CLAPP: Characterizing Loops in Android Applications

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. As part of our evaluation, we provide the detailed results of our empirical study, we show how our analysis was able to determine that the execution of 63.28% of the loops is bounded, and we discuss several interesting 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.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786873

When performing program analysis, one of most important aspects that needs to be taken into account are *loops*. Loops 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, undecidable.

Loops also have particular importance when applying program analysis for optimization or security purposes. One important observation is that, in certain scenarios, operations in a loop might have a greater impact than operations which are not part of any loop. In fact, consider, as an example, an operation that is performance-intensive (e.g., a GUI-related operation): while this functionality might not constitute a problem when executed only occasionally, it could be deemed as problematic when executed multiple times within a loop. Similarly, in the case of security, consider a file deletion operation: this operation is not dangerous *per se*; however, it could be deemed as such when the same functionality is executed within a loop, as the application might have the potential to wipe all of the user's data. Another relevant scenario is a malicious application that implements and executes an infinite loop with the aim of draining the device's battery. While this malicious functionality might not constitute a serious issue for the average user, it might have disastrous effects when the application is used in security-critical scenarios (e.g., by a soldier in the battlefield).

In the past, much research has been focused on the analysis of loops. For example, research done in the context of compiler-level optimizations has focused on the analysis of loops to understand, for example, whether it is possible to *unwind* their execution – a process known as *loop unrolling* [29, 30, 7, 23]. In this scenario, the goal is to optimize a program's execution speed at the expense of its binary size. Another thrust of research has focused on Worst-Case Execution Time (WCET) analysis [22, 13, 21, 9], which aims to statically determine how many times a loop can be executed in the *worst possible case*. This analysis is particularly relevant for the design of real-time systems, where it is critically important to determine a conservative time estimate of when a given function will terminate its execution, so that it is possible to know when its output can be considered as *valid*, and thus ready to be consumed.

While these approaches are powerful, they focus on handling only very specific types of loops (e.g., the ones for which all the relevant information is statically known), and they would consider as *out of scope* all loops that do not satisfy specific requirements. For example, consider a loop for which the number of iterations directly depends on the size of a list: since, in the general case, the size of a list cannot be statically known, this loop cannot be properly analyzed by existing approaches.

For this work, we developed a generic loop analysis framework (based on static analysis) to characterize loops under many different aspects. In particular, we focus on the Android platform, for which a vast amount and variety of applications are available. Android is fundamentally an event-driven system, and the execution of an application should be mainly driven by the underlying system. However, loops within the main application are opaque to the framework, and it is not possible to control them without using performance-intensive, fine-grained dynamic monitoring systems. These observations make loops a particularly interesting and relevant construct to be studied in the context of the Android platform.

We implemented our static analysis techniques in a tool, called CLAPP. CLAPP works directly on Dalvik bytecode, and it therefore does not rely on having access to the application's source code. At its core, our analysis extracts detailed information about the operations that influence and control the number of iterations of a loop, and the operations that constitute the loop's body. This is achieved by combining several static analysis techniques, such as loop identification, backward data-flow analysis on use-def chains, selective abstract interpretation, and code reachability analysis. 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 to characterize, under many different aspects, the usage of loops in Android applications. In particular, we analyzed 11,823 Android applications, which cumulatively contained 4,110,510 loops. As part of our evaluation, we analyzed the results of our static analysis system to gain insights related to many interesting aspects of the usage of loops in Android applications.

As one of the first analysis steps, we perform *bound analysis*, which consists in the identification and characterization of what kinds of operations control the number of iterations for a given loop. More in general, we try to characterize if loops are bounded and how often: interestingly, our system was able to determine that the execution is guaranteed to terminate for 63.28% of the loops. We also found that it was possible to statically estimate the number of iterations in the worst-case only for 2.26% of all loops: this suggests that existing loop analysis techniques might be applicable only in a very limited number of cases. As a second step, we perform *body analysis*, which consists in characterizing what kinds of operations are performed for each iteration of the loop. The results from bound and body analysis are then combined to increase the precision of the analysis and to gain even more insights. These results are used as a basis for studying the different use cases for writing loops in Android applications, and, more in general, to study the usage of loops under two main perspectives, *performance* and *security*.

First, we evaluated the performance aspect, and we determined whether loops are written according to the official guidelines provided in the Android documentation. For example, we identified cases in which developers do not use appropriate constructs (e.g., iterators when iterating over a list), which are known to provide a non-negligible performance benefit. As another example, we study whether performance-intensive operations in a loop are executed within the context of the main UI thread, in which case the user might perceive the GUI as stuck and the operating system might consider the app as non-responsive. As another interesting aspect, we identified several cases in which the Java compiler itself misses important optimizations. We also found that developers often code loops in a *risky* way, for which a small mistake might suddenly lead to the introduction of an infinite loop.

Second, we evaluated loops from the security perspective, and we highlighted cases where high-risk operations are executed in loops (hence increasing the potential for damage), or loops for which the number of iterations depends on a component *external* to the application (e.g., network data): depending on the threat model, such external components could be controlled by an attacker to perform, for example, a denial-of-service attack through the introduction of a long-running, potentially-infinite loop in the application.

In summary, this paper makes the following contributions:

- We developed a generic loop analysis framework (based on static analysis) to characterize the usage of loops in Android applications. Our framework aims to analyze *generic* loops, and it does not rely on techniques based on pattern matching.
- We implemented our static analysis in a tool called CLAPP, and we used it to perform the first large-scale empirical study to characterize the usage of 4,110,510 loops contained in 11,823 Android applications. Our prototype operates directly on Dalvik bytecode and, therefore, it does not rely on the application's source code.
- The results of our large-scale analysis offer interesting insights, such as why and how software developers implement loops in Android applications. Moreover, we show how it is possible to automatically identify *problematic* loops for what concern both the performance and the security aspect.

2. LOOPS CHARACTERIZATION

Our analysis aims at extracting as much information as possible about loops. In particular, our analysis first identifies all loops in a given application, and then focuses on the *loop control* aspect (i.e., "How is the number of the loop's iterations controlled?"), on the *loop body* aspect (i.e., "Which are the actions performed by each loop's iteration?"), and on the dependency between these two aspects.

To determine how a loop is controlled, the analysis first identifies which are the conditional instructions that can directly influence whether the loop's execution terminates. Then, our analysis performs, for each register used by these conditional instructions, a combination of backward slicing and selective abstract interpretation to reconstruct how the registers' values are initialized and evolve during the execution of each iteration of the loop. We characterize the loop's body by determining which framework API functions could be possibly invoked as part of each loop's iteration. This analysis is inter-procedural, and it works by analyzing the inter-procedural call graph of the application.

Additionally, our analysis takes into account the relationship between the operations performed in the loop, and how the loop is controlled. This allows for a more comprehensive characterization of loops. In fact, as an example, consider a loop that is bounded by the value returned by an API function call: in the general case, it is impossible to characterize how this value changes during each loop's iteration. However, in some cases, it is possible to establish that the API function call's return value would not change if, for example, the loop's

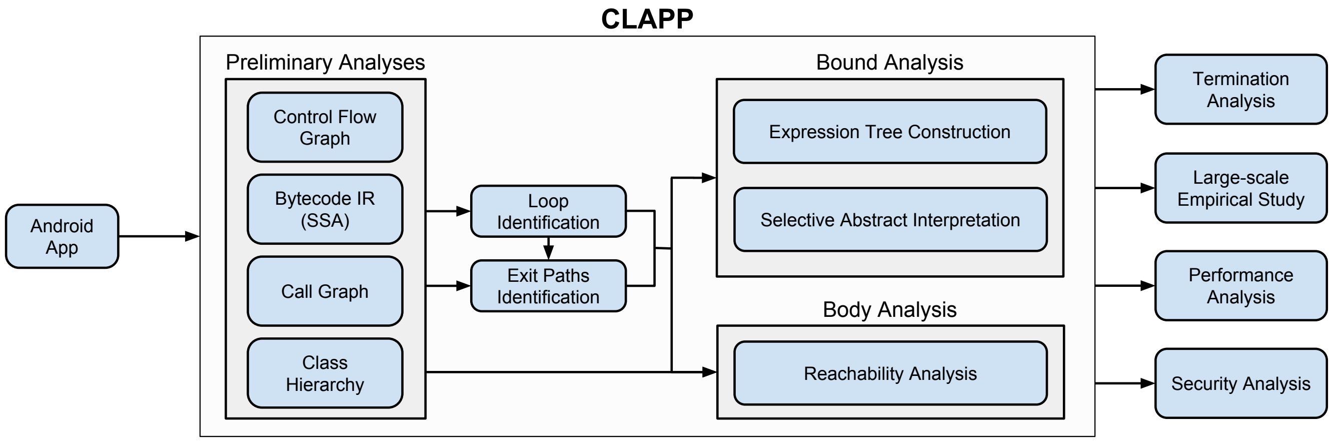


Figure 1: Overview of our loop analysis framework.

```
public void f() {
    List l = ...
    bool skipFirst = ...
    int startIdx, i;

    startIdx = skipFirst ? 1 : 0;

    for (i = startIdx; i < l.size(); i += 2) {
        Elem e = l.get(i);
        if (e == null) {
            break;
        }
        processElem(e);
    }
}
```

Figure 2: A sample program containing a loop.

body is only constituted by math-related operations. For this reason, we extract additional information from the body of the loop, and we combine them with information extracted during the bound analysis step to determine, for example, if the value returned by a given API function call can be assumed to be constant. As two concrete examples, consider the `List.size()` and `String.length()` methods: if the body of the loop does not perform any significant operations on these objects, their return values will not change after each iteration of the loop. In certain cases, this additional information is critical to prove a given loop's termination.

3. STATIC ANALYSIS FRAMEWORK

This section discusses the details of our static analysis framework, whose overview is shown in Figure 1. As input, the framework takes the APK file of an Android application (an APK is the archive format used in Android to deliver applications' code and resources), and it extracts its Dalvik bytecode, a register-based representation of the program that is executed by the so-called Dalvik Virtual Machine (DVM). Then, the analysis performs several steps, based on static analysis, that extract detailed information about the control and body aspects of each loop. These results are then post-processed to perform, for example, bound analysis and to gain insights related to the performance and security aspects. The remainder of this section describes in details all these analysis steps. Throughout our description, we will use the snippet of code in Figure 2 as a running example.

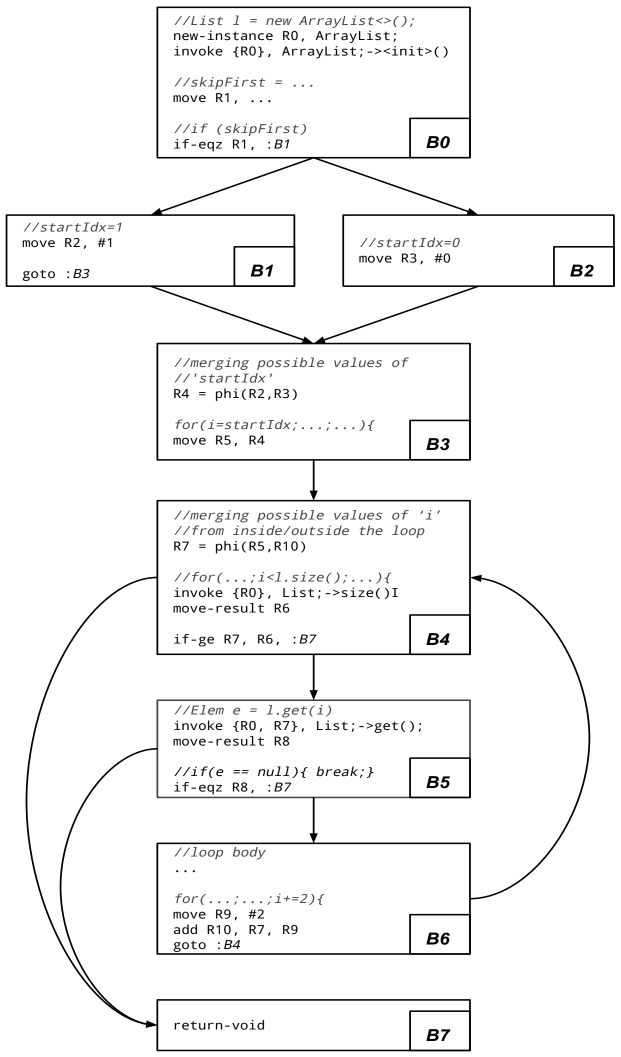
3.1 Preliminary Analysis Steps

The analysis begins by performing several preliminary steps. These steps have the goal of extracting the information on top of which our loop analysis framework is built. First, the Dalvik bytecode (DEX) of the application is parsed and disassembled. For this step, we rely on Androguard [11] to convert Dalvik bytecode into easily-accessible Python objects. After this step, we compute the Control-Flow Graph (CFG), and we lift the bytecode to an internally-developed Intermediate Representation (IR) that is more suitable for performing static analysis. One of the main features of this IR is that it comes in Static Single Assignment (SSA) form [10]. Figure 3 shows the CFG and the bytecode representation, in SSA form, corresponding to the example in Figure 2.

After this step, we perform Class Hierarchy Analysis (CHA), which reconstructs the inheritance relationship among all classes and interfaces defined in the application and the Android framework. The last of these preliminary steps is the computation of a coarse over-approximation of the inter-procedural call graph. This is performed by first identifying all the *invoke* bytecode instructions in the application code base. Then, for each of them, the analysis first determines the set of possible dynamic types of the object receiving the method invocation (this is done by using the results from the CHA step and other type-related information), and then it simulates the dynamic dispatch mechanism to compute all possible targets.

3.2 Loop Identification

After these preliminary steps, our analysis proceeds to identifying all loops in the given Android application. To do this, we implemented the algorithms proposed by Wei *et al.* [28]. In the context of our work, a loop is represented as a set of basic blocks. Among these basic blocks, the analysis then identifies the *header*, which can be seen as the entry point of the loop. More formally, the header is the basic block that *dominates* the execution of all the other blocks belonging to the loop. The algorithm proposed by Wei *et al.* can also identify and correctly handle *nested* loops. In our work, nested loops are treated exactly as non-nested loops, with the only difference that we annotate which *inner* loop belongs to which *outer* loop. For our example in Figure 2 and 3, the loop header is `B4` and the body is constituted by `B4`, `B5`, and `B6`.


```
//List l = new ArrayList<>();
new-instance R0, ArrayList;
invoke {R0}, ArrayList;-><init>()

//skipFirst = ...
move R1, ...

//if (skipFirst)
if-eqz R1, :B1
```

```
//startIdx=1
move R2, #1

goto :B3
```

```
//startIdx=0
move R3, #0
```

```
//merging possible values of
//'startIdx'
R4 = phi(R2,R3)

for(i=startIdx;...;...){
move R5, R4
```

```
//merging possible values of 'i'
//from inside/outside the loop
R7 = phi(R5,R10)

//for(...;i<l.size();...){
invoke {R0}, List;->size()I
move-result R6

if-ge R7, R6, :B7
```

```
//Elem e = l.get(i)
invoke {R0, R7}, List;->get();
move-result R8

//if(e == null){ break;}
if-eqz R8, :B7
```

```
//loop body
...

for(...;...;i+=2){
move R9, #2
add R10, R7, R9
goto :B4
```

```
return-void
```

Figure 3: Annotated control-flow graph of the function in Figure 2. The loop header is B4, and the loop's body is constituted by B4, B5, and B6.

Exit Paths Identification. The next step is to identify all possible *exit paths*. We define an exit path as a set of conditional instructions (or *conditions*) that need to be satisfied so that the execution of the loop terminates. Naturally, a loop might have multiple exit paths. For example, the snippet of code in Figure 2 has two exit paths: 1) {`i >= l.size()`}; 2) {`i < i.size()`, `e == null`}.

To determine all exit paths, we first identify all basic blocks (belonging to the loop's body) for which at least one of their successors is a basic block that is *not* part of the loop. These are the basic blocks that contain conditional instructions (i.e., `if-*` bytecode instructions) directly guarding the loop's *exit*. Then, the analysis computes all control dependence paths to each of these basic blocks from the loop header, by using the algorithm proposed in [17]. The resulting set of paths constitutes all possible loop exit paths. For our example in Figure 3, the exit paths are: 1) {(`if-ge R7,R6`)}; 2) {(`!if-ge R7,R6`), (`if-eqz R8`)}.

3.3 Loop Control Characterization

One of the main aspects that our analysis characterizes is the *control* aspect. That is, we are interested in characterizing which are the factors that control the number of iterations of the loop to determine, for example, if it is possible to statically guarantee that the loop (eventually) terminates. This analysis step operates on the exit paths identified as part of the previous preliminary steps. Depending on the number of exit paths and their type, we label a loop as *simple* or *complex*. In particular, we consider a loop as simple if it has only one exit path constituted by only one condition. In other words, a loop is simple if there is only one `if-*` bytecode instruction that determines whether the loop should terminate its execution or perform another iteration. Instead, all loops that have more than one exit path, or that have only one exit path but multiple conditions, are considered as complex. Given this definition, the loop in Figure 2 is considered complex, since it has two different exit paths. Note that it is possible that a loop might not have any (explicit) exit path: in these cases, the loop is labeled as *Potentially Infinite*. We discuss this last case in Section 4.

3.3.1 Overview

From a high-level point of view, our analysis works by characterizing each `if-*` bytecode instruction independently. More precisely, our analysis works by first determining which are the relevant registers (for each of these instructions), and by then analyzing how the registers' values are initialized and how they evolve during the iterations of the loop. This is done by first reconstructing the *expression tree* that encodes how the value contained in a register is updated after each iteration, and by then performing *selective abstract interpretation* to extract some relevant properties. This information is then used to establish if it is possible to statically determine whether the condition specified in the `if-*` instruction will be eventually satisfied (in other words, if it possible to determine whether the loop will terminate), and to characterize which kind of operations influence the control. These analysis steps are described in Section 3.3.2, 3.3.3, and 3.3.4.

The described analysis steps focus on the analysis of a single `if-*` instruction, which is enough to characterize simple loops. For complex loops, the analysis first studies all the relevant `if-*` instructions independently, and then it combines the extracted information together, as explained in Section 3.3.5.

3.3.2 Expression Trees

For each `if-*` bytecode instruction, the analysis first determines used registers, and reconstructs the operations that update their values after each iteration.[1] In particular, the analysis creates, for each register, an *expression tree* that encodes how the register is updated after each iteration. Specifically, each node of the tree is characterized by a type of operation (e.g., scalar addition) and by its operands, which, in turn, can be a value (e.g., a scalar value, the return value of a method invocation), or another expression tree. The expression trees computed by our analysis are intra-procedural, and they are obtained by performing a backward traversal of the use-def chains (provided by the SSA form). It is important to note that, since we work with an IR in SSA form, each assignment to a Dalvik register will *produce* a new register in the SSA form. Thus, in our representation, each register is associated with one and only one expression tree.

[1]In Dalvik bytecode, conditional instructions have either one or two registers as their operands. In case the conditional instruction has only one register, the comparison is implicitly performed against the value zero.

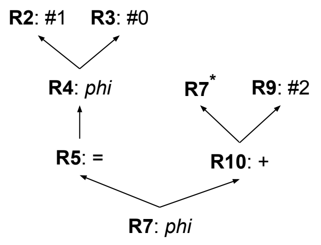


Figure 4: Expression tree associated with register `R7` of the function `f` defined in Figure 2 and 3. The `*` suffix identifies a cyclic dependency in the tree: in fact, the *next* value of `R7` depends on the *current* one.

Note that while performing the backward traversal, the analysis will inevitably encounter loops in the use-def chains. Intuitively, a loop in the use-def chain encodes the fact that a register's value depends on a previous version of itself. Our analysis handles these cyclic dependencies by not continuing the exploration of registers that have already been processed (i.e., those registers that are already present in a path from the root of the expression tree to the current node). Of course, the analysis will annotate the non-explored registers with a special flag, so that the subsequent analysis step (i.e., the selective abstract interpretation) can properly handle this situation. As an explanatory example on what the output of this step looks like, we report in Figure 4 the expression tree associated to register `R7`.

3.3.3 Selective Abstract Interpretation

The next step of the analysis consists in performing selective abstract interpretation on top of the expression trees computed for each register. In particular, this analysis step annotates each node of the expression tree to characterize how the value of a given register is initialized, how it evolves during each iteration, and the *trend* of the value, which consists in studying whether the value a register can assume is *bounded*, regardless on the number of iterations. The result of this step is an annotated expression tree. In the remainder of this section, we first describe our annotation system, and we then discuss how these annotations are computed.

An annotation for a node consists of a set of labels. We now define each of these labels, we discuss the possible values they can assume, and, for each of them, we provide both an intuitive and formal definition. In this section, we will use the notation $(R_x)^n$ to indicate the value assumed by the register R_x after n iterations of the loop.

Statically bounded. This label can be set to `true` or `false` depending on whether the value stored in the target register R_x can be statically determined as bounded. In other words, this flag is set to `true` if it is possible to statically extract two values ξ_1,ξ_2 such that $\forall n\colon \xi_1 < (R_x)^n < \xi_2$.

Fixed. This label can be set to `true` or `false` depending on whether the value of the target register R_x does not change during the execution of the loop. Formally, this flag is set to `true` if the analysis can guarantee that $\exists \xi\, \forall n\colon (R_x)^n = \xi$. Note the difference from the previous case, the value ξ must exist, but it is not required to be statically computable. This label is set to `true` for any register that is defined outside the loop, since their value is guaranteed to not change during the loop's iterations: In fact, the SSA form guarantees that a register can be defined only once, and, in case of a re-assignment within the loop's body, a new register would be created.

Sign. This label can be set to one of the following values: `positive`, `zero`, `negative`, or `unknown`. This label encodes information related to the sign assumed by the register's value. For example, the label will be set to `positive` if the analysis is able to determine that $\forall n\colon (R_x)^n > 0$. Instead, if the analysis cannot determine the sign of the value, this label is set to `unknown`.

Value Trend. This label indicates how the value of a register R_x evolves *at the limit*. In particular, this label indicates whether it is possible to guarantee (through static analysis) that a value will *eventually increase* (or *decrease*), that a value is *bounded*, or that a value will be *eventually zero*.[2] Concretely, this label will be set to one of the following values:

- **Increasing (>).** This indicates that the register's value will *eventually increase*. Intuitively, this implies that, given a sufficient number of iterations, the value in the target register will necessary be greater than any arbitrary value. Formally, the following condition needs to be satisfied: $\forall \xi, \exists n\colon (R_x)^n > \xi$. Note that this definition is weaker than the classic definition of *monotonically increasing*. This is intentional, as in most cases our (weaker) definition is enough to extract useful properties about the bound of the loop.
- **Decreasing (<).** Similar to the previous case, this indicates that the value in the register will *eventually decrease*. Formally, $\forall \xi, \exists n\colon (R_x)^n < \xi$.
- **Bounded (=).** This indicates that the value assumed by the given register is *bounded*. Intuitively, this indicates that there exist two values such that it is guaranteed that the value of the register will always be *between them*. Formally, $\exists \xi_1,\xi_2\; \forall n\colon \xi_1 < (R_x)^n < \xi_2$. Note that this value indicates something different than the Statically Bounded label. In fact, the values ξ_1,ξ_2 must exist, but they are not required to be statically computable.
- **Eventually Zero (0).** This indicates that the value of the register will be *eventually zero*. Formally, $\exists n\colon (R_x)^n = 0$.
- **Unknown (?).** This indicates that our analysis cannot precisely determine the trend of this register's value.

This labeling system proved to be really powerful and generic. In fact, as our evaluation shows, this mechanism allowed us to characterize a very large number of loops, without having to rely on techniques based on pattern matching. We chose this specific set of labels as they are helpful in modeling and capturing the common behavior and evolution of the registers' values that play a key role in controlling the number of iterations in a loop. For example, loops are often controlled by a variable that starts from a given value and increases at each iteration (e.g., the loop in Figure 2). In this case, our analysis will be able to capture exactly this property, by assigning the value *Increasing* to the *Value Trend* label. As we will discuss in the next section, this aspect of our work plays a key role when determining whether the number of loop iterations is bounded. We also note that some of the labels are introduced specifically to precisely model known API functions that are often used in Android. For example, the *Eventually Zero* value is helpful when modeling the return value of the `Iterator.hasNext()` API method that, under certain assumptions (see Section 3.5), is known to *eventually* return false (or zero).

We are now ready to discuss and explain how these labels are computed for each node of the expression tree. The system first assigns an *initial* label for each node in the tree without

[2]In the context of this paper, we do not consider the possibility of integer overflows.

children (i.e., leaf nodes). For example, a register that is set by the "`const/4 R5 #42`" bytecode instruction (which moves the constant #42 in register `R5`) will be annotated with the *Statically Bound* and *Fixed* labels set to `true`, the *Sign* label set to `positive`, and the *Value Trend* label set to *Bounded*.

When such precise information is not available, our analysis takes into account the operations performed within the body of the loop to extract more precise, meaningful labels. One common case is when a relevant register is set to the return value of a known framework API function. For example, if a register is set to the return value of the `List.size()` API function, the analysis first determines whether the list could be modified during the execution of each iteration, and, if not possible, the analysis is able to set the *Value Trend* label to *Bounded*. More in general, the analysis combines information about the control and the body of the loop. This aspect of our analysis is described in Section 3.5.

Note that the computation of the *initial* values is more challenging when the tree contains cyclic dependencies. For example, consider the register `R7` in Figure 4: the value that `R7` will assume during the *next* loop iteration, depends on the value assumed during the *current* one. This makes the analysis challenging especially when computing the value for the *Value Trend* label. In fact, in order to set this label to, for example, the *Increasing* value, the analyzer must first *prove* that one of the *next* values assumed by a given register will be greater than the *current* value. The analysis is able to reconstruct this relation by identifying cyclic dependencies in the expression trees and by determining, for example, whether the *next* version is the result of the addition of the *current* version and a quantity known to be positive (as for register `R7` in the example of Figure 4). In this case, this would be enough to prove that the register's value will *eventually increase*. This analysis step is implemented by iteratively propagating the information available, from the leaves to the root of the tree, until convergence is reached. Note that we called this analysis step *Selective Abstract Interpretation* because we are actually performing abstract interpretation (where the abstract domain is represented by our annotation system), by only considering those instructions that are relevant for the loop's bound analysis, i.e., the ones that appear in the associated expression trees.

Once the initial values are computed for all the leaves, propagating these labels throughout the entire tree is conceptually trivial. This is achieved by iteratively *merging*, for each node, the labels associated to their children nodes. The merge operation for all the annotations labels (i.e., *Statically Bounded*, *Fixed*, *Sign*, and *Bound Analysis*) is trivially implemented through a set of tables (one for each operand) that specify what the output label should be given two labels as input. For example, the addition of two registers that are known to be positive, is positive. As another example, consider a register that, at each iteration, is set to the subtraction of a register whose *Bound Analysis* label is set to *Decreasing*, and a register whose value is known to be positive: clearly, the *Bound Analysis* label associated to the *addition* node will be set to *Decreasing* as well.

3.3.4 Characterizing Conditional Instructions

Once all registers of a given conditional instruction are properly annotated, it is conceptually simple to characterize it. For example, the *Value Trend* labels of the registers can be used to determine whether the loop is guaranteed to terminate: If the first register is known to *increase*, and the second register is known to be *bounded* (or *decreasing*), then it is easy to see that the conditional instruction will be *eventually satisfied*, and hence that the loop will eventually terminate. In some cases, it is even possible to statically determine the number of iterations in the worst possible case. This is done by consulting the *Statically Bounded* labels of each register. Moreover, the information encoded in the expression trees can be used not only to perform bound analysis, but also to determine whether a given conditional instruction depends on the return value of specific methods, fields, or other factors. This additional information is useful to characterize which categories of API functions control the number of iterations of a given loop.

3.3.5 Generalization to Complex Loops

All analysis steps described so far focus on the characterization of a single conditional instruction. We now discuss how our analysis can be generalized to complex loops. For what concern the bound analysis, our analysis proceeds by considering all exit paths, and it reconstructs a boolean formula that represents the combination of all of them. Then, our system minimizes the boolean formula (by applying well-known simplification techniques), and converts it to a canonical representation. At this point, our analysis determines whether a given complex loop is guaranteed to terminate according to the following two observations: The conjunction of two conditions is *eventually satisfied* if and only if both the conditions are known to be *eventually satisfied*; The disjunction of two conditions is *eventually satisfied* if either one of the two conditions is *eventually satisfied*. By following these simple rules, our analysis is able to characterize complex loops as well. As an explanatory example, for the loop in Figure 2, the minimized boolean formula that represents all exit paths is: `i >= l.size()` $\vee$ (`i < i.size()` $\wedge$ `e == null`). Thus, the analysis can establish that the loop will terminate since the variable `i` (stored in `R7`) is known to *eventually increase*, while `l.size()` (stored in `R6`) is known to be a fixed value.

3.4 Loop Body Characterization

This section discusses how our analysis characterizes the behavior and the body of a given loop. From a high-level point of view, our analysis aims to determine the set of framework API functions that might be possibly invoked (intra- and inter-procedurally) within the context of a loop's iteration. To do that, the analysis first considers all basic blocks that belong to the body of the loop. Then, it identifies all `invoke-*` bytecode instruction, and, for each of them, the previously-computed inter-procedural call graph is consulted. For each potential target, the analysis proceeds according to the following algorithm: if the target is a framework method, then this method is added to the set of framework methods that could be potentially invoked; alternatively, if the target method is a method defined within the application, the analysis adds to the set of possible targets all the methods that are (directly, or indirectly) reachable by traversing the call graph.

As the next step, each framework method is associated with a fine-grained label that indicates what type of operation it performs. These labels are assigned according to a manually-written configuration file, which specifies which label should be assigned to which method. To compile this configuration file, we started by consulting the results from PScout [6] and SuSi [26], and then we augmented their results and annotations by consulting the Android documentation. Our configuration file currently specifies about 650 entries. Note

Table 1: Breakdown of the results related to the characterization on loops control aspect.

	Simple	Complex	No Exit Paths	Total
Bounded	2,060,485	540,755	0	2,601,240
Risky	20,431	4,411	0	24,842
Unknown	1,108,471	294,707	0	1,403,178
Potentially Infinite	890	1,816	3,550	6,256
Non-Supported	5,842	69,152	0	74,994
Total	3,196,119	910,841	3,550	4,110,510

Table 2: Number of loops controlled by various categories of framework APIs.

Category of API	Number of Invocations
Iterators	761,394
Parsing	630,076
GUI-related	474,683
Input/Output	439,558
Data Structure	296,541
Crypto	279,234
Information Gathering	175,307
User Data Access	165,465
Network	105,662
Polling Peripherals Status	64,860

that each entry often describes multiple methods through the usage of regular expressions. Section 4 reports the details about the labels currently supported by our analysis.

3.5 Dependency Between Control and Body

In some circumstances, to assign a precise control label (as the ones described in Section 3.3) to the return value of a method, it is necessary to characterize the operations performed in the body of the loop. For example, consider a loop bounded by the `List.size()` API method. In many cases, the size of the list will not change after each iteration. However, it could be possible that the loop's body modifies the list, thus changing its size. Similarly, consider the example of the `Iterator.hasNext()` API method. The `hasNext` method is known to return zero when no more items can be processed, which, under normal conditions, is eventually going to happen. However, if the body of the loop does not invoke the `Iterator.next()` API method (to *process* the current item in the list), the `hasNext` API method will never return the value zero, thus functionally creating an infinite loop.

For this reason, we extended our static analysis framework so that the actions in the body are taken into account when characterizing the return value of some known API functions. For example, when a loop is bounded by the `List.size()` API method, our analysis verifies that there is no path (within the body of the loop) that might invoke a method to modify the list: if that is the case, the analysis will label the return value of the `size()` API method as a *fixed* value. Similarly for the `hasNext()` example, our analysis verifies that all possible paths invoke the `next()` method, in which case the analysis will be able to label the return value of the `hasNext()` API method as *Eventually Zero*. As part of the process, our analysis also performs a conservative intra-procedural on-demand alias analysis step to determine whether the body of the loop operates on the same (or different) object than the one used for control. This simple, but effective technique proved to significantly improve our results both in terms of precision and performance.

4. EMPIRICAL STUDY

In this section, we discuss how we used our static analysis framework to perform the first large-scale empirical study on why and how Android applications make use of loops. First, we describe the dataset we used for our study. Then, we discuss the results of the bound and body analysis, by also including several insights related to the performance and security aspects.

4.1 Dataset

To build our dataset, we considered the applications collected by the PlayDrone project [27]. Essentially, PlayDrone is a crawler for the official Google Play Store, and it has been used to crawl more than one million applications between 2013 and 2014. We opted to use this dataset because it is the most representative source of Android apps that spans over the entire market, and because this dataset is publicly accessible. For our experiments, we selected, at random, a subset of 15,240 applications. These applications span over several categories on the market store, and they contain hundreds to several thousands methods, depending on the complexity of the application and the libraries they include.

4.2 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 was reached (given the size of the dataset and the complexity of the analysis, we opted to enforce a timeout of 30 minutes for the analysis of each application). For the applications that were successfully processed, our tool identified and analyzed a cumulative total of 4,110,510 loops, and a total of 118,190,014 API framework methods that could potentially be invoked in these loops. On average, analyzing each application took 96.77 seconds, and analyzing each loop took 50.86 seconds.

4.3 Loop Control and Bound Analysis

In this section, we will discuss the results of our analysis that are related to how the number of iteration of each loop is controlled. As a first aspect, 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 was no exit path associated to it. We discuss this case later in this section (see the *Potentially Infinite* paragraph). As another interesting statistic, we found that 266,667 (6.48%) of the loops contain at least one nested loop.

Another important aspect that our analysis helps characterizing is related to the following question: how often it is possible to statically determine whether a loop can be guaranteed to terminate? Table 1 reports the details about the different categories of our results. In particular, it shows the breakdown of the bound analysis with respect to the complexity of the loop. The remainder of this section discusses the different analysis results.

Bounded Loops. Our analysis was able to determine that 2,601,240 of the loops are guaranteed to terminate (indicated with *Bounded* in the table). This constitutes an interesting result, as these loops correspond to a substantial portion (63.28%) of the analyzed loops. In fact, it shows that, even if determining whether a loop terminates is an undecidable problem in the general case, it is possible to provide an answer in a surprisingly-high number of cases. Another interesting aspect is that the method annotations and alias analysis described in the previous sections were critical to prove termination for 1,037,105 loops. Moreover, our tool identified 92,795 (2.26%) loops for which the number of iterations in the worst case can be statically determined. Note that most existing techniques only focus on this (small) set of loops' categories, which indicates the need for more generic loop analysis techniques, like the one proposed in this paper.

Risky Loops. Our analysis identified 24,842 (0.60%) loops that are implemented in a *risky* way (indicated as *Risky* in the table). With the term "risky," we refer to loops implemented so that, independently from whether they terminate or not, a subtle change in the loop's body might cause the loop 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. A much safer yet equivalent alternative to implement the loop in example would be to convert the *different than* comparison (`!=`) to a *less than* comparison (`<`).

Loops with Unknown Bound. For 34.16% of the loops, our analysis was not able to determine whether the execution is bounded or not (indicated as *Unknown* in the table). One of the root causes for which our static analysis framework cannot determine whether a loop is bounded or not is constituted by the fact that the number of iterations of a loop can be influenced by the return value of a method invocation. In 512,675 cases, the number of iterations is controlled by the return value of the invocation of a method implemented in the application. Alternatively, the loop could be controlled by the value returned by a framework API function. As part of our experiments, we explored which categories of API functions are controlling the execution of loops more often. The most frequent entries are reported in Table 2. Unsurprisingly, the most common API functions are those that associated with iteration and parsing. However, other results are more interesting: For example, our analysis identified 105,662 cases where a loop is directly controlled by network-related API functions. Although this is innocuous in most scenarios, a loop that depends on an *external component* might have several security-related implications, which we discuss at the end of this section.

Potentially Infinite and Non-Supported Loops. A minor portion of the loops (6,256 in total) were classified as *Potentially Infinite*. For these loops, our analysis determined that there was no (explicit) exit path in their control-flow graph. Clearly, this property indicates that a loop *might* be an infinite loop, but, of course, it is not necessarily the case. In fact, these loops might have implicit exit paths implemented by means of exceptions or by means of concurrently modifying values in different threads. These two aspects are not supported by our prototype, thus making it currently impossible to discern whether a loop is actually infinite or not. Nonetheless, we believe these loops represent cases of poorly-implemented functionality, regardless on whether they are infinite or not. In fact, in these cases, the loop's termination condition is implemented in a different place than the loop itself, and this decreases the readability of the code. Moreover, the Android framework is fundamentally event-driven and it offers to a developer a plethora of callback-based mechanisms to avoid implementing loops whose termination condition is *triggered* from a thread different than the one executing the loop. To make things worse, aggressive power savings in Android [2] often pause and later resume an application, potentially rendering these loops infinite, if not carefully implemented. Finally, our analyzer was not able to analyze 74,994 loops, the reason being that they rely on switch-like bytecode instructions (instead of the simpler `if-*` bytecode instruction), which our prototype currently does not fully support. Of course, the analysis can be easily extended to handle these cases as well, which is left as future work.

Performance Aspect. Our analysis can be useful to identify (potential) missed opportunities for performance-related optimizations. In particular, our analysis determined that for 259,014 loops (6.30% of the total), the body of the loop repeatedly invokes an API function whose return value does not change after each iteration nor has side effects. For example, this situation arises when a developer writes a loop to iterate over the items of a list by using an *index* variable that goes from zero to the `size()` of the structure: If the structure is not modified, the return value of the `size()` API method would not change after each iterations, and hence its return value could be cached.

This kind of situations can arise for two reasons. First, it might be a developer's mistake: she could write a loop that invokes an API function within the body of a loop, instead of invoking it – just once – before the loop's first iteration. In this case, the value could have been cached. Second, it could be that the compiler itself does not have enough information to determine whether the return value of the API function is going to change or not. Hence, it has no other choice than invoking the API function for every iteration.

Although our analysis is not precise enough to identify all missed opportunities (hence, it might be affected by false negatives), the number of problematic loops we identified is already non-negligible. Moreover, we believe these optimizations could lead to important performance boosts, since they would prevent the invocation of framework API functions. As another observation, we note that in all these cases, it would have been possible to use one of the known Java constructs, such as iterators, which are known to improve the performance. Specifically, on the Android platform, index-based iterations are known to be slower than by using iterators, explicitly or implicitly through the "enhanced" *for-each* ":" syntax. This aspect is specifically mentioned in the performance-related tips in the official Android documentation [4].

Security Aspect. The analysis on how the number of iterations is controlled can be used to gain insights related to the security aspect as well. In fact, loops could be intentionally written to be infinite loops, so to make the device unusable and drain its battery. Our loop control analysis helps in highlighting these problematic cases, by identifying *potentially infinite* loops that do not have any explicit exit paths, or whose exit conditions do not seem to be satisfiable. As we already mentioned, most of these loops, in practice, do terminate. However, as the number of *warnings* only represents the 0.15% of the total number of loops, we believe it is worth it to check them manually. At the very least, these cases might indicate a poorly-implemented loop, thus deserving human attention anyways.

Table 3: Breakdown of the number of loops that can possibly invoke a method with a given semantic.

	Invoked Within Loop	Invoked Within UI/Main Thread
Alarm	35,598	25,293
Android-Specific	1,001,833	772,053
Audio	58,931	44,856
Bluetooth	60,590	39,528
Camera	5,944	5,286
Concurrency	1,247,208	882,272
Crypto	2,445,049	1,786,780
Data Structure	3,552,636	2,519,611
Device Settings	2,330	1,877
Exception	505,232	339,887
Face Detector	173	156
Garbage Collector	15,539	11,089
GUI	3,420,748	2,546,585
Device Data	1,992,174	1,473,369
Iterators	3,960,567	2,666,318
Intent	831,777	635,166
Internals	319,602	228,487
Input/Output	2,975,636	2,145,467
Keyguard	6,270	4,898
Log	945,011	706,759
Multithread	445,365	351,762
Network	1,057,628	764,240
NFC	60,372	39,363
Object Comparison	638,745	479,699
Parsing	1,967,496	1,452,295
Phone	696,440	522,025
Power Manager	298,692	210,882
Privileged Operation	8,095	3,426
Process	6,391	5,187
Random	728,221	537,012
Reflection	933,875	680,074
Resource	296,015	221,096
Sensor	6,869	4,882
SMS	796	709
Speech	2	1
StrictMode-related	295,617	207,888
String	432,772	298,039
Telephony	688,520	514,868
Usb	691,891	518,117
Userdata	2,647,813	1,976,974
Video	315,706	225,922
VPN	21	19
Webkit	35,554	27,806
WIFI	297,730	209,340

Another interesting aspect related to security is that we identified some loops whose number of iterations depends from a factor *external* to the application itself. For example, we identified 105,662 loops that depend on network input. In certain scenarios, this external dependency might cause issues: depending on the threat model, the attacker might have a chance to alter network data and create an infinite (or, at the very least, a very long-running) loop by sending properly-crafted data to the device.

4.4 Loop Body and Behavior Analysis

As we discussed in Section 3.4, our analysis characterizes the operations performed by the body of the loop by determining the framework API functions that could potentially be invoked from within the loop body. This is done by both considering the class hierarchy analysis (to conservatively take into account the dynamic dispatch mechanism) and by consulting the inter-procedural call graph. Then, our system associates a semantic label to the reached framework methods (according to the manually-written policies described in Section 3.4). Table 3 reports how many loops have been found to possibly invoke a method with a given semantic label.

Our results 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 (`Iterators`), perform cryptographic operations (`Crypto`), generating random numbers (`Random`), parsing data (`Parsing`), iterating over different data structures (`Data Structure`). Thus, the majority of loops appear to be innocuous. Loops also use Android-specific methods, identified in Table 3 by the following labels: `Android-Specific`, `Face Detector`, `Device Data`, `User Data`. However, as we will discuss in the following paragraphs, we also found that a non-negligible portion of loops might invoke several APIs that, in an event-driven system like the Android platform, can impact both performance and security.

Finally, our analysis found that 306,593 (7.45%) loops do not invoke any framework API functions. After manual investigation of few samples, we found out the most common case to be one of the following: either the loop performs some sort of mathematic computation, or it updates application-specific objects' fields.

Performance Aspect. Every Android application is executed by several threads: one UI thread, which takes care of handling user interaction, and several other non-UI threads. Since the UI thread is in charge of executing the *interactive* part of an application, blocking tasks, such as I/O operations or network connections, should always be executed in a non-UI thread. Doing otherwise is specifically discouraged by the official guidelines for Android developers [4]. This is because Android has strict time constraints on UI threads: If a UI thread is not ready to handle a UI action, then the application will be terminated with the infamous Application Not Responding (ANR) error message [1].

This aspect is so problematic that a recent version of Android introduced StrictMode [3], which is, quoting the official documentation, "a developer tool which detects things you might be doing by accident and brings them to your attention so you can fix them." In particular, it is often used to "catch accidental disk or network access on the application's main thread." However, this mechanism can be explicitly disabled by an Android application. For example, to allow the invocation of network-related APIs within the UI thread, an application can invoke: `android.os.StrictMode$ThreadPolicy$Builder!permitNetwork()`.

To analyze this potentially-problematic aspect, we implemented a simple analysis step to determine whether a given loop could be executed within a UI thread. In particular, this step performs reachability analysis starting from a given method, and it proceeds backward until the entry points are reached. Then, the analysis flags a loop as potentially-executed within a UI thread if at least one of these entry points is a method known to be associated to the

GUI (i.e., `Activity.onCreate()`). Note that this analysis step might be affected by imprecisions. Nonetheless, we believe this to be an interesting experiment.

The breakdown of our results is reported in Table 3. For example, our analysis identified that 1,057,628 loops could potentially invoke network-related API functions (`Network`), 764,240 of which might be invoked within a UI thread. More importantly, our analysis identified 207,888 loops that explicitly invoke at least one `StrictMode`-related API function within a UI thread, and thus could create performance-related issues. We note that, although these loops are not malicious, they do not follow the suggested guidelines, and hence we think it is interesting to report them.

Security Aspect. The security relevance of the invocation of an API function can increase depending whether it is executed within the context of a loop: the mere fact that a method can be invoked *multiple times* makes a given operation intrinsically more dangerous. For instance, a single invocation of the `File.delete()` method (which can be used to delete a file) is practically innocuous: However, if this method is executed within a loop, then the application has suddenly the capability to wipe out the phone's data, and thus should be considered as suspicious, or, at least, should be inspected by an analyst. Our analysis comes in handy because it provides detailed labels for a loop body indicating the kind of operations it can perform. Thus, these labels can be used to identify suspicious loops. A very interesting example we found is the following: our analysis identified 442 loops that repeatedly invoke the `android.os.Debug!isDebuggerConnected()` framework method. This method is used to check whether the application is being debugged or not. However, this very same API function can also be used by a malicious application to avoid debugging, and it should thus be considered potentially-suspicious.

5. RELATED WORK

Previous research has been focused on automatically analyzing loops in programs using techniques similar to the ones we used, but with different goals. Specifically, loop analysis has been often used for Worst-Case Execution Time (WCET) analysis, whose goal is to determine the maximum length in time of the execution of a given functionality. Clearly, the analysis of loops plays a key role when precisely estimating the worst-case time, and several approaches have been proposed in the past. Specifically, [22] uses a pattern-base approach to determine the upper bound of the number of iterations for specific classes of program loops. Other approaches, such as [13, 21], achieve the same goal by using program analysis and abstract interpretation, similarly to what we implemented in this work. Different approaches have been proposed too. For instance, [9] develops an approach based on data-flow and compares it with the approach used by aiT [16] (an automatic tool to detect timing behavior of safety-critical software) showing strengths and weaknesses of both.

Other works focus on designing "compilation passes" to optimize the compilation of loops. For instance, many tools focus on analyzing loops to automatically rewrite them to exploit parallelism, both in CPUs [29] and GPUs [30]. Refer to [7, 23] for an introduction to loop-optimization techniques used, respectively, in GCC and LLVM. The tool we developed uses some of the techniques presented in these works, but with a different goal. Our goal is not to precisely measure a program's execution time nor to improve code compilation, but to automatically identify problematic cases of loop usage, to help developers or the market-scale vetting process of applications. In addition, we use our tool to perform a large-scale analysis of Android applications to automatically obtain insights about why and how loops are currently used by developers.

Although it is an undecidable problem in the general case, previous research also focused on automatically verifying loops' termination in programs written both in C [15] and Java [8]. Our tool can also be used to detect non-terminating loops, however our goal is much broader. In fact, we designed our analysis to be more generic and, to this end, we *relaxed* the requirement of exactly understanding the number of iterations of a given loop. In this way, our analysis is able to extract interesting insights even for loops whose number of iterations cannot be precisely computed by using static analysis.

Static analysis has been also extensively used to automatically detect a variety of risky programming practices and vulnerabilities (e.g., [25, 14, 24, 12, 31, 19, 18]) in Android applications. However, to the best of our knowledge, our work is the first to specifically focus on automatically analyzing problems related to loops, which, given the event-driven nature of Android, play a key role when analyzing applications. In addition, static analysis has been also used to identify malware in the Android ecosystem (e.g., [20, 32, 5]). Although our tool does not focus specifically on malware detection, it can be used to identify specific malicious behaviors (such as time-consuming or non-terminating loops, exhausting device's resources or device's battery) that are not within reach of current generic malware-detection systems.

6. CONCLUSIONS

In this paper, we presented CLAPP, a tool to automatically extract information about many aspects of loops, such as how they are controlled, and their body and behavior. For this work, we developed our analysis for the Android system, and we performed the first large-scale study on how and why loops are used in Android apps. In particular, we used our tool to analyze 4,110,510 loops contained in 11,823 distinct Android applications, and we discussed several insights related to the performance and security aspects. In the future, we envision CLAPP to be used to help developers in identifying incorrectly-programmed loops, and to assist the market-scale application vetting process by pinpointing applications containing suspicious or risky loops that should be manually investigated.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback. We would also like to thank Jacopo Corbetta, for his valuable help and many suggestions related to the large-scale evaluation, and Ali Zand, for the insightful discussion about the terminology used in an earlier draft of this paper. This material is based on research sponsored by DARPA under agreements number FA8750-12-2-0101 and FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The work was also supported by the Office of Naval Research (ONR) under grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, by the National Science Foundation under grant CNS-1408632, by Lastline, Inc., and Secure Business Austria.

8. ARTIFACT EVALUATION

The Clapp analysis system has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations.

9. REFERENCES

[1] Android Official Documentation – Application Not Responding Dialog. `http://developer.android.com/training/articles/perf-anr.html`, 2012.

[2] Android Official Documentation – Pausing and Resuming an Activity. `http://developer.android.com/training/basics/activity-lifecycle/pausing.html`, 2012.

[3] Android Official Documentation – StrictMode. `http://developer.android.com/reference/android/os/StrictMode.html`, 2012.

[4] Android Official Documentation – Use Enhanced For Loop Syntax. `http://developer.android.com/training/articles/perf-tips.html#Loops`, 2012.

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2014.

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

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

[8] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated Detection of non-termination and NullPointerExceptions for Java Bytecode. In *Formal Verification of Object-Oriented Software (FoVeOOS)*, 2012.

[9] C. Cullmann and F. Martin. Data-Flow Based Detection of Loop Bounds. In *Workshop on Worst-Case Execution Time Analysis (WCET)*, 2007.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[11] A. Desnos. Androguard. `http://code,google,com/p/androguard`.

[12] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[13] 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.

[14] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)Security. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[15] S. Falke, D. Kapur, and C. Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *International Conference on Rewriting Techniques and Applications (RTA)*, 2011.

[16] C. Ferdinand and R. Heckmann. aiT: Worst-case Execution Time Prediction by Static Program Analysis. In *Building the Information Society*. 2004.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987.

[18] Google. Android Lint. `http://developer.android.com/tools/help/lint.html`.

[19] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2012.

[20] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[21] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution. In *IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[22] J. Knoop, L. Kovacs, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Perspectives of Systems Informatics*. 2012.

[23] LLVM. ScalarEvolution and Loop Optimization. `http://llvm.org/devmtg/2009-10/ScalarEvolutionAndLoopOptimization.pdf`.

[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[25] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2014.

[26] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2014.

[27] 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.

[28] T. Wei, J. Mao, W. Zou, and Y. Chen. A New Algorithm for Identifying Loops in Decompilation. In *International Conference on Static Analysis (SAS)*. 2007.

[29] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[30] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[31] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2013.

[32] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, 2012.