CLAPP: Characterizing Loops in Android Applications (Invited Talk)

Yanick Fratantonio
UC Santa Barbara, USA
yanick@cs.ucsb.edu

Aravind Machiry
UC Santa Barbara, USA
machiry@cs.ucsb.edu

Antonio Bianchi
UC Santa Barbara, USA
antoniob@cs.ucsb.edu

Christopher Kruegel
UC Santa Barbara, USA
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara, USA
vigna@cs.ucsb.edu

ABSTRACT

When performing program analysis, *loops* are one of the most important aspects that needs to be taken into account. In the past, many approaches have been proposed to analyze loops to perform different tasks, ranging from compiler optimizations to Worst-Case Execution Time (WCET) analysis. While these approaches are powerful, they focus on tackling very specific categories of loops and known loop patterns, such as the ones for which the number of iterations can be statically determined.

In this work, we developed a static analysis framework to characterize and analyze *generic* loops, without relying on techniques based on pattern matching. For this work, we focus on the Android platform, and we implemented a prototype, called CLAPP, that we used to perform the first large-scale empirical study of the usage of loops in Android applications. In particular, we used our tool to analyze a total of 4,110,510 loops found in 11,823 Android applications, and we gained several insights related to the performance issues and security aspects associated with loops.

Categories and Subject Descriptors

D.4.6 [**Software Engineering**]: Security and Protection

Keywords

Android, Static Analysis, Loop Analysis

1. INTRODUCTION

Over the past few decades, there has been an explosion in the development and application of program analysis techniques to achieve a variety of goals. Program analysis has been used for compilation and optimization purposes, for studying a variety of program properties, for detecting bugs, vulnerabilities, malicious functionality, and, ultimately, for understanding program behavior. When performing program analysis, one of the most important aspects that needs to be taken into account are *loops*, which are undoubtedly one of the most useful and essential constructs when writing programs. However, they are also one of the most challenging ones to handle: In fact, even answering the simplest questions (e.g., "Is a given loop going to terminate?") is, in the general case, an undecidable problem.

When applying program analysis, loops also have particular importance for optimization or security purposes: the execution of a performance-intensive operation (e.g., a GUI-related operation) or of a security-relevant operation (e.g., file deletion) might not constitute a problem when executed only occasionally, but it could be deemed as problematic when executed multiple times within a loop. In the past, much research has been focused on the analysis of loops, mainly to perform Worst-Case Execution Time (WCET) analysis [2], which aims to statically determine how many times a loop can be executed in the *worst possible case*, and to perform *loop unrolling* [1], which aims to *unwind* loops' execution to gain a performance boost. While these approaches are powerful, they rely on pattern matching or focus on handling only very specific types of loops.

In this work[1], we developed a novel loop analysis framework (based on static analysis) to characterize loops under many different aspects, such as how they are controlled, which operations they perform, and their impact under both the performance and security aspects. In particular, we focused on the analysis of Android applications, and we developed a tool, called CLAPP, which works directly on Dalvik bytecode, and it therefore does not rely on having access to the application's source code. The key advantage of our approach is that it is completely generic and can be applied to any kind of program. Moreover, our approach does not rely on the identification of *known* cases through techniques based on pattern matching.

We used CLAPP to perform the first large-scale empirical study on 4,110,510 loops contained in 11,823 distinct Android applications. The results allowed us to study the different use cases for writing loops in Android applications, and, more in general, to characterize the usage of loops under two main perspectives, performance and security.

2. LOOP ANALYSIS FRAMEWORK

Our loop analysis framework is constituted by several analysis steps. First, the analyzer unpacks the given Android

[1]The full version of this paper has been published in FSE 2015 [3].

DeMobile'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3815-8/15/08
http://dx.doi.org/10.1145/2804345.2804355

application, and it parses the Dalvik bytecode into a custom intermediate representation (in SSA form) suitable for performing static analysis. As a second step, the analyzer identifies all loops defined in the application, and it then performs the analyses at the core of our approach, *control analysis* and *body analysis*.

Control Analysis. This analysis aims to determine whether the number of loop's iterations is bounded, and, more in general, to characterize the factors that control it. This is achieved by first identifying all exit paths and exit conditions. Then, for each register involved in each condition, the analyzer constructs an *expression tree* that encodes the operations that *initialize* the register's value before the first iteration of the loop, and that *update* it during each iteration. Then, the analysis performs selective abstract interpretation to determine what is the *trend* of each register's value, attempting to answer questions such as "Is the value constant?" and "Is the value going to eventually increase?". As we discuss in the full version of this paper [3], the answers to these questions proved to be of key importance to answer termination-related questions, and to characterize which kind of external factors can influence the number of iterations of a given loop.

Body Analysis. This analysis aims to characterize the operations executed for each loop's iterations. For each loop, the analysis computes the set of framework APIs that could be potentially invoked within the loop's body. This is done by first computing an over-approximation of the callgraph, and by then performing reachability analysis. This set of methods characterizes the intent of a given loop, and it allows us to perform subsequent powerful analyses, such as the identification of problematic loops.

3. EVALUATION

This section discusses the large-scale empirical study we performed, the results we obtained, and the insights we gained.

Dataset. Our dataset is constituted by 15,240 applications selected, at random, from the ones collected by the PlayDrone project [4].

Overall Results. Among the 15,240 applications selected for the experiments, our prototype was able to successfully analyze 11,823 (77.57%) of them. The analysis of the remaining 3,417 applications did not terminate before the timeout (30 minutes per app). For the applications that were successfully processed, our tool analyzed a cumulative total of 4,110,510 loops, and identified a total of 118,190,014 API framework methods that could potentially be invoked in these loops. On average, analyzing each application takes 96.77 seconds, and analyzing each loop takes 50.86 seconds.

Control Analysis Results. We now report the results related to the control aspect of loops. Our analysis identified 3,196,119 (77.70%) *simple* loops (i.e., loops with only one exit path with one condition) and 910,841 (22.22%) *complex* loops (i.e., loops with one or more exit paths with several exit conditions). For the 3,550 (0.08%) remaining loops, our analysis determined that there were no explicit exit paths, which might indicate the presence of infinite loops. As another interesting statistic, we found that 266,667 (6.48%) of the loops contain at least one nested loop. Our analysis also determined that 2,601,240 (63.28%) of the loops are guaranteed to terminate, and that all the exit paths associated to 6,256 loops do not seem to be satisfiable, thus once again indicating the presence of potentially-infinite loops. Our analysis also classified 24,842 (0.60%) loops as *risky*, with which we refer to loops that, independently from whether they terminate or not, a subtle change in their body might cause them to become infinite. As a clarifying example, consider the loop "`for (i=0; i != 12; i+=3){...}`": this loop will iterate exactly four times. However, a modification to how the variable `i` is updated could suddenly introduce an infinite loop.

Body Analysis Results. The results of the body analysis show that, in most cases, developers make use of loops to invoke low-risk APIs. For example, they use loops to perform simple iterations over app-specific objects, perform cryptographic operations, generating random numbers, parsing data, and iterating over different data structures. However, our analysis also identified 1,057,628 loops that could potentially invoke network-related API functions, 764,240 of which could be executed by the app's main UI thread. The Android official documentation clearly states that no potentially-blocking operations should be ever performed within the main UI thread, since, in certain scenarios, the app might be terminated with the infamous Application Not Responding (ANR) error message. This aspect is so problematic that a recent version of Android introduced `StrictMode`, which is, quoting the official documentation, a tool to "catch accidental disk or network access on the application's main thread." However, this mechanism can be explicitly disabled by an Android application, and, interestingly, we found 207,888 loops that potentially do so. We invite the interested reader to consult the detailed results reported in the full version of this paper [3].

4. CONCLUSIONS

We presented CLAPP, a tool that implements a general loop analysis framework (based on static analysis) to study a variety of aspects related to the usage of loops in Android applications. We used CLAPP to perform the first large-scale empirical study on 4,110,510 loops contained in 11,823 Android apps, and we gained several insights related to their control, body, performance, and security aspects.

5. REFERENCES

[1] D. Berlin, D. Edelsohn, and S. Pop. High-level Loop Optimizations for GCC. In *Proceedings of the GCC Developers Summit*, 2004.

[2] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *Workshop on Worst-Case Execution Time Analysis (WCET)*, 2007.

[3] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. CLAPP: Characterizing Loops in Android Applications. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2015.

[4] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRIC)*, 2014.