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 paper, 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 evaluate the effectiveness and performance of our system using two Android apps. Preliminary experimental results show that our offloading technique can effectively preserve user privacy while reducing hardware resource consumption at the same time.

1 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 [1–8] 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 [9] which however has fundamental design issues in protecting privacy effectively.[1] 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 [10–12] which does not focus on offloading code to reduce hardware resource consumption.

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 code statements that operate on private user data. The non-private ones 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.

[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.

```
1  public void OnTouchEvent(MotionEvent e) {
2      if(e is the correct touch event){
3        //First load picture to bitmap
4        //then embed location info after editing
5        Bitmap bm = LoadPicture(path);
6        EditPicture(bm);
7        ...
8      }
9  }
10 Bitmap LoadPicture(String path) {
11     ...
12     Bitmap bm = BitmapFactory.decodeFile(file);
13     return bm;
14 }
15 void EditPicture(Bitmap bm) {
16     //heavy computational picture editing
17     bm.getPixels(pix, 0, mPhotoWidth, 0, 0,
        mPhotoWidth, mPhotoHeight);
18     for(every pixel) {
19       ...
20     }
21     //embed location information into picture
22     Location loc = getLastKnownLocation(Provider);
23     double longitude = loc.getLongitude();
24     double latitude = loc.getLatitude();
25     EmbedLocation(bm, longitude, latitude);
26 }
```

Figure 1: Privacy-preserving Offloading Example

Our main contributions in this paper can be summarized as follows:

- We propose a novel technique to maximize the offloading of mobile app code to the public cloud while preserving user privacy-related information.
- We design and implement a prototype system which automatically identifies offloadable code via static data-flow analysis. It utilizes instrumentation techniques to make offloading decisions at runtime.
- We evaluate the effectiveness of the system by using two Android apps. Experimental results show that our offloading system is able to improve runtime performance and reduce battery consumption while preserving user privacy.

2 Overview

2.1 Motivating Example

Figure 1 shows a simplified photo editor for which we perform privacy-preserving offloading. This photo editor app registers a callback function `OnTouchEvent()` which will first call `LoadPicture()` to load a picture into a bitmap from a local file and then invoke `EditPicture()` to edit it. During editing, the app will retrieve and embed the current location into the photo. In order to save battery and gain performance, we offload the heavy computation in `EditPicture()` to the server. However, existing approaches only allow us to enable offloading at the method level. Thus, the location information will also be sent to the cloud server, resulting in privacy leakage (if user considers location as private). If we choose to preserve privacy, we have to keep the whole `EditPicture()` running locally. This app contains both offloadable and non-offloadable code. Code statements in bold text are non-offloadable. For line 12, `decodeFile()` loads a bitmap from a local file. It is non-offloadable because it relies on the local file to execute. Lines 22-25 retrieve location information using `getLastKnownLocation()` and embed the information into the edited photo. Since they manipulate user private data (i.e. location), we choose not to offload them so as to protect user privacy. Other than those non-offloadable code statements, we automatically offload heavy computation (e.g. photo editing part) to the cloud. This requirement motivates us to find a new design to perform offloading during which user privacy preservation is guaranteed.

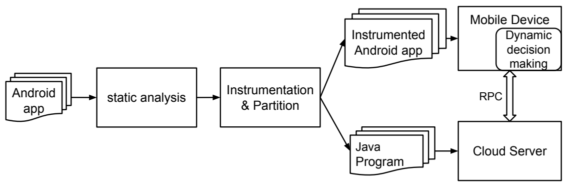

Figure 2: Overview of the System

2.2 Problem Statement

In this work, we address the problem of privacy-preserving mobile app offloading and propose an automatic and fine-grained approach to solve the problem. More specifically, we aim at achieving the following design goals:

- **Privacy Preservation**: Our approach keeps user private data within mobile devices during offloading, thus preventing privacy leakage. This goal requires us to conduct offloading in a fine-grained manner (i.e. at the code statement level).
- **Automatic Offloading**: Our approach performs app offloading in an automatic way. It does not involve any human effort such as user annotation.
- **High Performance**: Our approach aims for high performance. Thus, we have to address the intensive code instrumentation and network communications due to fine-grained app offloading.

2.3 Architecture Overview

To achieve the above goals, 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. To achieve high performance, our proposed technique performs pre-filtering for offloadable code regions and ensures lightweight communication between a device and the cloud by transmitting only necessary data. Figure 2 illustrates the overall workflow of our privacy-preserving offloading technique. It mainly takes the following steps:

(1) **Static Analysis:** The privacy-preserving nature of our system is guaranteed by static analysis. We first perform static program analysis to locate and mark all non-offloadable code. Then we extract code regions that are separated by non-offloadable code and evaluate the potential offloading performance gain in the pre-filtering process. We mark them as offloadable only if the potential performance gain is positive.

(2) **Instrumentation & Partition:** Having marked offloadable code regions , we then perform code instrumentation at the beginning of every offloadable code region. This is to invoke a dynamic decision making component to decide whether we should really offload the code at runtime or not. We also create a wrapper function for the code region. The parameters are the necessary data used within the offloadable code region. We then partition the Android app by extracting offloadable code regions, construct a self-contained offloadable Java program, and push it to the cloud. Finally, we instrument the app so that we can execute the offloadable code locally or remotely, depending on the decision made dynamically (as will be described below).

(3) **Cloud Side Deployment:** Due to the fine-grained nature of our approach, we might introduce more communications between the mobile device and cloud than existing offloading approaches. In order to obtain high performance, we need to keep the communication lightweight. To this end, we do not maintain an Android VM in the cloud. Instead, we directly execute the offloaded code as a Java program on the cloud and only transmit necessary data from the mobile device. Our static analysis enables us to know exactly what data are needed.

(4) **Dynamic Decision Making:** The fact that a code region is marked as offloadable does not necessarily mean it should be offloaded at runtime. For example, if the current network connection is poor, offloading might actually introduce severe delay. The dynamic decision making component collects related information at runtime and makes decisions on whether the code should indeed be offloaded.

3 Design & Implementation

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

3.1 Static Analysis

We first convert the Android Dalvik bytecode into Java bytecode program using dex2jar [13], then perform our static analysis which is built on top of Soot [14]. The static analysis process contains three major steps: non-offloadable code identification, offloadable code grouping and pre-filtering.

Non-Offloadable Code Identification In order to detect offloadable code regions we need to identify the non-offloadable code first. Specifically, we mark the following four types of code statements as non-offloadable: 1) private data manipulation statements which retrieve (e.g. `getLastKnownLocation()`) and manipulate user private data; 2) GUI components that directly interact with users; 3) local resource access statements such as `FileInputStream()` and `mkdir()`, and 4) other Android APIs that rely on either Android OS or physical device to execute (e.g. `SendTextMessage()`). For private data manipulation statements, we perform context-sensitive, flow-sensitive and inter-procedural data-flow analysis proposed in [15] to locate them. Our system allows users to select which data is considered senisitive in a configuration file. Currently, we consider location, system setting and device ID as private by default. For other three types of statements, we manually define an API list and statically search for them.

Offloadable Code Grouping After identifying all the non-offloadable statements, we extract offloadable code regions. In theory, all statements other than the non-offloadable ones are offloadable. However, to maintain the original program logic, we group those offloadable 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. In order to achieve these two goals, the entry point of each code region can be 1) the entry point of an original method, 2) the immediate successor of a non-offloadable region, or 3) the jump target of another offloadable code region. We achieve this by using Algorithm 1.

The inputs of the algorithm are 1) Set_{NO}: a set of all non-offloadable statements in the app identified in the previous step and 2) Set_{method}: a set of all methods within the app. The output is Set_O which is a set of all the offloadable code regions extracted. The whole algorithm contains two loops. The first loop is to analyze the program and collect inter-split branch statements (e.g. goto statements), and the second one is to extract real offloadable code regions. In the first loop, for every method m within the app, the algorithm calls `GetControlFlowGraph()` to retrieve the control-flow graph (CFG) of m, then it invokes `SplitCFG()`.

Algorithm 1 Offloadable Code Grouping

```
$Set_{NO} \leftarrow$ all the non-offloadable statements in app
$Set_{method} \leftarrow$ all the methods in app
$Set_{intersplitBranch}, Set_{intersplitTarget} \leftarrow$ null
$Set_O \leftarrow$ null
for $m \in Set_{method}$ do
    $cfg_m \leftarrow$ GetControlFlowGraph($m$)
    $Set_{cfg'} \leftarrow$ SplitCFG($cfg_m$, $Set_{NO}$)
    for $cfg' \in Set_{cfg'}$ do
        $Set_{branch} \leftarrow$ GetBranches($cfg'$)
        if any $b$ in $Set_{branch}$ has a target $t$ out of $cfg'$ then
            $Set_{intersplitBranch} \leftarrow Set_{intersplitBranch} \cup b$
            $Set_{intersplitTarget} \leftarrow Set_{intersplitTarget} \cup t$
        end if
    end for
end for
for $m \in Set_{method}$ do
    $cfg_m \leftarrow$ GetControlFlowGraph($m$)
    $Set_{cfg''} \leftarrow$ SplitCFG($cfg_m$, $Set_{NO} \cup Set_{intersplitTarget}$)
    $Set_{cfg''} \leftarrow$ DeleteNode($Set_{cfg''}$, $Set_{NO}$)
    $Set_{cfg''} \leftarrow$ SetAsReturn($Set_{cfg''}$, $Set_{intersplitBranch}$)
    $Set_O \leftarrow Set_O \cup Set_{cfg''}$
end for
output $Set_O$ as a set of offloadable code regions
```

`SplitCFG()` will first locate all the statements in Set_{NO} within cfg_m, treat them as new entry points and their predecessors as new exit points, and split cfg_m to get a set of CFG splits $Set_{cfg'}$. Then for each split cfg' in $Set_{cfg'}$, the algorithm calls `GetBranches()` to retrieve Set_{branch} which is a set of branch statements in cfg'. If any branch statement b has a target t that is out of the current cfg', t will become a new entry point to a code region and later may become an entry point of a Java function in the cloud. The algorithm will put b and t into $Set_{intersplitBranch}$ and $Set_{intersplitTarget}$ respectively.

Next, the second loop iterates through each method again to calculate a new set of CFG splits $Set_{cfg''}$ with statements in $Set_{NO} \cup Set_{intersplitTarget}$ as new entry points. For $Set_{cfg''}$, the algorithm first calls `DeleteNode` to delete non-offloadable statements from it and then sets branch statements from $Set_{intersplitBranch}$ to be the new exit points. So by the end of the algorithm, the exit points of code regions can be 1) the exit point of an original method, 2) the immediate predecessor of a non-offloadable region, or 3) the branch statement which has a target in another offloadable code region. The output Set_O contains offload-safe code regions which later may become Java methods on the server side.

Pre-filtering Due to the existence of non-offloadable statements and the requirement of keeping the original control flow, our algorithm may produce a large amount of small code regions that are offloadable. Since each code region is offloaded as a remote Java method, simply offloading all of them will introduce excessive instrumentation and communication overhead. To deal with this issue, we 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 increase of app size introduced by instrumentation.

So far we statically mark one code region as offloadable if it contains loops or an excessive number of statements. Ideally, we could leverage a dynamic profiler, such as CloneCloud [6], to quantitatively measure the performance gain.

3.2 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. For example, if only a subset of an array is used in an offloadable code region, our system will not transmit the entire array.

Instrumentation We then instrument the original program by inserting decision making code and remote procedural calls. Figure 3 depicts the local code after instrumentation. First, for each offloaded RPC method, we locate its counterpart in the local code. Then, prior to the entry point of a counterpart, we introduce a method call to decide whether to run this local copy or the remote one. Next, we 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.

3.3 Cloud Side Deployment

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

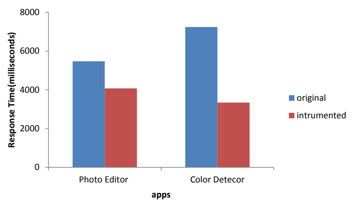

Figure 4: Runtime Performance

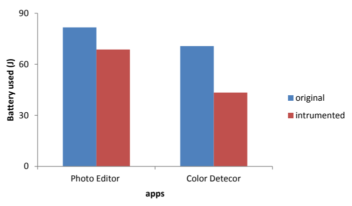

Figure 5: Power Consumption

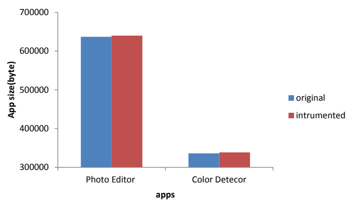

Figure 6: App Size

```
void EditPicture(Bitmap bm) {
    boolean callRemote = makeDecision();
    if(callRemote){
        callRPCEditPicture1(bm);
    }else{
      //heavy computational picture editing
      bm.getPixels(pix, 0, mPhotoWidth, 0, 0,
     mPhotoWidth, mPhotoHeight);
      ...
    }
    //embed location information into picture
    Location loc = getLastKnownLocation(Provider);
    ...
    EmbedLocation(bm, longitude, latitude);
}
```

Figure 3: Instrumented Code

3.4 Dynamic Decision Making Component

We create an Android service to make offloading decisions based on runtime information. To communicate with this component, we instrument an app by inserting code at the beginning of each offloadable code region, so that the inserted code can send the request to the service and receive the decision at runtime.

This service component runs in the background to gather real-time system information (CPU usage, Memory usage and Network status) by parsing `/proc/stat` files periodically. Currently we simply use network connectivity to make offloading decisions.

This Android service will not bring security issues such as permission re-delegation problem [16] since it merely returns offloading decisions and does not disclose any system information. Prior research [17] has shown that the consumption of hardware resources is negligible for such a service.

4 Evaluation

In this section, we present some preliminary evaluation results of our system. We evaluate our system with two Android applications. The first app is the Photo Editor which loads a photo from a local file, performs photo editing and embeds location information into the photo. The second app is a Color Detector that utilizes the camera on the device to capture a picture, detects colors in an area selected by users and shows text in different languages based on current locale setting. In these two apps, location information and locale setting are user private data that we preserve from offloading. We conduct experiments on a Nexus S model device running Android 4.0.4. We assume the network status is good and no other app is running so the dynamic decision making service will always return true.

Runtime Performance & Battery Saving Figure 4 illustrates the runtime performance of the system. We compare the runtime performance of instrumented apps with the original ones. The X-axis represents the apps (original and instrumented versions for each Android app) while the Y-axis is the average response time. As shown from the figure, response times for Photo Editor and Color Detector are reduced by 25.5% and 53.8% respectively. The major reason why the performance gain for Photo Editor is much smaller than that for Color Detector is that the communication between the device and the server is much heavier for the former one. For Photo Editor, we have to transmit the whole picture to the server in order to perform editing while for Color Detector we could transmit only the pixels in the user selected area. We then leverage PowerTutor [18] to evaluate the battery consumption. As depicted in Figure 5, power consumption for the two apps decreases from 81.7J and 70.7J to 68.7J and 43.4J, resulting in the reduction rates of 15.9% and 38.7% respectively.

App Size We also evaluate the sizes of our instrumentation code. As shown in Figure 6, the sizes of apps before and after instrumentation are almost the same with negligible increases around 0.4% and 0.8%. This means the storage on mobile devices will not be affected by our offloading technique.

5 Conclusion

We propose a novel technique to automatically perform fine-grained privacy-preserving Android app offloading. We implement a prototype and evaluate our system by utilizing two Android apps. Experimental results show that our system can effectively boost performance and save battery while preventing privacy data from leaking to the cloud.

6 Discussion

Loop Quantification At present, we do not quantify the computation of loop operations. Instead, we qualitatively consider all loop operations to be equally computation-intensive. However, in practice, different loop operations consume distinctive amounts of resources, depending on how many times the iterations are performed. Sometimes, a loop counter is statically resolvable. Thus, we can perform static dataflow analysis to trace its origin. In other cases, the number of iterations cannot be determined in static analysis because it is originated from a runtime value, for example, a use input. To address this problem, we need to leverage dynamic analysis as well as machine learning technique. This presents a possible direction for our future work.

Privacy Policy Configuration So far, we have a static and general set of privacy policies for all users. Our system considers location, system setting and device ID as sensitive data, and prevents them from being offloaded to the cloud. Nevertheless, an individual user may be more concerned about specific private information under particular circumstances. For example, an end user is more careful about her geolocation data when she is at home but could be less cautious when she is at work. This fact requires our privacy policies to be configurable at runtime. In addition, the relaxation of privacy restrictions may lead to performance benefits because more executions can be potentially offloaded to the cloud. Thus, how to design such a flexible and efficient dynamic policy system is worth investigating.

Communication Versus Computation There is fundamentally a trade-off between communication delay and computation overhead, with respect to code offloading. On the one hand, we delegate computation-intensive workloads to the cloud. On the other hand, we also hope to reduce the network communication between the mobile device and the cloud. To minimize the communication overhead, we look for the maximal connected graphs from control-flow graphs. We further perform a pre-filtering to select a subset from these candidates, so that we only offload the most resource-consuming instructions. However, other factors may also have impacts on the network overhead. For instance, if the offloaded code requires a huge data input from the local device, it can considerably affect the overall performance. Fundamentally, we would like to consider the ratio of computation and communication instead of just the computation. This is an optimization problem with various parameters. It requires future discussion to construct such a target function, including the selection of significant parameters and their weights.

7 Acknowledgment

We would like to thank anonymous reviewers for their comments. This research was supported in part by NSF Grant #1054605. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] A. Gember, C. Dragga, and A. Akella, “Ecos: practical mobile application offloading for enterprises,” in *Proc. Int. Conf. Mobile Systems, Applications And Services*, 2012.

[2] 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.

[3] Y.-W. Kwon and E. Tilevich, “Energy-efficient and fault-tolerant distributed mobile execution,” in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. IEEE, 2012, pp. 586–595.

[4] M. A. Hassan, K. Bhattarai, Q. Wei, and S. Chen, “Pomac: Properly offloading mobile applications to clouds,” in *Proceedings of the 6th USENIX conference on Hot Topics in Cloud Computing*. USENIX Association, 2014.

[5] C.-K. Lin and H. Kung, “Mobile app acceleration via fine-grain offloading to the cloud,” in *Proceedings of the 6th USENIX conference on Hot Topics in Cloud Computing*. USENIX Association, 2014, pp. 8–8.

[6] 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.

[7] E. Chen, S. Ogata, and K. Horikawa, “Offloading android applications to the cloud without customizing android,” in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*. IEEE, 2012.

[8] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, “Comet: Code offload by migrating execution transparently.” in *OSDI*, 2012, pp. 93–106.

[9] 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.

[10] L. Zheng, S. Chong, A. Myers, and S. Zdancewic, "Using replication and partitioning to build secure distributed systems," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, 2003.

[11] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[12] 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.

[13] "dex2jar," http://code.google.com/p/dex2jar/.

[14] "Soot: a Java Optimization Framework," http://www.sable.mcgill.ca/soot/.

[15] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[16] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[17] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets," in *Proceedings of 21th USENIX Network Distributed System Security Symposium*, 2014.

[18] "PowerTutor," http://ziyang.eecs.umich.edu/projects/powertutor/#overview.