Poster: Privacy-Preserving Offloading of Mobile App to the Public Cloud

Yue Duan Mu Zhang Heng Yin Yuzhe Tang
Department of EECS, Syracuse University, Syracuse, NY, USA
{yuduan,muzhang,heyin,ytang100}@syr.edu

***Abstract*— To support intensive computations on resource-restricting mobile devices, studies have been made to enable the offloading of a part of a mobile program to the cloud. However, none of the existing approaches considers user privacy when transmitting code and data off the device, resulting in potential privacy breach. In this poster, we present the design and implementation of a system that automatically performs fine-grained privacy-preserving Android app offloading. It utilizes static analysis and bytecode instrumentation techniques to ensure transparent and efficient Android app offloading while preserving user privacy. We conduct the evaluation of effectiveness and performance of our system using two Android apps. Experimental results show that our offloading technique can effectively preserve user privacy while reducing hardware resource consumption at the same time.**

I. Introduction

Enabled by various mobile devices (ranging from tablets, smartphones to emerging wearable devices), modern mobile computing grows increasingly popular. Recently, sophisticated mobile applications (e.g. photo-editing Android apps) are developed. The limited hardware resources however present a major performance problem for supporting those computation-intensive applications. To address the problem, mobile application offloading has been proposed [1], [2] to alleviate the workload on the mobile devices by offloading the computation-intensive portion of program execution to the cloud.

Mobile app offloading, while reducing resource consumption, may leak user privacy. To be specific, mobile app offloading needs to send data to the public cloud (e.g. Amazon AWS or MS Azure) to enable the program execution there. The data sent which may contain sensitive personal information (e.g. user location) would leak the user privacy; the problem compounds especially when the public clouds are deemed untrustworthy, evidenced by various security incidents (due to attacks, hacks or "evil" nature of the cloud service companies). This potential privacy-breach problem of mobile app offloading, if not treated appropriately, could become an obstacle for the use in practice.

While most existing research work focuses on identifying computation-intensive portion of the program to offload, there is little work to address the privacy-leakage issue. To the best of our knowledge, the only privacy-aware offloading work is a data-oriented offloading approach [3] which however has fundamental design issues in protecting privacy effectively.[1]

[1]Being more specific, the data-partitioning approach only considers preserving privacy at certain point during program execution, totally ignoring the continuity of private data flow, which is addressed by our approach.

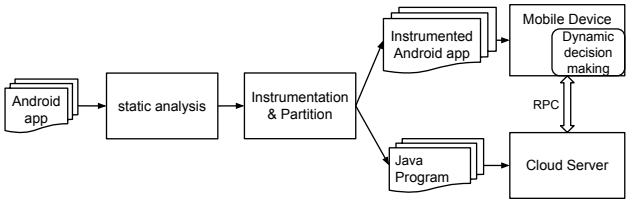

Fig. 1: Overview of the System

We also argue that our problem specific to mobile apps is different from the conventional research on privacy-aware partitioning in the client-server scenario [4] which does not consider limiting the resource consumption as much as in the mobile scenario.

In this work, we address the privacy-preserving offloading of mobile apps to the public cloud. Our proposed approach is to enforce privacy preservation in a fully automatic and end-to-end fashion. Concretely, our proposed approach performs static data-flow analysis to discover all the instructions that operate on private user data. The non-private instructions are then offloaded to the cloud as lightweight RPC methods. Because our offloading analysis occurs at the fine-grained statement level, it could potentially amplify the communication overhead to the cloud. To overcome this inefficiency, we propose a novel technique to group offloadable statements in a way to minimize communication overhead while preserving the original program logic. To improve the runtime efficiency, we instrument the original program to determine in real-time where the offloadable code should run, remotely or locally. Decisions are made dynamically based on device and network conditions.

II. Design & Implementation

In this section, we discuss the design and implementation of our system prototype.

Our design leverages static program analysis to identify the non-offloadable code, utilizes instrumentation techniques to rewrite the app and relies on a decision making component to make offloading decisions dynamically. Figure 1 illustrates the overall design of our privacy-preserving offloading technique.

A. Static Analysis

The static analysis process contains three major steps: non-offloadable code identification, offloadable code grouping and pre-filtering.

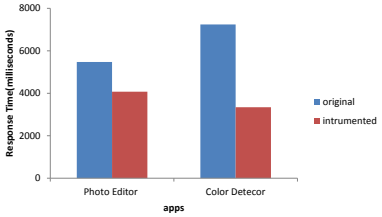

Fig. 2: Runtime Performance

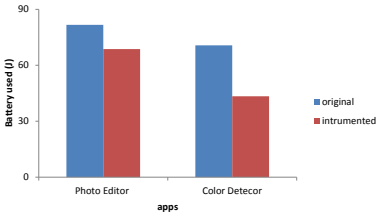

Fig. 3: Power Consumption

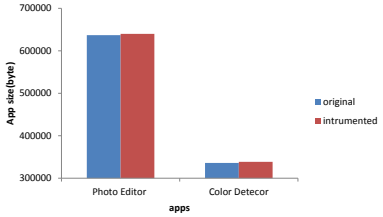

Fig. 4: App Size

Non-Offloadable Code Identification: In order to detect offloadable code regions, we identify and mark the following four types of code statements as non-offloadable: 1) private data manipulation code statements; 2) GUI components that directly interact with users; 3)local resource access code statements and 4) other Android APIs that rely on either Android OS or physical device to execute. We perform context-sensitive, flow-sensitive and interprocedural data-flow analysis to locate them.

Offloadable Code Grouping: In theory, all statements other than the non-offloadable ones are offloadable. However, to maintain the original program logic, we group those offloadable code statements into a number of code regions without breaking the original control-flow. At the same time, we keep the code regions as large as possible to minimize instrumentation and communication overhead.

Pre-filtering: We also perform pre-filtering to figure out which code regions actually contain heavy computation and can potentially bring performance gain if offloaded. This process is necessary to ensure runtime performance and minimize the app size increase introduced by instrumentation.

B. Instrumentation & Offloading

Based on dataflow analysis, we create an offloaded Java class for a target program. We also instrument the original program, so that it can dynamically choose to execute the offloaded code or its corresponding local copy, according to the runtime performance measurement.

Offloaded Java Class: We first make a copy of the offloadable code regions in the app. Then, each copied code region is formed into one RPC (i.e. remote procedural call) method. In the end, all the RPC methods are encapsulated into one dummy class, which is deployed on the cloud side. To minimize data transmission, we perform points-to analysis to discover only the necessary data for execution.

Instrumentation: We then instrument the original program by inserting decision making code and remote procedural calls. For each offloaded RPC method, we locate its counterpart in the local code, introduce a method call to decide whether to run this local copy or the remote one and insert a conditional statement which checks the return value of the decision making method. Depending on this return value, it may jump to one of the two target branches. The first branch is the original local code and the second one is a call to the remote code. To this end, we insert a call statement to invoke the corresponding RPC method.

C. Cloud Side Deployment

We then push the dummy class generated during instrumentation to server-side. This Java class contains all the offloaded code regions as each one is an individual function. We choose not to maintain a cloned Android VM because maintaining such a VM involves heavy synchronization overhead. We use RPC to communicate between mobile device and cloud.

D. Dynamic Decision Making Component

We create an Android service to make runtime decisions. The decision making service runs in the background to gather real-time system information periodically and make decisions based on collected information. In order to invoke this component, we instrument the app at the beginning of each offloadable code region to send an intent to the service and receive the offloading decision at runtime.

III. Preliminary Results

We evaluate our system with two Android applications. The result shows that our system can effectively boost runtime performance and reduce battery consumption while preserving privacy. As shown in Figure 2, response times for the two apps are reduced by 25.5% and 53.8% respectively. As depicted in Figure 3, power consumption for the two apps decreases from 81.7J and 70.7J to 68.7J and 43.4J, resulting in reduce rates of 15.9% and 38.7% respectively. We also evaluate the sizes of our instrumentation code. Figure 4 shows that the sizes of apps before and after instrumentation are almost the same with negligible increases around 0.4% and 0.8%. This means storage on mobile devices will not be affected by our offloading technique.

References

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[3] M. Al-Mutawa and S. Mishra, "Data partitioning: An approach to preserving data privacy in computation offload in pervasive computing systems," in *Proceedings of the 10th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, 2014.

[4] O. Arden, M. George, J. Liu, K. Vikram, A. Askarov, and A. Myers, "Sharing mobile code securely with information flow control," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.