Jorge A Navas. A context-sensitive memory model for verification of C/C++ programs. In *International Static Analysis Symposium*, pages 148–168. Springer, 2017.

[102] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 232–241, New York, NY, USA, 2006. ACM.

[103] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 22nd USENIX Security Symposium*, pages 49–64, 2013.

[104] Roee Hay and Avi Dayan. Android keystore stack buffer overflow. Web page, 2014. `https://www.exploit-db.com/docs/english/33864-android-keystore-stack-buffer-over%EF%AC%82ow.pdf`.

[105] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *International SPIN Workshop on Model Checking of Software*, pages 235–239. Springer, 2003.

[106] Thomas Hofer. Evaluating static source code analysis tools. Master's thesis, School of Computer and Communications Science, Switzerland, 2010.

[107] Gerard J Holzmann. UNO: Static source code checking for user-defined properties. In *6th World Conf. on Integrated Design and Process Technology*, number June, pages 26–30, 2002.

[108] Gerard J Holzmann. Code inflation. *IEEE Software*, 32(2), 2015.

[109] Keman Huang, Jia Zhang, Wei Tan, and Zhiyong Feng. An empirical analysis of contemporary Android mobile vulnerability market. In *2015 IEEE International Conference on Mobile Services*, pages 182–189, June 2015.

[110] IDC Research Inc. Smartphone Market Share. Web page, May 2020. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[111] Matthieu Jimenez, Mike Papadakis, Tegawendé F Bissyandé, and Jacques Klein. Profiling Android vulnerabilities. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 222–229. IEEE, 2016.

[112] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[113] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[114] Sean Michael Kerner. Heartbleed SSL flaw's true cost will take time to tally. Web page, 2014. http://www.eweek.com/security/heartbleed-ssl-flaw-s-true-cost-will-take-time-to-tally.

[115] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[116] Sunghun Kim and Michael D Ernst. Prioritizing warning categories by analyzing software history. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 27–27. IEEE, 2007.

[117] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang Moo Choe. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology*, 52(2):210–219, 2010.

[118] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 563–573. IEEE, 2016.

[119] Goh Kondoh and Tamiya Onodera. Finding bugs in Java native interface programs. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118, 2008.

[120] Kendra Kratkiewicz and Richard Lippmann. A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, pages 500–265, 2005.

[121] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. Towards efficient, multi-language dynamic taint analysis. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 85–94, 2019.

[122] Daniel Kroening and Michael Tautschnig. CBMC- C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[123] James A. Kupsch, , and Barton P. Miller. Why do software assurance tools have problems finding bugs like Heartbleed?, 2014.

[124] Skybox Research Lab. Vulnerability and threat: Analysis of current vulnerabilities, exploits and threats in play. Web page, 2018. https://lp.skyboxsecurity.com/WICD-2018-02-Report-Vulnerability-Threat-18_Asset.html.

[125] Patrik Lantz and Bjorn Johansson. Towards bridging the gap between Dalvik bytecode and native code during static analysis of Android applications. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2015 International*, pages 587–593, Aug 2015.

[126] Wei Le and Mary Lou Soffa. Marple: A demand-driven path-sensitive buffer overflow detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, SIGSOFT '08/FSE-16, pages 272–282, New York, NY, USA, 2008. ACM.

[127] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 36–49, 2010.

[128] Sungho Lee. JNI program analysis with automatically extracted C semantic summary. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 448–451, 2019.

[129] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[130] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[131] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of Android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[132] Lian Li, Cristina Cifuentes, and Nathan Keynes. Practical and effective symbolic analysis for buffer overflow detection. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 317–326, New York, NY, USA, 2010. ACM.

[133] Siliang Li and Gang Tan. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 442–452, New York, NY, USA, 2009. ACM.

[134] Zhenkai Liang and R Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222. ACM, 2005.

[135] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. *SIGSOFT Softw. Eng. Notes*, 28(5):317–326, September 2003.

[136] Mike Lyman. How to avoid the blind spot in static analysis tools caused by frameworks. Web page, 2016. https://www.synopsys.com/blogs/software-security/static-analysis-tool-framework-blind-spot/.

[137] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 145–152, Aug 2014.

[138] Daniel Marjamäki. Cppcheck. Web page. http://cppcheck.sourceforge.net/.

[139] Mathworks. Polyspace Bug Finder. https://www.mathworks.com/products/polyspace-bug-finder.html.

[140] Philip Mayer, Michael Kirsch, and Minh Anh Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development*, 5(1):1, 2017.

[141] John P McGregor, David K Karig, Zhijie Shi, and Ruby B Lee. A processor architecture defense against buffer overflow attacks. In *Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on*, pages 243–250. IEEE, 2003.

[142] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, 2015.

[143] Zaigham Mushtaq, Ghulam Rasool, and Balawal Shehzad. Multilingual source code analysis: A systematic literature review. *IEEE Access*, 2017.

[144] Alexios Mylonas, Anastasia Kastania, and Dimitris Gritzalis. Delegate the smartphone user? security awareness in smartphone platforms. *Computers & Security*, 34:47–66, 2013.

[145] Meiyappan Nagappan, Nuthan Munaiah, Craig Cabrey, Steven Kroh, and Nimish Parikh. Reporeapers. Web page, 2016. https://reporeapers.github.io/results/1.html.

[146] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 580–586, May 2005.

[147] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[148] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[149] NVD. National vulnerability database statistics. Web page, 2019. https://nvd.nist.gov/vuln/search/statistics.

[150] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with Munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1475–1482, 2018.

[151] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 780–785, Sept 2016.

[152] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, pages 93–104, 2006.

[153] Nicholas J. Percoco and Sean Schulte. Adventures in bouncerland: failures of automated malware detection with in mobile application markets. *Black Hat USA*, 2012.

[154] Rolf-Helge Pfeiffer and Andrzej Wąsowski. Taming the confusion of languages. In *European Conference on Modelling Foundations and Applications*, pages 312–328. Springer, 2011.

[155] Theodoros Polychniatis, Jurriaan Hage, Slinger Jansen, Eric Bouwers, and Joost Visser. Detecting cross-language dependencies generically. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 349–352, 2013.

[156] Davide Pozza, Riccardo Sisto, Luca Durante, and Adriano 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*, pages 1–7, 2006.

[157] C. Qian, X. Luo, Y. Le, and G. Gu. VulHunter: Toward discovering vulnerabilities in Android applications. *IEEE Micro*, 35(1):44–53, Jan 2015.

[158] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS*, volume 14, page 1125. Citeseer, 2014.

[159] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[160] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[161] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[162] H Gordon Rice. On completely recursively enumerable classes and their key arrays. *The Journal of Symbolic Logic*, 21(3):304–308, 1956.

[163] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[164] Risk Based Security. Vulndb quickview 2015 vulnerability trends. Web page, 2015. https://www.riskbasedsecurity.com/researchadv/.

[165] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[166] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[167] 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*, 43(6):492–530, June 2017.

[168] Robert C Seacord. *Secure Coding in C and C++*. Pearson Education, 2005.

[169] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[170] Hossain Shahriar and Mohammad Zulkernine. Classification of static analysis-based buffer overflow detectors. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 94–101, June 2010.

[171] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David A Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.

[172] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[173] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(4):410–457, 2006.

[174] Mukesh Soni. Defect prevention: Reducing costs and enhancing quality. *IBM: iSixSigma. com*, 19, 2006.

[175] Alexander Ivanov Sotirov. Automatic vulnerability detection using static source code analysis. Master's thesis, The University of Alabama, May 2005.

[176] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136. ACM, 2006.

[177] Statista. Number of available applications in the Google Play Store from December 2009 to June 2017, June 2017. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[178] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph: A program representation for optimal program transformations. In *European Symposium on Programming*, pages 389–405. Springer, 1990.

[179] Eric Stoltz, Michael P Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *HICSS (2)*, pages 43–53, 1994.

[180] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

[181] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[182] Mengtao Sun and Gang Tan. NativeGuard: Protecting Android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 165–176. ACM, 2014.

[183] Symantec. 2015 Internet security threat report. *Internet Security Threat Report*, 20:119, April 2015.

[184] Synopsys. Coverity Static Analysis. Web page. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.

[185] Gang Tan and Jason Croft. An empirical security study of the native code in the JDK. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[186] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[187] The European Union Agency for Network and Information Security (ENISA). Is software more vulnerable today? Web page, March 2018. https://www.enisa.europa.eu/publications/info-notes/is-software-more-vulnerable-today/.

[188] Harry Thornburg. 2019 global developer report: Devsecops, 2019. https://about.gitlab.com/2019/07/15/global-developer-report/.

[189] Sander Tichelaar. *Modeling object-oriented software for reverse engineering and refactoring.* PhD thesis, Verlag nicht ermittelbar, 2001.

[190] Lucas Torri, Guilherme Fachini, Leonardo Steinfeld, Vesmar Camara, Luigi Carro, Érika Cota, Po Box, and Porto Alegre. An evaluation of free/open source static analysis tools applied to embedded software. In *Test Workshop (LATW), 2010 11th Latin American*, pages 1–6, March 2010.

[191] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of Web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[192] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[193] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[194] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 880–881. ACM, 2007.

[195] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[196] Kostyantyn Vorobyov and Padmanabhan Krishna. Comparing model checking and static program analysis: A case study in error detection approaches. *Proc. SSV*, pages 1–7, 2010.

[197] David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, pages 2000–02, 2000.

[198] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishi, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 14–24. IEEE, 2017.

[199] Michael Weber, Vishal Shah, and Chengfang Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 1–11, 2001.

[200] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *Proceedings of the 2018*

ACM SIGSAC Conference on Computer and Communications Security, CCS '18, pages 1137–1150, New York, NY, USA, 2018. ACM.

[201] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341, 2014.

[202] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[203] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Ndss*, volume 3, pages 149–162, 2003.

[204] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, September 2003.

[205] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. NDroid: Toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828, 2018.

[206] Zhemin Yang and Min Yang. LeakMiner: Detect information leakage on Android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104, Nov 2012.

[207] Amir Reza Yazdanshenas and Leon Moonen. Tracking and visualizing information flow in component-based systems. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 143–152. IEEE, 2012.

[208] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36. ACM, 2011.

[209] Yves Younan. 25 years of vulnerabilities: 1988–2012. *Sourcefire Vulnerability Research Team*, 2013.

[210] Misha Zitser. *Securing software: An evaluation of static source code analyzers*. PhD thesis, Massachusetts Institute of Technology, 2003.

Appendices

Appendix A

Buffer error attribute examples in the source code

Table A.1: Write/read attribute adopted from [120]

Illegal Operation	Example
write	buffer[7] = 'X'
read	a = buffer[7]

Table A.2: Container and pointer attributes adopted from [120]

Container	Example
array	char buffer[5][7];
pointer	char *p or (*p)[10];

struct	typedef struct {
	char *p;
	} buf_struct;

union	union typedef union {
	char buffer[7];
	int val;
	} buf_union;

class	class buf_class {
	int *p;
	.....
	}

array of blocks (class, union, or struct)	buf_union array_buffer[7];
pointer of blocks (class, union, or struct)	buf_struct *p_buffer;

struct and class include pointers, while union include array

Table A.3: Data type attribute adopted from [120]

Data Type	Example
void	void * p;
boolean	bool buffer[7];
character	char buffer[7];
wide character	wchar_t buffer[7];
integer	int buffer[7];
floating point	float buffer[7];
double	double buffer[7];
pointer	char * buffer[7];

the bit width of each data type is matter in some situations, also it could be platform-dependent. Also, some data types could has data modifier, such as signed, unsigned, short, and long.

Table A.4: Magnitude attribute adopted from [120]

Magnitude	Example
1 byte	buffer[10] = ‘A‘;
8 bytes	buffer[17] = ‘A‘;
4096 bytes	buffer[4105] = ‘A‘;

Table A.5: Length/limit complexity attribute adopted from [120]

Container	Example
constant	strncpy(buffer, src, 10)
variable	strncpy(buffer, src, i)
linear expression	strncpy(buffer, src, 5*i + 2)
non-linear expression	strncpy(buffer, src, i%3)
function return value	strncpy(buffer, src, getSize())
array contents	strncpy(buffer, src, array[i])

Table A.6: Upper/lower bound attribute adopted from [120]

Bound	Example
upper	buffer[7]
lower	buffer[-1]

Buffer offset should be always: lower <= offset <= upper

Table A.7: Index complexity attribute adopted from [120]

Index Complexity	Example
constant	buffer[7]
variable	buffer[i]
linear expression	buffer[2*i + 3]
non-linear expression	buffer[i%5] or buffer[i*i]
function return value	buffer[strlen(str)]
array contents	buffer[array[i]]

Table A.8: Address complexity attribute adopted from [120]

Address Complexity	Example
constant	- buffer[x] - (buffer+2)[x] - (0x80097E34)[x] - *(p+2) - strcpy(buffer+2, src)
variable	- (buffer+i)[x] - (bufAddrVar)[x] - *(p+i) - strcpy(buffer+i, src)
linear expression	- (buffer+(5*i + 2))[x] - *(p+(5*i + 2)) - strcpy(buffer+(5*i + 2), src)
non-linear expression	- (buffer+(i%3))[x] - *(p+(i*i)) - strcpy(buffer+(i%3), src)
function return value	- (buffer+f())[x] - (getBufAddr())[x] - *(p+f()) - *(getBufPtr()) - strcpy(buffer+f(), src) - strcpy(getBufAddr(), src)
array contents	- (buffer+array[i])[x] - (array[i])[x] - *(p+ array[i]) - strcpy(buffer+ array[i], src)

Table A.9: Alias of buffer address attribute adopted from [120]

Container	Example

one alias	char buffer[7];
	char * alias_one;
	alias_one = buffer;
	alias_one[7] = ‘A‘;

two aliases	char buffer[7];
	char * alias_one;
	char * alias_two;
	alias_one = buffer;
	alias_two = alias_one;
	alias_two[7] = ‘A‘;

Table A.10: Alias of buffer index attribute adopted from [120]

Container	Example

one alias	int i, j;
	i = 7; j = i;
	buffer[j] = ‘A‘;

two aliases	int i, j, k;
	i = 7; j = i; k = j;
	buffer[k] = ‘A‘;

Table A.11: Continuous/discrete attribute adopted from [120]

Continuous/Discrete	Example
discrete	buffer[7] = ‘A‘;

continuous	for (i=0; i<8; i++) {
	buffer[i] = ‘A‘; }

Table A.12: Signed/unsigned mismatch attribute adopted from [120]

Mismatch	Example
no	memcpy(buffer, src, 8);

yes	signed int size = 6-sizeof(buffer);
	memcpy(buffer, src, size);

Table A.13: Loop complexity attribute adopted from [120]

Loop Complexity	Example
none	for (i=0; i<8; i++) { buffer[i] = ‘A‘; }
one	init = 0; for (i=init; i<8; i++) { buffer[i] = ‘A‘; }
two	init = 0; test = 8; for (i=init; i<test; i++) { buffer[i] = ‘A’; }
three	init = 0; test = 8; inc = k − 7; for (i=init; *i* <test; i += inc) { buffer[i] = ‘A‘; }

Table A.14: Taint attribute adopted from [120]

Taint type	Example
argc/argv	using values from argv
environment variables	getenv
file read	(or stdin) fgets, fread, read
socket/service	recv
process environment	getcwd

Table A.15: Memory location attribute adopted from [120]

Data Location	Example
Stack	void fun (){ int buffer[7] ;}
Heap	p=(int*)calloc(n, sizeof(int));
Data	static char buffer[7] = "0123456";
BSS	static char buffer[7];
Shared memory	any buffer located inside shared memory

Appendix B

Seven pernicious kingdom taxonomy of security warnings

B.1 Input Validation and Representation (IVR)

- Buffer Overflow
- Command Injection
- Cross-Site Scripting
- Format String
- HTTP Response Splitting
- Illegal Pointer Value
- Integer Overflow
- Log Forging
- Path Manipulation
- Process Control
- Resource Injection
- Setting Manipulation
- SQL Injection
- String Termination Error Handling

- Unsafe JNI
- XML Validation

B.2 Improper Fulfillment of API Contract, or API Abuse (API)

- Dangerous Function
- Directory Restriction
- Heap Inspection
- Often Misused: Authentication
- Often Misused: Exception Handling
- Often Misused: File System
- Often Misused: Privilege Management
- Often Misused: Strings
- Unchecked Return Value

B.3 Security Features (SF)

- Insecure Randomness
- Least Privilege Violation
- Missing Access Control
- Password Management
- Password Management: Empty Password in Config File
- Password Management: Hard-Coded Password
- Password Management: Password in Config File
- Password Management: Weak Cryptography
- Privacy Violation

B.4 Time and State (TS)

- Deadlock
- Failure to Begin a New Session upon Authentication
- File Access Race Condition: TOCTOU
- Insecure Temporary File
- Signal Handling Race Conditions

B.5 Error Handling (ERR)

- Empty Catch Block, Language-Independent
- Overly-Broad Catch Block
- Overly-Broad Throws Declaration

B.6 Indicator of Poor Code Quality (CQ)

- Double Free
- Inconsistent Implementations
- Memory Leak
- Null Dereference
- Obsolete
- Undefined Behavior
- Uninitialized Variable
- Unreleased Resource
- Use After Free

B.7 Insufficient Encapsulation (ENC)

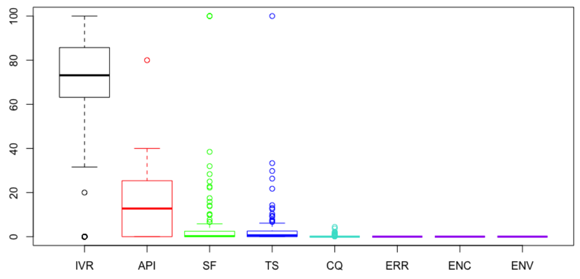
- Data Leaking Between Users
- Leftover Debug Code
- Mobile Code: Non-Final Public Field
- Private Array-Typed Field Returned From a Public Method
- Public Data Assigned to Private Array-Typed Field
- System Information Leak
- Trust Boundary Violation

B.8 Environment (ENV)

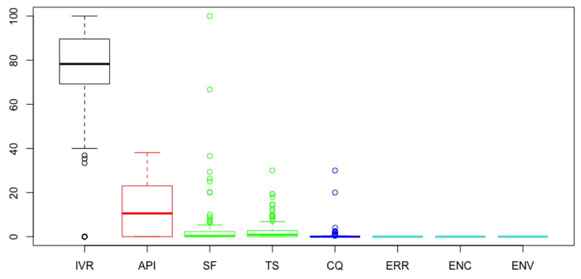
- Insecure Compiler Optimization
- Insecure Configuration Management

Appendix C

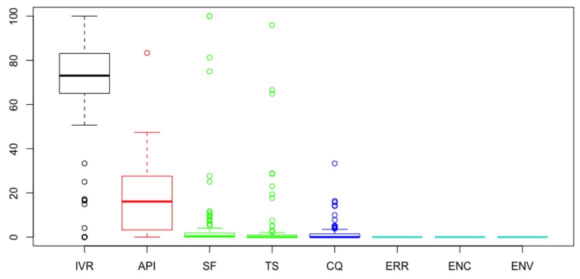
Bug warning distribution in 2012 and 2017 among SAST tools



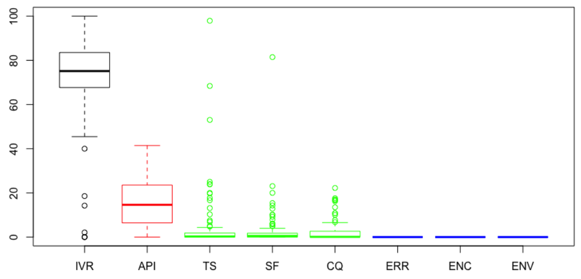
(a) Bug warning categories in RATS at 2012



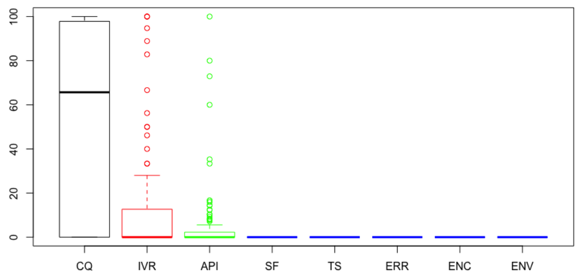
(b) Bug warning categories in RATS at 2017



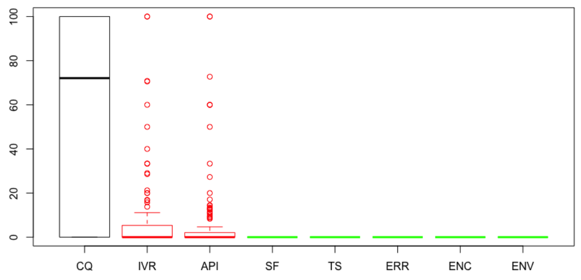
(c) Bug warning categories in Flawfinder at 2012



(d) Bug warning categories in Flawfinder at 2017

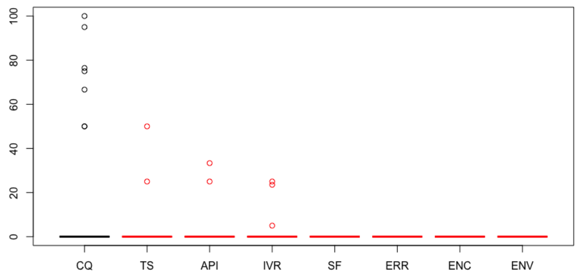


(e) Bug warning categories in Cppcheck at 2012

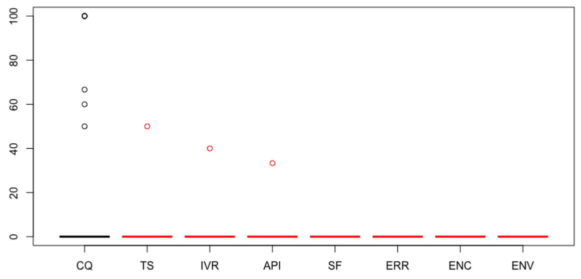


(f) Bug warning categories in Cppcheck at 2017

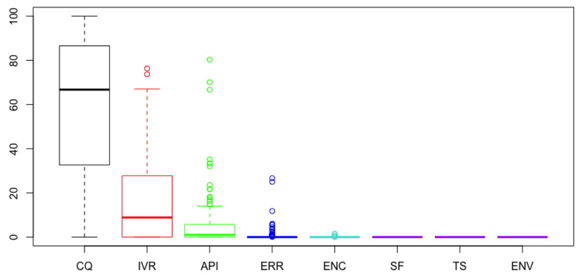
Figure C.1: Box-and-whisker plot and Scott-Knott ESD of the number of bugs detected by RATS, Flawfinder, and Cppcheck. In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different classes of SPK bugs. Each data point represents a project.



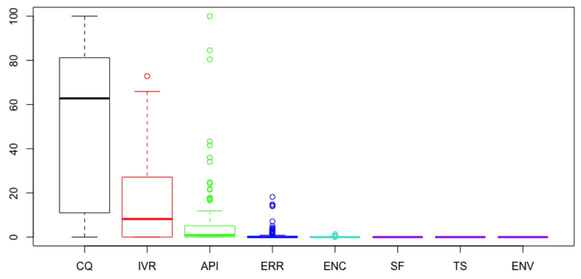
(a) Bug warning categories in Clang Static Analyzer at 2012



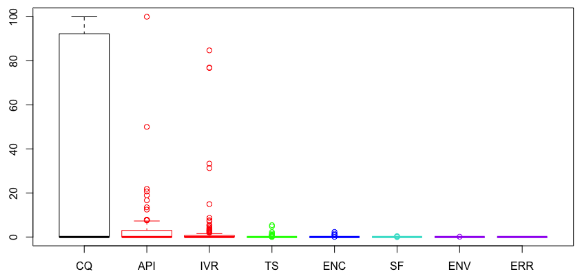
(b) Bug warning categories in Clang Static Analyzer at 2017



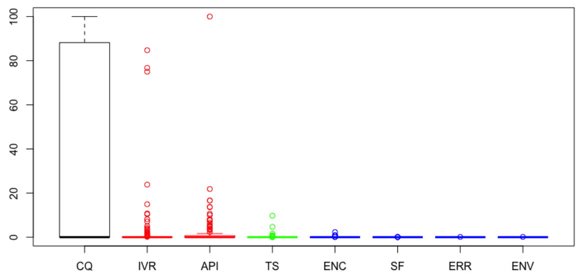
(c) Bug warning categories in PVS-Studio at 2012



(d) Bug warning categories in PVS-Studio at 2017



(e) Bug warning categories in Parasoft C/C++test at 2012

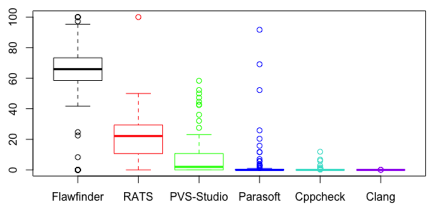


(f) Bug warning categories in Parasoft C/C++test at 2017

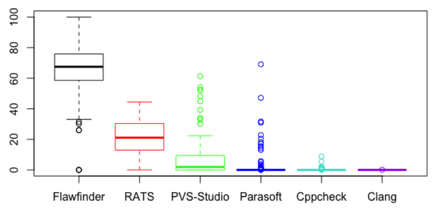
Figure C.2: Box-and-whisker plot and Scott-Knott ESD of the number of bugs detected by Clang Static Analyzer, PVS-Studio, and Parasoft C/C++test. In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different classes of SPK bugs. Each data point represents a project.

Appendix D

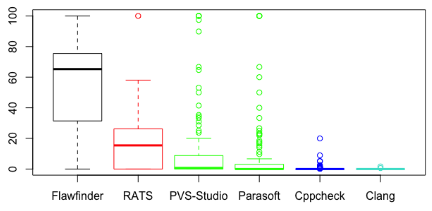
Bug warning distribution in 2012 and 2017 among SPK categories



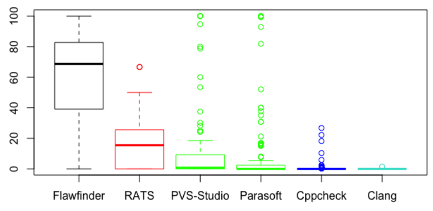
(a) IVR warnings among SAST tools in 2012



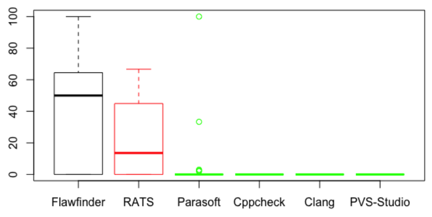
(b) IVR warnings among SAST tools in 2017



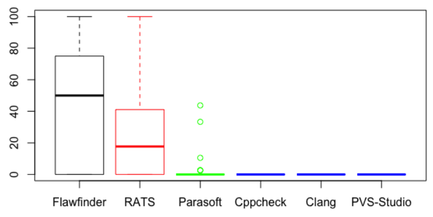
(c) API warnings among SAST tools in 2012



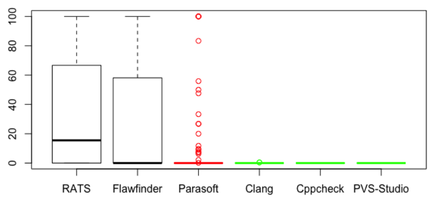
(d) API warnings among SAST tools in 2017



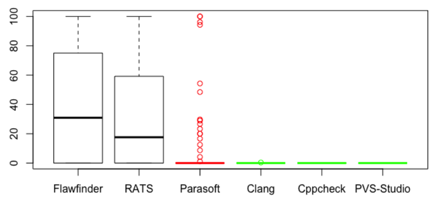
(e) SF warnings among SAST tools in 2012



(f) SF warnings among SAST tools in 2017

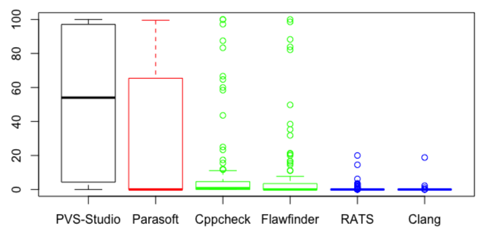


(g) TS warnings among SAST tools in 2012

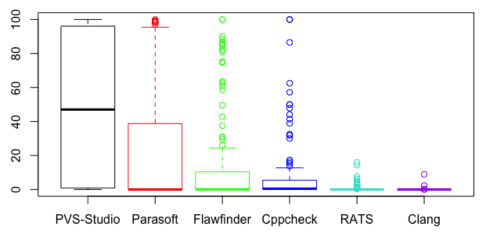


(h) TS warnings among SAST tools in 2017

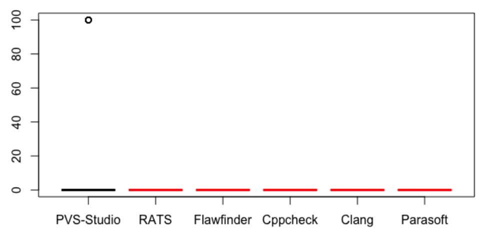
Figure D.1: Box-and-whisker plot and Scott-Knott ESD of the number of bugs classified as Input Validation and Representation (IVR), API Abuse (API), Security Feature (SF), and Time and State (TS). In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different SAST tools. Each data point represents a project.



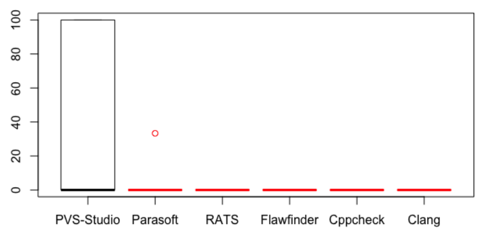
(a) CQ warnings among SAST tools in 2012



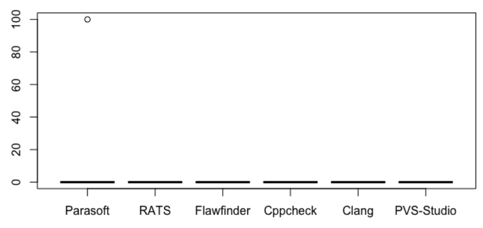
(b) CQ warnings among SAST tools in 2017



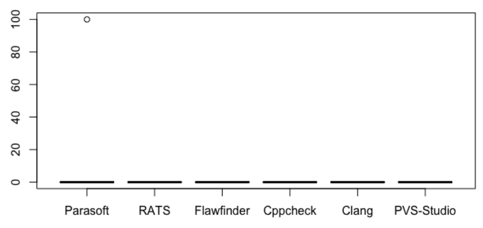
(c) ERR warnings among SAST tools in 2012



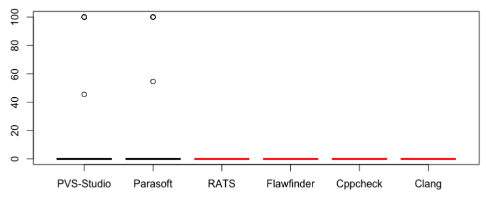
(d) ERR warnings among SAST tools in 2017



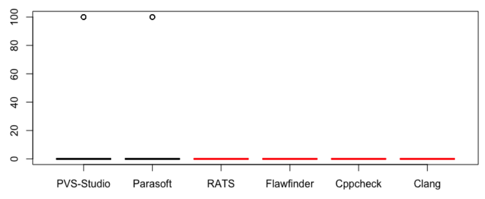
(e) ENC warnings among SAST tools in 2012



(f) ENC warnings among SAST tools in 2017



(g) ENV warnings among SAST tools in 2012

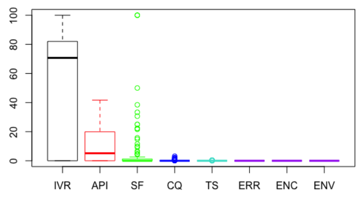


(h) ENV warnings among SAST tools in 2017

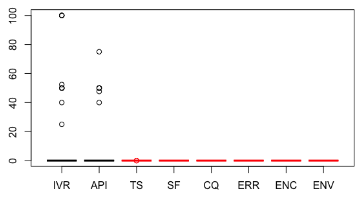
Figure D.2: Box-and-whisker plot and Scott-Knott ESD of the number of bugs classified as Code Quality (CQ), Errors (ERR), Encapsulation (ENC), and Environment (ENV). In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different SAST tools. Each data point represents a project.

Appendix E

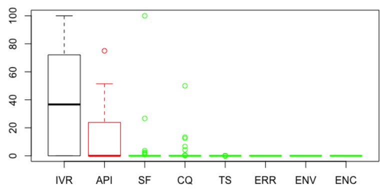
Bug warning rates among SAST tools



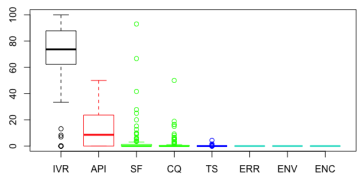
(a) *Remained Warnings* in RATS



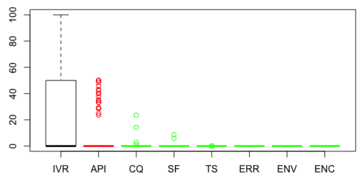
(b) *Modified Warnings* in RATS



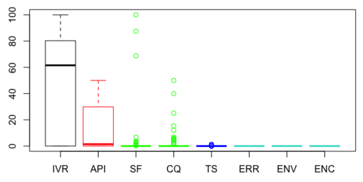
(c) *Disappeared Warnings* in RATS



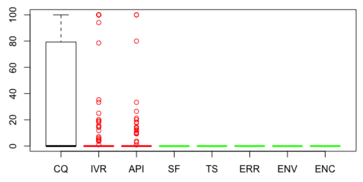
(d) *Remained Warnings*in Flawfinder



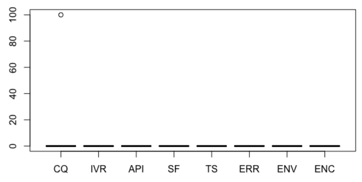
(e) *Modified Warnings* in Flawfinder



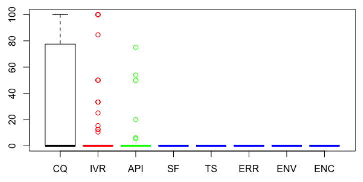
(f) *Disappeared Warnings* in Flawfinder



(g) *Remained Warnings* in Cppcheck

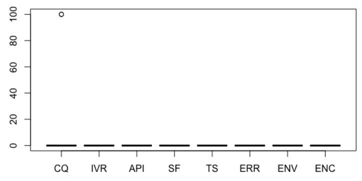


(h) *Modified Warnings* in Cppcheck

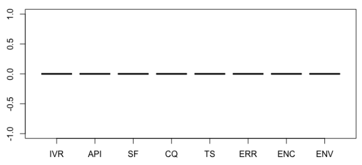


(i) *Disappeared Warnings* in Cppcheck

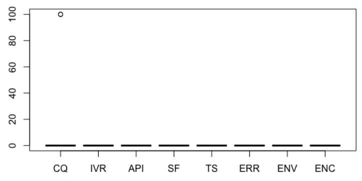
Figure E.1: Box-and-whisker plot and Scott-Knott ESD of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories by RATS, Flawfinder, Cppcheck. In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different classes of SPK bugs. Each data point represents a project.



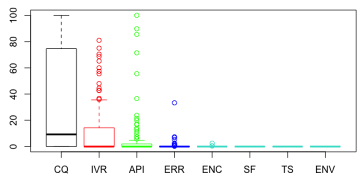
(a) *Remained Warnings* in Clang Static Analyzer



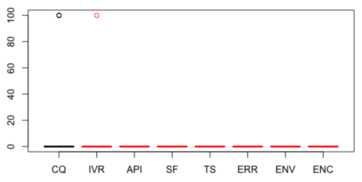
(b) *Modified Warnings* in Clang Static Analyzer



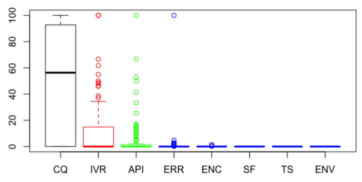
(c) *Disappeared Warnings* in Clang Static Analyzer



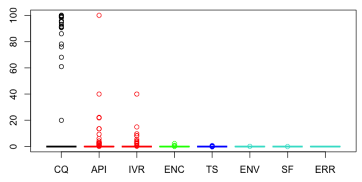
(d) *Remained Warnings* in PVS-Studio



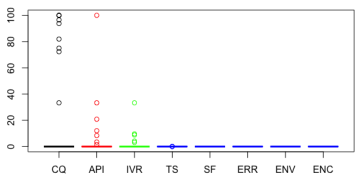
(e) *Modified Warnings* in PVS-Studio



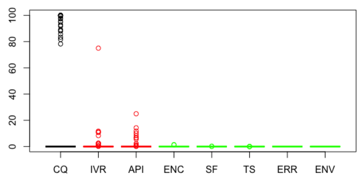
(f) *Disappeared Warnings* in PVS-Studio



(g) *Remained Warnings* in Parasoft C/C++test



(h) *Modified Warnings* in Parasoft C/C++test

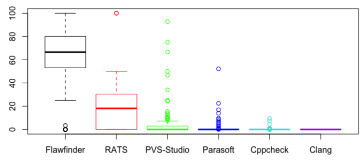


(i) *Disappeared Warnings* in Parasoft C/C++test

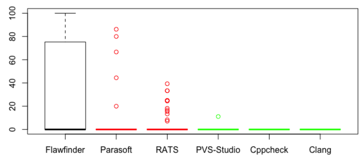
Figure E.2: Box-and-whisker plot and Scott-Knott ESD of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories by Clang Static Analyzer, PVS-Studio, and Parasoft C/C++test. In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different classes of SPK bugs. Each data point represents a project.

Appendix F

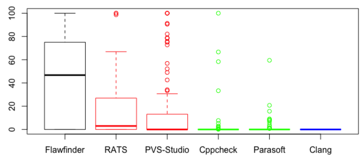
Bug warning rates among SPK categories



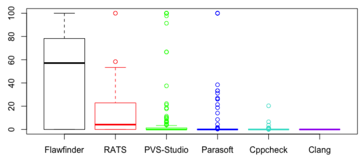
(a) *Remained Warnings* among IVR



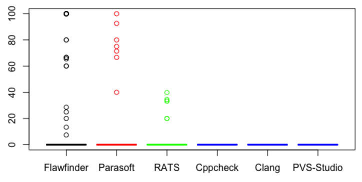
(b) *Modified Warnings* among IVR



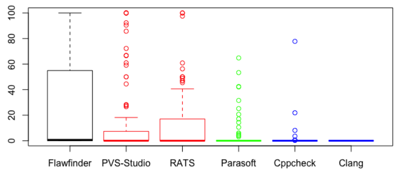
(c) *Disappeared Warnings* among IVR



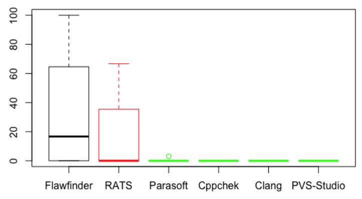
(d) *Remained Warnings* among API



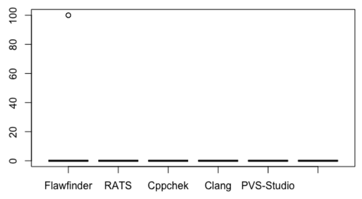
(e) *Modified Warnings* among API



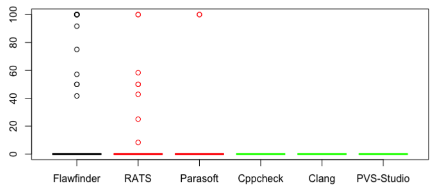
(f) *Disappeared Warnings* among API



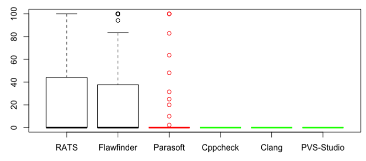
(g) *Remained Warnings* among SF



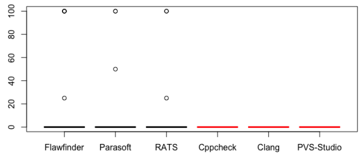
(h) *Modified Warnings* among SF



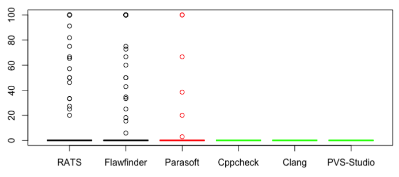
(i) *Disappeared Warnings* among SF



(j) *Remained Warnings* among TS

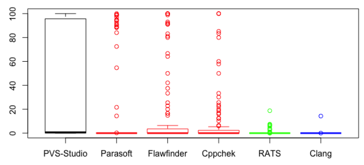


(k) *Modified Warnings* among TS

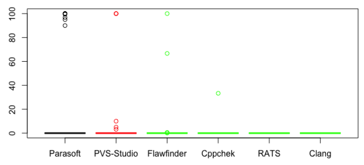


(l) *Disappeared Warnings* among TS

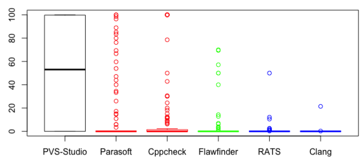
Figure F.1: Box-and-whisker plot and Scott-Knott ESD of the *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories among Input Validation and Representation (IVR), API Abuse (API), Security Features (SF), and Time and State (TS). In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different SAST tools. Each data point represents a project.



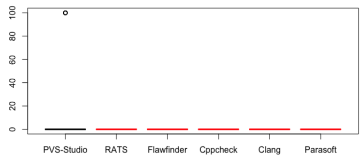
(a) *Remained Warnings* among CQ



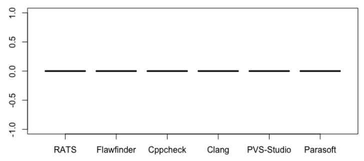
(b) *Modified Warnings* among CQ



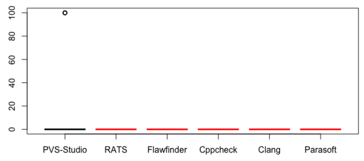
(c) *Disappeared Warnings* among CQ



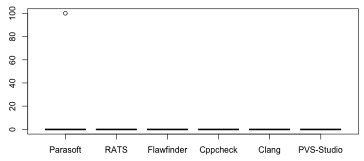
(d) *Remained Warnings* among ERR



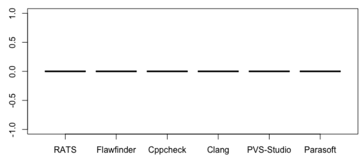
(e) *Modified Warnings* among ERR



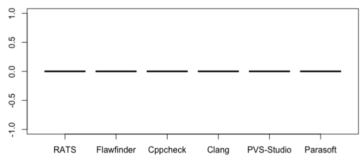
(f) *Disappeared Warnings* among ERR



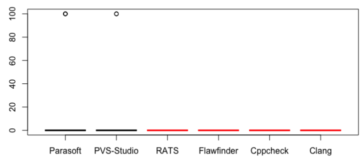
(g) *Remained Warnings* among ENC



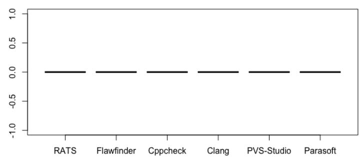
(h) *Modified Warnings* among ENC



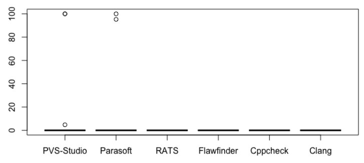
(i) *Disappeared Warnings* among ENC



(j) *Remained Warnings* among ENV



(k) *Modified Warnings* among ENV



(l) *Disappeared Warnings* among ENV

Figure F.2: Box-and-whisker plot and Scott-Knott ESD of tthe *Remained Warnings* (FP), *Modified Warnings* and *Disappeared Warnings* (TP) among different bug categories among Code Quality (CQ), Errors (ERR), Encapsulation (ENC), and Environment (ENV). In this figure, the y-axis represents normalized number of bug(bugs/LOC), and the x-axis represents different SAST tools. Each data point represents a project.

Appendix G

Cross-link (xlink) functions in JNI

CallStaticObjectMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticObjectMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticObjectMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticBooleanMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticBooleanMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticBooleanMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticByteMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticByteMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticByteMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticCharMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticCharMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticCharMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticShortMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticShortMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticShortMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticIntMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticIntMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticIntMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticLongMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticLongMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticLongMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticFloatMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticFloatMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticFloatMethodA (JNIEnv*, jclass, jmethodID, jvalue*)

CallStaticDoubleMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticDoubleMethodV(JNIEnv*, jclass, jmethodID, va_list)
CallStaticDoubleMethodA(JNIEnv*, jclass, jmethodID, jvalue*)
CallStaticVoidMethod (JNIEnv*, jclass, jmethodID, ...)
CallStaticVoidMethodV (JNIEnv*, jclass, jmethodID, va_list)
CallStaticVoidMethodA (JNIEnv*, jclass, jmethodID, jvalue*)
CallObjectMethod (JNIEnv*, jobject, jmethodID, ...)
CallObjectMethodV(JNIEnv*, jobject, jmethodID, va_list)
CallObjectMethodA(JNIEnv*, jobject, jmethodID, jvalue*)
CallBooleanMethod(JNIEnv*, jobject, jmethodID, ...)
CallBooleanMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallBooleanMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallByteMethod (JNIEnv*, jobject, jmethodID, ...)
CallByteMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallByteMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallCharMethod (JNIEnv*, jobject, jmethodID, ...)
CallCharMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallCharMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallShortMethod (JNIEnv*, jobject, jmethodID, ...)
CallShortMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallShortMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallIntMethod (JNIEnv*, jobject, jmethodID, ...)
CallIntMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallIntMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallLongMethod (JNIEnv*, jobject, jmethodID, ...)
CallLongMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallLongMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallFloatMethod (JNIEnv*, jobject, jmethodID, ...)
CallFloatMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallFloatMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallDoubleMethod (JNIEnv*, jobject, jmethodID, ...)
CallDoubleMethodV(JNIEnv*, jobject, jmethodID, va_list)
CallDoubleMethodA(JNIEnv*, jobject, jmethodID, jvalue*)
CallVoidMethod (JNIEnv*, jobject, jmethodID, ...)
CallVoidMethodV (JNIEnv*, jobject, jmethodID, va_list)
CallVoidMethodA (JNIEnv*, jobject, jmethodID, jvalue*)
CallNonvirtualObjectMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualObjectMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)

CallNonvirtualObjectMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualBooleanMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualBooleanMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualBooleanMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualByteMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualByteMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualByteMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualCharMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualCharMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualCharMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualShortMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualShortMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualShortMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualIntMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualIntMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualIntMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualLongMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualLongMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualLongMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualFloatMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualFloatMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualFloatMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualDoubleMethod (JNIEnv*, jobject, jclass, jmethodID, ...)
CallNonvirtualDoubleMethodV (JNIEnv*, jobject, jclass, jmethodID, va_list)
CallNonvirtualDoubleMethodA (JNIEnv*, jobject, jclass, jmethodID, jvalue*)
CallNonvirtualVoidMethod (JNIEnv*, jobject, jclass, jmethodID, ...)

Appendix H

Safety constraints

H.1 Constraint

Table H.1: Safety constraints implemented in BEFinder

Constraint #	Constraint description
1	SourceLen_Lt_SinkAllocSize
2	Len_Le_SinkAllocSize
3	Len_Le_SourceAllocSize
4	Len_Ge_Zero
5	SourceString_Is_NullTerminated
6	Source_Is_NullChecked
7	Sink_Is_NullChecked
8	FormatArg_Is_Constant
9	FormatSpecifiersNumber_Match_VarArg
10	FormatBufferSize_Is_Controlled
11	Index_Is_WithinBound
12	SourceLen_plus_SinkLen_Gt_SinkSize
13	Len_plus_SinkLen_Gt_SinkSize
14	Len_Le_BufferLen

H.2 String functions

Table H.2: String functions: String concatenation

Function	Constraints #
char *strcat(char *dest, const char *src)	6,7,12
wchar_t *wcscat(wchar_t *dest, const wchar_t *src)	6,7,12
char strlcat(char *dst, const char *src, size_t size)	6,7,13
errno_t strcat_s(char *restrict dest, rsize_t destsz, const char *restrict src)	6,7,13
errno_t strncat_s(char *restrict dest, rsize_t destsz, const char *restrict src, rsize_t count)	6,7,13
char *strncat(char *dest, const char *src, size_t n)	6,7,13
wchar_t *wcsncat(wchar_t *dest, const wchar_t *src, size_t n)	6,7,13
errno_t wcsncat_s(wchar_t *restrict dest, rsize_t destsz, const wchar_t *restrict src, rsize_t count)	6,7,13
errno_t wcscat_s(wchar_t *restrict dest, rsize_t destsz, const wchar_t *restrict src)	6,7,13

Table H.3: String functions: String copy and transformation

Function	Constraints #
char * strcpy (char * destination, const char * source)	1,5,6,7
wchar_t *wcpcpy(wchar_t *dest, const wchar_t *src)	1,5,6,7
char *stpcpy(char *dest, const char *src)	1,5,6,7
wchar_t *wcscpy(wchar_t *dest, const wchar_t *src)	1,5,6,7
char *strncpy(char *dest, const char *src, size_t n)	2,4,6,7
errno_t strncpy_s(char *restrict dest, rsize_t destsz, const char *restrict src, rsize_t count)	2,4,6,7
size_t strlcpy(char *dst, const char *src, size_t size)	2,4,6,7
char *stpncpy(char *dest, const char *src, size_t n)	2,4,6,7
errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict src)	2,4,6,7
wchar_t *wcsncpy(wchar_t *dest, const wchar_t *src, size_t n)	2,4,6,7
errno_t strncpy_s(char *restrict dest, rsize_t destsz, const char *restrict src, rsize_t count)	2,4,6,7
errno_t wcsncpy_s(wchar_t *restrict dest, rsize_t destsz, const wchar_t *restrict src, rsize_t n)	2,4,6,7
wchar_t *wcpncpy(wchar_t *dest, const wchar_t *src, size_t n)	2,4,6,7
errno_t wcscpy_s(wchar_t *restrict dest, rsize_t destsz, const wchar_t *restrict src)	2,4,6,7
size_t strxfrm(char *dest, const char *src, size_t n)	2,6,7
size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2, size_t n)	2,6,7
size_type copy (charT* s, size_type len, size_type pos = 0) const	2,6,7

H.3 Format string functions

Table H.4: Formatted string input functions

Function	Constraints #
int fwscanf(FILE * stream, const wchar_t *restrict format, ...)	8,9,10
int wscanf(const wchar_t *restrict format, ...)	8,9,10
int vscanf (const char * format, va_list arg)	8,9,10
int vsscanf (const char * s, const char * format, va_list arg)	8,9,10
int fscanf (FILE * stream, const char * format, ...)	8,9,10
int vfscanf(FILE *stream, const char *format, va_list ap)	8,9,10
int sscanf(const char *str, const char *format, ...)	8,9,10
int swscanf(const wchar_t *restrict ws, const wchar_t *restrict format, ...)	8,9,10
int vwscanf (const wchar_t * format, va_list arg)	8,9,10
int vswscanf (const wchar_t* ws, const wchar_t* format, va_list arg)	8,9,10
int vfwscanf (FILE* stream, const wchar_t* format, va_list arg)	8,9,10

Table H.5: Formatted string output functions

Function	Constraints #
int printf(const char *format, ...)	8,9
int sprintf(char *str, const char *format, ...)	8,9,10
int wprintf(const wchar_t *format, ...)	8,9
int vwprintf(const wchar_t *format, va_list args)	8,9
int vfprintf(FILE *stream, const char *format, va_list ap)	8,9
int fwprintf(FILE *stream, const wchar_t *format, ...)	8,9
int vfwprintf(FILE *stream, const wchar_t *format, va_list args)	8,9
int fprintf (FILE * stream, const char * format, ...)	8,9
int vprintf (const char * format, va_list arg)	8,9
int vsprintf(char *str, const char *format, va_list ap)	8,9,10
int asprintf(char **strp, const char *fmt, ...)	8,9
int vasprintf(char **strp, const char *fmt, va_list ap)	8,9
void syslog(int priority, const char *format, ...)	8,9
void vsyslog(int priority, const char *format, va_list ap)	8,9
int snprintf(char *str, size_t size, const char *format, ...)	8,9,10
int vsnprintf(char *str, size_t size, const char *format, va_list ap)	8,9,10
int swprintf(wchar_t *wcs, size_t maxlen, const wchar_t *format, ...)	8,9,10
int vswprintf(wchar_t *wcs, size_t maxlen, const wchar_t *format, va_list args)	8,9,10
vasnprintf (char *resultbuf, size_t * lengthp, const char *format, va_list args)	8,9,10

H.4 Memory functions

Table H.6: Memory operation functions

Function	Constraints #
void *memcpy(void *dest, const void *src, size_t n)	2,3,4,6,7,14
void *mempcpy(void *dest, const void *src, size_t n)	2,3,4,6,7,14
errno_t memcpy_s(void *restrict dest, rsize_t destsz, const void *restrict , rsize_t)	2,3,4,6,7,14
wchar_t *wmempcpy(wchar_t *dest, const wchar_t *src, size_t n)	2,3,4,6,7,14
wchar_t * wmemcpy (wchar_t *restrict wto, const wchar_t *restrict wfrom, size_t size)	2,3,4,6,7,14
errno_t wmemcpy_s(wchar_t *restrict dest, rsize_t destsz, const wchar_t *restrict , rsize_t)	2,3,4,6,7,14
void * mempcpy (void *restrict to, const void *restrict from, size_t size)	2,3,4,6,7,14
void *memccpy(void *dest, const void *src, int c, size_t n)	2,3,4,6,7,14
wchar_t* wmemmove (wchar_t* destination, const wchar_t* source, size_t num)	2,3,4,6,7,14
errno_t wmemmove_s(wchar_t *dest, rsize_t destsz, const wchar_t *src, rsize_t count)	2,3,4,6,7,14
void *memmove(void *dest, const void *src, size_t n)	2,3,4,6,7,14
errno_t memmove_s(void *dest, rsize_t destsz, const void *src, rsize_t count)	2,3,4,6,7,14
void bcopy(const void *src, void *dest, size_t n)	2,3,4,6,7,14

Table H.7: Memory write functions

Function	Constraints #
ssize_t write(int fildes, const void *buf, size_t nbyte)	3
ssize_t writev(int fd, const struct iovec *iov, int iovcnt)	3
size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream)	3
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt,	3
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset)	3
ssize_t pwrite64(int fildes, const void *buf, size_t nbyte, off64_t offset)	3