CoseDroid: Effective Computation- and Sensing-offloading for Android Apps

Xiangyu Wu[†], Chang Xu[*‡], Ziling Lu[†], Yanyan Jiang[§], Chun Cao[‡], Xiaoxing Ma[‡], Jian Lu[‡]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
[†]{shawnwu20147,nju10lzl}@gmail.com, [‡]{changxu, caochun, xxm, lj}@nju.edu.cn, [§]jiangyy@outlook.com

***Abstract*—Smartphone applications are becoming increasingly popular. However, these applications can suffer limited power budgets or malfunctioned sensing issues from their host devices. Computation offloading addresses this issue by delegating local computation workloads to remote servers. In this paper, we present CoseDroid, a framework that supports dynamic computation- and sensing-offloading across different Android mobile devices. This enables Android applications to virtually "borrow" computation or sensing resources from other devices. We experimentally evaluated CoseDroid with real-world Android applications. The experimental results confirmed CoseDroid's effectiveness in on-demand offloading, as well as supporting sensor variety and spontaneous sensing recovery.**

***Index Terms*—Android app, computation offloading, sensing offloading**

I. Introduction

Mobile devices are becoming increasingly popular [1]. Mobile applications (or *apps* for short), however, may sometimes demand excessive computation resources for complicated tasks and thus threatens the battery life. To address such issues, *offloading* computation tasks to a remote server is proposed [2]–[7]. We in this paper focus on the design and implementation of a closely related, yet different aspect, i.e., combining both computation- and sensing-offloading between different mobile devices. Specifically, we not only allow computation tasks to be executed on another mobile device, but also enable a host mobile device to transparently read sensory data from a server device, as if the host device was equipped with the server device's sensor.

Computation offloading between mobile devices is useful in the following senses. First, such offloading can be conducted on an ad-hoc mobile network, which is useful when there is no stable mobile-sever connection. Moreover, code snippets that cannot successfully run on a server (e.g., due to reading a system-dependent state) oftentimes work well on another mobile device. Offloading code between mobile devices essentially extends what code snippets are considered "offloadable".

Such advantages of computation offloading between mobile devices further motivated us to also implement sensing offloading scheme between mobile devices. As installed sensors for different devices may vary, sensing offloading broadens the capability of a single device (e.g., authentication using another mobile device's fingerprint sensor). Moreover, sensing offloading also resolves the challenge of testing sensor-aware apps in an emulator, as feeding the emulator with mocked or realistic sensory data is a highly non-trivial task even if a sensor simulation tool (e.g., SensorSimulator [8]) is available. To the best of our knowledge, we are the first presenting both computation- and sensing-offloading between mobile devices to support such application scenarios.

In this paper, we present the design and implementation of our <u>co</u>mputation- and <u>se</u>nsing-offloading framework, CoseDroid, for An<u>droid</u> apps. CoseDroid contains two major components: an app refactoring tool and a runtime support platform. The refactoring tool automatically decides whether a method can be offloaded using a static program analysis tool built on Soot [9], as well as instruments the methods to be offloaded. An instrumented method transparently runs on a server mobile device with our runtime support platform. The runtime support platform also serves as the bridge between mobile devices, and manages naming and data exchanges.

We experimentally evaluated our CoseDroid's effectiveness. The experimental results confirmed that CoseDroid can successfully support on-demand computation- and sensing-offloading. For computation offloading, CoseDroid helped improve app performance and reduced power consumption on host devices. For sensing offloading, CoseDroid supported different types of sensors (a.k.a. *sensor variety*) and runtime sensor switching when sensors were malfunctioned (a.k.a. *spontaneous sensing recovery*).

The remainder of this paper is organized as follows. Section II presents the architecture of our CoseDroid framework and introduces its components. Sections III and IV elaborate on the design of CoseDroid and introduce its offloading methodology. Section V experimentally evaluates CoseDroid with real-world Android apps. Section VI discusses related work, and finally Section VII concludes this paper.

II. CoseDroid's Architecture

CoseDroid aims to support computation- and sensing-offloading across Wi-Fi connected Android mobile devices. Compared with the traditional offloading from mobile devices to remote servers, the following two challenges should be addressed:

[*]Chang Xu is the corresponding author.

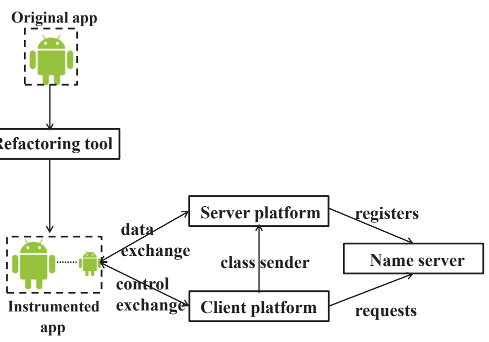


Fig. 1: CoseDroid architecture

- **Challenge 1:** Mobile devices are usually deployed in dynamic and open environments that change over time (e.g., IP address). Therefore, static configurations used by COMET [2] are not feasible for our setting.
- **Challenge 2:** Existing refactoring based approaches require that offloading servers should already have refactored class files prepared (e.g., DPartner [3]). However, for our case *every* mobile device can work as an offloading service provider, and the required class files may not have been provided in advance.

To enable such features, we design CoseDroid's architecture as illustrated in Fig. 1. CoseDroid consists of an app refactoring tool and a runtime support platform. We choose to refactor an app's bytecode to fit devices' execution environments through automated program instrumentation. Compared with COMET [2] that requires its host device burning a customized operating system ROM, our CoseDroid is both light-weight and platform-independent. The instrumented app connects the offloading engine (illustrated by the small Android robot), which is the bridge between the original app and the runtime support platform.

The runtime support platform consists of three parts: a *name server*, a *client platform* and a *server platform*.

To address the first challenge, the name server has fixed name, and maintains IP addresses of offloading service supporters for platforms to use. For the transparency concern, the lookup process should not be done by the app itself. Therefore, the support platform has a client-server structure and the request-reply is done within it, adding no burden to the original app.

To address the second challenge, CoseDroid deploys a class sending module on the client platform as well as a class receiving module at the server platform. The client sends necessary class files to the server on the fly whenever computation offloading is initiated.

III. Design of Computation Offloading

A. The Soot Framework

Our refactoring tool is built upon the program analysis and optimization framework, Soot [9]. Soot is a framework for analyzing and transforming Java and Android applications. It uses Jimple, a typed three-address intermediate representation.

TABLE I: Example class

class C{
    int [][]array;
    void foo(D para1){
        ...
    }
}
C obj;
	class D{
	    int i;
	}

CoseDroid instrumentations are done at Jimple level, and therefore difficulties in handling Dalvik bytecode is avoided.

B. Overview of Computation Offloading

The target of CoseDroid's computation offloading is to provide portable method offloading, allowing a method to be executed correctly on another device. We call the device initiating the offloading request *host device*, and offloaded method's execution site *server device*.

As methods are invoked with contexts (e.g., heap data, system states, etc.), not every method can be safely executed on a server device. Therefore, we use static analysis to identify offloadable methods. Static analysis results are then presented to users to choose which methods are actually offloaded, and these methods are finally instrumented by the refactoring tool.

C. Detecting Offloadable Methods

To ensure correctness of offloaded executions, we require any offloading execution to be *non-interactive* and the offloaded method to be *safe*. We define these two concepts as follows.

An execution is "non-interactive" if there is no information exchange between the host device and the server device during the offloading execution (e.g., reading the host device's system state). A method is "safe" if under the non-interactive execution on the server device, the program state is the same as if the method was executed locally on the host device. Offloading a safe method and conducting non-interactive execution is equivalent to executing the method remotely and sending the new object back.

Let use consider the example shown in Table I. Consider a method invocation `obj.foo(parameters)`. If this method is offloaded, `foo` should be executed on the server device. The corresponding fields of `obj` at the host device should be updated if `array` is modified in the server-side execution. In fact, CoseDroid directly copies the received reference of any field back to the original field like `this.array = newobject.array`. CoseDroid conducts no inconsistency resolution for parameters. So if objects referenced by `parameters` are changed, it would be recommended the method should not be offloaded to avoid inconsistency issues. In other words, our static analysis detects the following issues to ensure successful offloading:

- **Issue 1:** Whether a method is related to I/O or Android system. For example, a method that invokes `android.app.Activity.recreate` in its body must be host-only (otherwise, it may cause unexpected behavior if offloaded).

TABLE II: Jimple three-address representation

Source code	Jimple code

`a[i][j] = 8`	`b = a[i]`
	`b[j] = 8`

`a.i.j = 7`	`b = a.i`
	`b.j = 7`

- **Issue 2:** Whether objects referenced by the method's parameters are changed during an execution.
- **Issue 3:** Whether the fields of their owner class are changed during a method's execution.

Issue 1 is the simplest to address. Given a class, CoseDroid enumerates any of its super classes, interfaces and fields, and transitively determines whether each class is related to I/O or low-level system functionalities (such as UI). If so, any method in this class cannot be offloaded by CoseDroid.

Otherwise, CoseDroid checks whether a method corresponds to I/O or the Android system. We need to check whether it defines or uses I/O or Android instances and whether it invokes I/O or Android functions in a transitive way. As a result, some methods in a class are marked as non-offloadable due to their relations with I/O or the Android system.

To address Issues 2 and 3, suppose that a method has a reference-typed parameter p and CoseDroid aims to check whether objects referenced (directly or indirectly) by p might be changed during an execution (Issue 2). We enumerate all possible access types of arrays and fields of Jimple in Table II as well as our corresponding detection strategy in Table III. Specifically, CoseDroid defines a flow set S with initial value $S = \{p\}$. It needs to store objects that can be referenced by p. As CoseDroid incrementally checks the method body, S is updated accordingly. If CoseDroid detects that an object in S is changed inside, it verdicts that the method may change objects referenced by parameters. It is clear that to change objects referenced by p, the code in fact changes the fields or some index values of objects in S. Fields (Issue 3) are checked in a similar fashion, and the only different is setting the initial value of S to be $\{$`this`$\}$.

Things are a little more complex at method invocation. We need to check the issue of the invoked method as illustrated in Table III. We do not expand the discussions here due to the space limitation. More detailed descriptions are given by our project webpage [10].

D. Code Transformation

To offload a method to another device, the host device sends the (1) class name, (2) method signature, (3) serialized object instance, and (4) serialized method parameters to the server device via *Interface Serializable*. The server device loads the class by Java's reflection API, deserializes the object, and then invokes the requested method on that object. After the execution has completed, the server device will return the method's return value as well as the serialized object to the host. Finally, the host object copies all fields from the received object to resolve any consistency issues.

TABLE III: Detection strategy

Statements	Conditions	Actions	Explanation
`u = a.i`	u is a reference, $a \in S$	S.add(u)	Monitor u later
`u.field = new_value`	$u \in S$	The method changes objects referenced by parameters	The change of u in fact changes objects referenced by p
`u = a[i]`	u is a reference, $a \in S$	S.add(u)	Monitor u later
`u[index] = new_value`	$u \in S$	The method changes objects referenced by parameters	The change of u in fact changes objects referenced by p
`u = a`	$u \in S$, $a \notin S$	S.remove(u)	At first u is referenced by p, but now not
`u.foo()`	$u \in S$	Check whether `foo` changes fields	See reference [10]
`obj.foo(u)`	$u \in S$	Check whether `foo` changes objects referenced by parameters	See reference [10]

TABLE IV: Android's sensor usage

G1: get a SensorManager:	`sm = getSystemService(type)`
G2: get a Sensor:	`sensor = sm.getDefaultSensor(type)`
G3: write a SensorEventListener: sel	
R1:	`sm.registerListener(sel, sensor, delay)`
U1:	`sm.unregisterListener(sel)`
U2:	`sm.unregisterListener(sel, sensor)`

IV. Design of Sensing Offloading

A. Android's Sensor Usage Pattern

Developers interact with Android's sensor subsystem through sensor APIs [11]. Specifically, *SensorManager* is a system service in Android and with it developers can get *Sensor* instances and conduct operations of registering and unregistering sensors. *Sensor* is an instance which stores information of a specific sensor. *SensorEventListener* handles sensor events and digests sensory data via the `onSensorChanged` callback method.

Typical sensor usages are shown in Table IV. A sensor can be registered to a listener as pattern R1 or be unregistered as pattern U1 or pattern U2.

B. Overview of Sensing Offloading

The sensors in one host device may be absent or malfunctioned. As a result, many apps that are dependent on these sensors cannot work. It is desired that the host device can "borrow" sensors from another device. Specifically, "borrow" indicates whenever sensory data are generated at the server device, these data will be sent back to the host device for handling, making the remote sensor act like a local sensor. CoseDroid is capable of shielding apps from low-level details as well as transferring high-level sensory data between devices.

TABLE V: Data structure for sensing offloading

```
Class SensorStateLogger{
    SensorEventListener: listener
    int: already_reg
    int: local_remote
}[ MAX_SENSOR_NUM]
```

C. *Offloading Sensors*

1) The Virtual Sensor Manager: We implemented a *Virtual Sensor Manager* (*VSM*) to replace the Android *Sensor Manager* to enable on-the-fly sensing offloading (specifying the sensory data source to be local sensors or remote sensors).

To realize this feature, CoseDroid instruments sensor registering and unregistering calls to manage (1) registered sensors (through the method `getType` of *Sensor* instance), (2) each sensor's registered listeners (through a parameter of registering calls), (3) and the sensory data (through the fields of *SensorEvent*, which is a parameter of `onSensorChanged`). Such data are maintained by a list of *SensorStateLogger* instances, as illustrated in Table V.

When registering a sensor, fields of the corresponding *SensorStateLogger* instance are updated according to the sensor's type. Also, *VSM* notifies host/server device to register this sensor. When unregistering sensor(s), there are two patterns as listed in Table IV. Since many sensors can be registered to the same listener, the pattern U1 which unregisters a listener's all associated sensors requires the *VSM* to traverse the *SensorStateLogger* list to update each related sensor.

2) Sensory Data Handling: When sensors are offloaded, CoseDroid registers the server device's corresponding `onSensorChanged` events, sends the data back, and at the host side assembles *SensorEvent* instances with the received sensory data. Such callback mocking is also used in previous work [12]. For local sensor usage, CoseDroid invokes the originally registered local handler, and therefore sensory data are handled locally. More detailed implementation issues are discussed at [10].

V. Evaluation

We conducted our evaluations on a typical application scenario that friends are sharing their smartphones' computation power and sensors. An HTC Desire (G7) is used as the host device while a Galaxy S5 is used as the server device. Both devices are connected via a public 100Mbps Wi-Fi network. We divide our experiments into two parts, one for computation offloading and the other for sensing offloading, which are in-depth described in the following subsections.

A. *Computation Offloading*

1) Experiment Design: Computation offloading is designed for time and energy saving for computation-intensive tasks. Therefore it is questionable whether time-consuming computation-intensive methods are actually offloadable. Moreover, we also wonder the merits of computation offloading, specifically, how much time and energy costs can be reduced if computation-intensive methods are offloaded. These intuitions lead to the following research questions.

RQ1: *Among all the methods that are marked as offloadable by CoseDroid, how many of them can be safely offloaded?*
RQ2: *If the methods to be offloaded are well chosen, can the performance improve and/or energy cost decline?*

To conduct the study, we chose two open-source computation-intensive apps, Gobang and Othello as our subjects. Both apps are chessboard games with artificial intelligence search engines. For RQ1, we offload every method that is marked as "offloadable" by CoseDroid, and judge each offloading run's correctness manually. For RQ2, we evaluate the time and energy benefits of computation offloading at different levels of workloads. We instrumented an offloadable method to repeatedly execute for 1, 10, 100, 500 and 1,000 times to simulate higher workloads, and the energy cost is measured by the current readings of a Monsoon PowerMonitor [13].

2) Results: CoseDroid considers a method offloadable if it changes neither fields nor objects referenced by parameters. Such strategy identified 5 offloadable methods for Gobang and 20 for Othello. A careful manual inspection showed that offloaded execution results for any of these methods are correct, and thus RQ1 has gained a positive answer.

To answer RQ2, we offload the method for computing the AI's next move for both apps. Measured energy costs are shown in Fig. 2(a) and 2(b). It is clear that by offloading such computation-intensive methods, CoseDroid successfully reduced the energy cost of the host device.

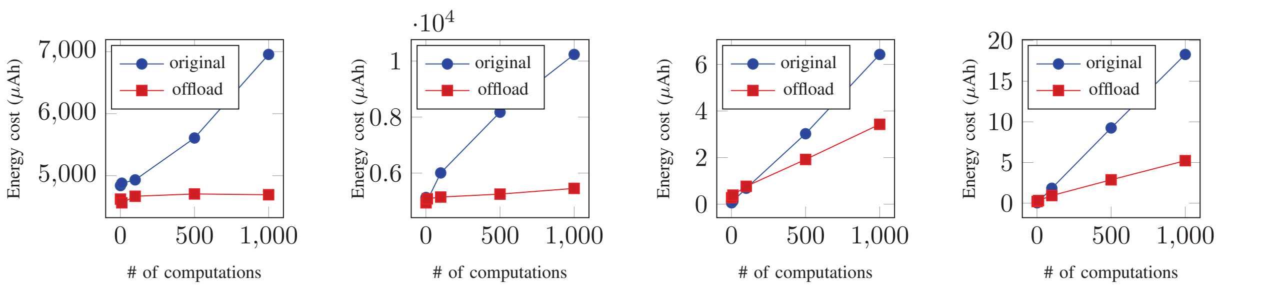
We also measured the average time of executing a offloaded method in Fig. 2(c) and Fig. 2(d). These results indicate that computation offloading indeed incurs a significant amount of overhead (e.g., establishing network connection and transferring data), but as the computation task becomes increasingly time-consuming, offloading a method to a more powerful computation device is clearly beneficial.

B. *Sensing Offloading*

1) Experiment Design: To demonstrate the effectiveness of sensing offloading, we should first answer the question that whether our sensing offloading implementation functionally achieved our goal. If sensing offloading can be correctly performed, as sensing offloading can be potentially energy-saving (by shutting down local sensors) or more energy-consuming (by introducing much network traffic), we also wonder its energy impact. We formulate these research questions as follows.

RQ1: *Is CoseDroid effective in functional needs of sensing offloading?*
RQ2: *Is CoseDroid effective in non-functional needs (energy cost) of sensing offloading?*

To conduct the study, we performed sensor offloading on a series of real-world games and utility tools that rely on sensors, which are shown in TABLE VI. The effectiveness is measured by whether apps are properly functioned when sensors are offloaded. To study the energy impact, we measured the app runs' energy consumption under different configurations: (1)



(a) Gobang energy consumption (b) Othello energy consumption (c) Gobang running time per step (d) Othello running time per step

Fig. 2: Evaluation results

TABLE VI: App selection result

App name	Category	Download
Doodle Jump	Games	XiaZaiBa
Pedometer	Sports	Google Code
My Sensor	System tools	AnZhi Market
CompassNavi	Utility tools	91 Market
Shake tool	Utility tools	Google Code
Smart Sensor	System tools	AnZhi Market
Testproximity	Demos	Manually built

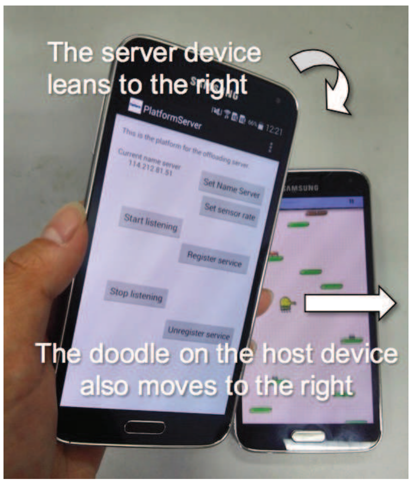


Fig. 3: Sensing Offloading for the Doodle Jump App

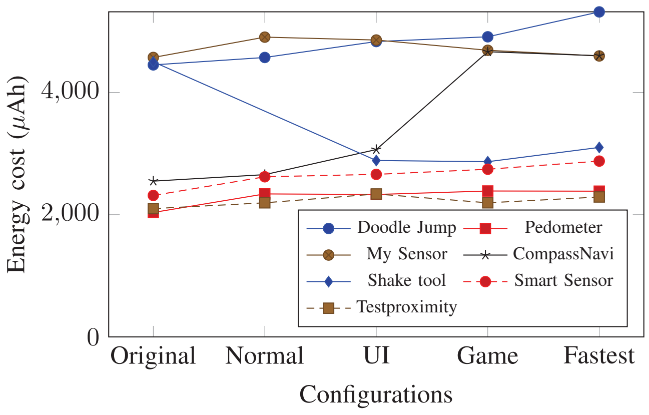


Fig. 4: Energy cost under different rates

whether senors are offloaded; (2) the generation rate of sensory data: “Normal”, “UI”, “Game” and “Fastest”. Each run has a duration of 50 seconds under the same sensor reading pattern, and the current readings are logged for each run.

2) *Results:* To answer RQ1, we refactored all the apps listed above with CoseDroid and observed each app's run. All these apps correctly respond to the server device's sensory data changes. All sensors used in these apps can be successfully offloaded, and CoseDroid succeeds in handling sensor lacking issues and sensing malfunctioned issues. A real-world demonstration of the Doodle Jump game is shown in Fig. 3. The game can be smoothly played using another device's accelerometer.

To answer RQ2, the evaluation results under different configurations are listed in Fig. 4. The value for Shake tool at “Normal” rate is missing because under this update rate, the app is not functioning as the shaking gesture cannot be identified.

Generally, it costs more energy with CoseDroid's sensing offloading functionality. It is not unexpected since sensing offloading is in fact a capability extension, and the extra energy is just the bill users have to pay. The overall trend indicates that one can choose lower update rates to get longer battery life, or use higher update rates to achieve more timely sensory data updates. Nevertheless, there is a chance that sensing offloading can save energy if the sensor itself is very energy-consuming (e.g., in the Shake tool case).

VI. Related Work

The quality of mobile apps hardly matches the rapid growth of the mobile market, and many users are frustrated with various defects of these apps. Therefore, testing approaches [14], [15] are proposed to improve the quality of such apps. On the other hand, to make an app more responsive and energy-saving, offloading computation-intensive workloads to another device is also studied.

Computation offloading can be traced back to early literatures discussing offloading among desktop computers [16], and extended to mobile platforms in recent years [2]–[7]. Compared with related work, both MAUI [4], Cuckoo [5] and [6] require source-level annotations or predefined programming patterns to enable computation offloading, while our CoseDroid uses static analysis to identify offloadable methods automatically. CloneCloud [7] and COMET [2] require modifications to their underlying operating systems to support offloading. Serendipity [17] and COSMOS [18] focus on network and connectivity issues in computation offloading. CoseDroid, like DPartner [3], refactors an Android app's code to realize on-demand computation offloading, which works at an application level and is light-weight. It works between mobile devices as Serendipity [17] does.

As to state-of-the-art sensing offloading, the main research focus is community sensing. Rachuri et al. considered using sensors to trace human behavior [19], which opportunistically offloads sensing to social environments so as to reduce energy cost. PRISM [20] is a platform for remote sensing with smartphones, and community sensing is supported in a sandbox fashion to maximize security. These techniques support rich applications, but require heavy runtime system modifications. CoseDroid, on the other hand, adopts light-weight program instrumentation and is therefore more flexible. There are also a few pieces of work considering motivation mechanism in sensing co-operation [21]. As we focus on the design and implementation of an offloading system, these high-level discussions are out of our scope.

VII. Conclusion and Future Work

In this paper, we present CoseDroid, a computation- and sensing-offloading framework for Android apps. Our evaluation results show that CoseDroid is effective in conducting both computation- and sensing-offloading across Android mobile devices. However, there are still several possible enhancements to our current design and implementation. First, dynamic profiling data can be useful in analyzing offloadable methods, as such data contain more precise information than static analysis. Second, our current implementation only supports limited categories of sensors. How to support complicated sensors (e.g., camera and fingerprint) is still a challenge. Finally, our instrumentation currently cannot be performed on obfuscated bytecode due to limitations of Soot. Offloading obfuscated code is also a future direction.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202) and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491) of China.

References

[1] "Smartphone OS Market Share, Q4 2014," http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently." in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 93–106.

[3] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring Android java code for on-demand computation offloading," in *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, vol. 47, no. 10, 2012, pp. 233–248.

[4] 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 (MobiSys)*, 2010, pp. 49–62.

[5] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Proceedings of the 2nd International Conference on Mobile Computing, Applications, and Services (MobiCASE)*. Springer, 2012, pp. 59–79.

[6] E. Chen, S. Ogata, and K. Horikawa, "Offloading Android applications to the cloud without customizing android," in *Proceedings of the 10th Internation Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2012, pp. 788–793.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, 2011, pp. 301–314.

[8] "SensorSimulator," https://code.google.com/p/openintents/wiki/SensorSimulator.

[9] "Soot," http://sable.github.io/soot/.

[10] "Details," http://moon.nju.edu.cn/spar/people/wxy/CoseDroidapp.html.

[11] "Sensor," http://developer.android.com/guide/topics/sensors/sensors_overview.html.

[12] Y. Liu, C. Xu, S. C. Cheung, and J. Lu, "GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications," in *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 9. IEEE, 2014, pp. 911–940.

[13] "Monsoon PowerMonitor," http://msoon.github.io/powermonitor/.

[14] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Li, "User guided automation for testing mobile apps," in *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, 2014, pp. 27–34.

[15] Y. Liu, C. Xu, and S. C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 1013–1024.

[16] M. Philippsen and M. Zenger, "Javaparty - Transparent remote objects in java," *Concurrency Practice and Experience*, vol. 9, no. 11, pp. 1225–1242, 1997.

[17] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," in *Proceedings of the 13th ACM international symposium on Mobile Ad Hoc Networking and Computing(MobiHoc)*, 2012, pp. 145–154.

[18] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: computation offloading as a service for mobile devices," in *Proceedings of the 15th ACM international symposium on Mobile Ad Hoc Metworking and Computing(MobiHoc)*, 2014, pp. 287–296.

[19] K. K. Rachuri, C. Efstratiou, I. Leontiadis, C. Mascolo, and P. J. Rentfrow, "Metis: Exploring mobile phone sensing offloading for efficiently supporting social sensing applications," in *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2013, pp. 85–93.

[20] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma, "Prism: platform for remote sensing using smartphones," in *Proceedings of the 8th international conference on Mobile systems, applications, and services(MobiSys)*, 2010, pp. 63–76.

[21] D. Yang, G. Xue, X. Fang, and J. Tang, "Crowdsourcing to smartphones: Incentive mechanism design for mobile phone sensing," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2012, pp. 173–184.