Vulnerability Evaluation for Securely Offloading Mobile Apps in the Cloud

He Zhu, Changcheng Huang and James Yan
Department of Systems and Computer Engineering, Carleton University
Ottawa, ON, Canada
{hzhu, huang}@sce.carleton.ca, jim.yan@sympatico.ca

Abstract—**The increasing complexity and explosive growth of smartphone applications, together with the more prevalent use of cloud computing, have inspired strong motivation for offloading computation of mobile apps to the cloud. However, there exist vulnerabilities in apps if they are offloaded in public cloud environment. In this paper, we argue that keeping sensitive information to a local device with other information offloaded would significantly reduce threats from the cloud while enjoy the benefits of cloud computing. Our mechanism divides an app into multiple parts and offloads the less vulnerable parts. To decide which parts should be kept locally to a mobile device, we develop an approach that can decide the impact of a component part (down to an individual object) on the overall vulnerability of an app. We demonstrate how our approach can be implemented with real mobile cloud applications.**

Index Terms— **Mobile Cloud Computing, Cloud Security, Offloading Apps.**

I. Introduction

Cloud computing provides computational resources with high availability and reasonable prices. Meanwhile, state-of-the-art mobile apps, such as face recognition, image and video processing, have increased the consumption of on-demand computational resources. Therefore, offloading apps to have them run on cloud servers has emerged [1] [2] [3] [4] as an attractive solution in order to exploit the capacity and efficiency of cloud computing. Offloading applications into the cloud would also save limited power of mobile devices and accelerate the speed of executing applications.

One critical issue to resolve in offloading apps is how to manage the security risks of cloud computing [5] [6]. Many mobile apps use and store personal information related to banking, health, business, messaging and so on. Sensitive information sent to the remote cloud has a good chance to be exposed to either service providers or malicious customers who have access to the same hardware [6].

Traditional ways to protect remote execution and data storage include mutual authentication, authorization and data encryption for the whole app. If every single part of the app was armored by well-prepared security protections, the system would be safe enough but the cost would also rocket up and not be acceptable for commercial use, because too much information has to be encrypted unnecessarily [7]. The redundant encryption and decryption operations hurt both energy efficiency and user experience. We need to consider the tradeoffs between security and usability, and to maximize the system security subject to a tolerable delay and resource cost.

Recent research focuses on the advantages of offloading mobile applications into remote locations, including both cloud servers and ubiquitous computation facilities nearby. Various systems and architectures have been proposed [1] [2] [3] [4] [8] [9] [10], usually aiming at higher computing power and lower energy consumption for resource-constrained mobile platforms. Resource allocation optimization schemes have been presented [11] [12] with the purpose of minimizing energy consumption. However, there is little discussion about the security of offloading, and mature solutions towards secure offloading cannot be found.

In this paper, we present a mechanism to offload less vulnerable app parts. This approach faces some challenges. First, the complex call relationships of classes and functions within an app make it difficult to highlight the vulnerable parts. Second, it is tricky to determine the level of protection for the parts because of the tradeoffs between security and performance. Besides, no mature systems have been built to break down and offload apps. To address these challenges, we present the following contributions in this paper:

- We propose a novel graph-based analytical model, namely object dependency graph (ODG). Rather than focus on the whole app as in existing work, we divide an app into multiple parts according to the instances of classes created at runtime, and connect them by their dependencies. The vulnerability of each part may propagate along its dependencies and can be evaluated by our model. The ODG structure enables us to identify problematic objects due to dependencies.
- We determine the vulnerability level of all app parts and make offloading decisions jointly so that cloud resources can be best utilized, while the security level is not violated. To our best knowledge, we are the first to take this approach.
- We design an algorithm based on the ODG model that can run on all popular smartphone OS's. Experimental results based on Android OS are illustrated in this paper. The unique point of our solution is the configurable location where the analysis is made. Our analytical algorithms can run either on local devices or in the cloud with the necessary inputs.

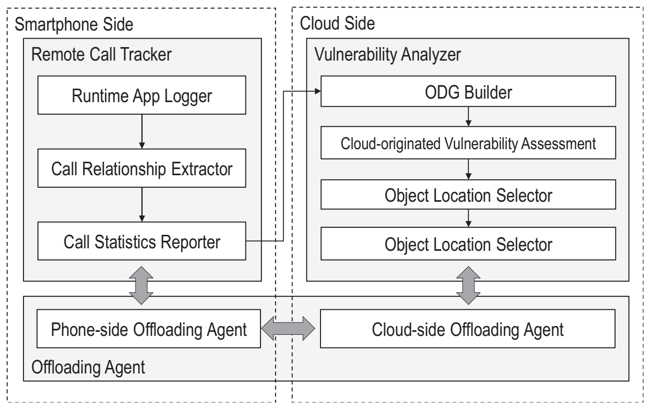
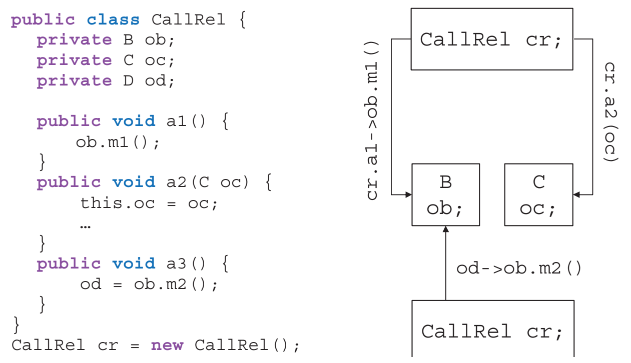


Fig. 1. The system architecture of the vulnerability-based secure offloading framework.



```
public class CallRel {
  private B ob;
  private C oc;
  private D od;

  public void a1() {
      ob.m1();
  }
  public void a2(C oc) {
      this.oc = oc;
      …
  }
  public void a3() {
      od = ob.m2();
  }
}
CallRel cr = new CallRel();
```

Fig. 2. The three methods in this class illustrate three types of call relationships. (1) Method calls: Object `cr` has a call relationship with `ob`, because Method `cr.a1()` invokes `ob.m1()`. (2) Method parameters: Object `cr` has a call relationship with `oc`, because `oc` is a parameter of method `cr.a2()`. (3) Method returns: Object `od` has a call relationship with `ob`, because its value is set by the return of `ob.m2()`.

The remaining sections are organized as follows. Section II describes the system architecture. The Object Dependency Graph (ODG) and the vulnerability model are introduced in the next two sections. Section V shows the evaluation results. The last section concludes the paper.

II. System Architecture

Our system is designed to address the security issues arising when mobile devices use public cloud computing services. These issues are mostly caused by the multi-tenancy feature [13] in cloud computing.

A. Basic Idea

The basic idea of the system is to divide an app into components and to keep the components that have the biggest impact on the vulnerability of the app to a local mobile device. Our idea is novel since existing research work on offloading components of apps mostly focuses on energy efficiency rather than on security. A key factor of our approach is the concept of vulnerabilities of an app and its components.

Defining the vulnerability is a challenging task because it refers to risks that may happen. Suppose a large number of copies of an app have been sold. We consider the ratio of the number of compromised instances of a component in the app to the number of instances of the component in use. In theory, this ratio will converge to a probability when the total number of copies of the app in use goes to infinity. The higher this probability is, the more vulnerable the component will be. We define this probability as the vulnerability of the object.

B. System Modules

To realize this basic idea, the system solution needs three functions. The first one is to partition an app into multiple parts. The second is to quantify how vulnerable each part is to ensure offloading the right ones. Therefore, it is paramount to build an analytical model for calculating vulnerabilities of app parts and their impacts on the overall vulnerability of the app for making our offloading decisions. Finally, the system must be able to actually carry out the decisions by offloading the appropriate objects to cloud servers. Meanwhile, we also need to consider the cost of data communications between local and offloaded objects.

Corresponding to the three functions are the three modules in our system as shown in Fig. 1. The *remote call tracker* is to properly divide the app into multiple parts to be ready for being evaluated and offloaded. The *vulnerability analyzer* is to quantify the vulnerabilities and to make optimal offloading decisions. It takes method call statistics from the smartphones as input, and outputs the offloading decisions. The *offloading agent* is the actual module to fulfill the offloading jobs.

III. Object Vulnerability

In object-oriented programming (OOP), objects are basic units of a running app as they are instantiated from encapsulated class definitions [14]. An object will be identified as a *local object* when it runs on the smartphone. And it will be called a *remote object* if it runs in the cloud.

A. Call Relationship and Message Passing

A call relationship refers to any method invocation or remote message passing that creates dependencies between two objects. The objects depend on each other according to how they are functionally related. On the smartphone side, the method calls defined in classes reveal the dependencies among objects. The forms of method calls are listed below.

- *Method Calls*: a method in an object calls a method in another object. For example, Object `cr` in Fig. 2 has a call relationship with `ob`, because Method `cr.a1()` invokes `ob.m1()`.
- *Method Parameters*: a method in an object has another object's reference as its parameter. Refer to Fig. 2, Object `cr` has a call relationship with `oc`, because `oc` is a parameter of method `cr.a2()`.
- *Method Returns*: the value of an object is set by a method call of another object. In Fig. 2, Object `od` has a call relationship with `ob`, because its value is set by the return of `ob.m2()`.

When the app is partly offloaded and the objects are at different locations, the call relationships would be in form of remote message passing. Depending on the locations and types of objects, there are two types of remote message passing:

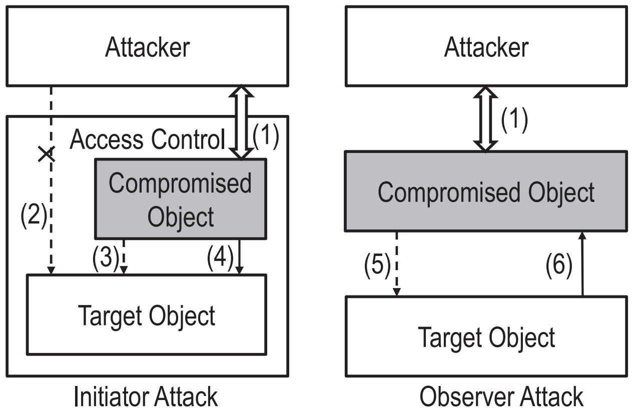


Fig. 3. The processes of initiator attacks and observer attacks. For both attacks, the attacker has the control of a compromised object shown by (1). For initiator attacks, the attacker cannot compromise the target object directly along (2), as there are access control mechanisms to protect illegal accesses. However, attackers may take the compromised object as a proxy, and propagate the vulnerabilities to the target object as illustrated by (3) through the call relationship of (4). The direction of the vulnerability propagation and that of the call relationship are the same. For observer attacks, if the target object calls the compromised object through the call relationship of (6), then the compromised object may send a forged response that could compromise the target object. So the vulnerabilities propagate from the compromised object to the target object, whose direction is opposite to the call relationship.

- *Local-remote Message* is the message passed between a local and a remote object. Smartphones communicate with cloud services through local-remote messages. And the propagation of vulnerabilities by these messages will affect the information on smartphones directly.
- *Remote-remote Message* stands for the message passed between two objects in the cloud. The remote-remote messages do not interact with the smartphone directly, but they may propagate vulnerabilities as well and threat the information security indirectly.

Both method calls and remote message passing enable attacks to gain access to other objects by exchanging data, which lead to the propagation of vulnerabilities, also known as propagated vulnerabilities. We start the analysis from the call relationships, and then define the propagated and the cloud-originated vulnerability.

B. Propagated Vulnerabilities

The propagated vulnerability of an object is defined as the probability that an attack originates from any other object and propagates to the object through call relationships or message passing. The propagated vulnerability could come from either direction of the call relationship by two types of attacks below:

- *Initiator Attacks* enable an attacker to take a compromised object of an app as a proxy. The private resources of the app can be reached through the proxy object on behalf of the attacker. The vulnerability in this case propagates from the object initiating method calls or message passing to the target object being called or passed to. Back to the example in Fig. 2, if `ob` has been attacked, it may behave incorrectly, such as missing necessary functions and performing malicious operations. Those behaviors also affect Function `cr.a1()` and then Object `cr`.
- *Observer Attacks* work in a different way compared to initiator attacks. When a compromised object is called, other objects calling it may pass parameters to it. Those parameters may contain sensitive information. Thus the vulnerabilities can be propagated from the target object being called to the one calling it. In Fig. 2, if `cr` has been compromised, attackers may get sensitive information by accessing `ob`. Similar scenarios happen with message passing too.

The processes of both types of attacks are also described in Fig. 3. The results of vulnerability propagation along opposite directions of call relationships are equal: whichever attacks propagate, the fact that both objects from such call relationship or message passing become compromised remains the same. Thus the ODG can be modeled as an undirected graph.

C. Cloud-originated Vulnerabilities

The cloud-originated vulnerability is caused by interactions between an object and the cloud environment hosting it. It is defined as the probability that an object is compromised due to the weaknesses or the attacks from the cloud environment. Specifically, it may be caused by memory and CPU cache leaks [15], or side/covert channels in the cloud [6]. Because it may be very hard to identify the exact sources that induce the cloud-originated vulnerability, we consider the compromised object as the *originator* if the vulnerability is caused by the cloud environment.

A feasible way to evaluate cloud-originated vulnerabilities is through measurement. To this end, the reputation-based systems can be adopted [13]. Being widely used in mobile Ad-Hoc Networks [16], E-mail Anti-spam [17], online shopping [18] and social networks [19], reputation systems provide us practical solutions to estimate cloud-originated vulnerabilities. It is possible to estimate the cloud-originated vulnerability of a component based on cross-analysis of the vulnerabilities of multiple apps if they share one or more components.

The research on the cloud-originated vulnerabilities will be discussed dedicatedly in the future and will be out of the scope of this paper. In this paper, we will focus on the propagated vulnerability as it is less studied. We will take the cloud-originated vulnerability of each object as a known variable.

D. Object Vulnerabilities

The vulnerability of an object can be defined now as the probability of that object to be compromised due to either cloud-originated or propagated vulnerability. The mathematical descriptions of the definitions above will be modeled in the next section.

IV. Object Dependency Graph

We model all objects in an app as a set of vertices $\mathbb{V}$, and all their call relationships as a set of undirected edges $\mathbb{E}$. They form an undirected graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ for the app, namely Object Dependency Graph (ODG). The ODG of an app reflects the dependencies of objects propagating vulnerabilities. In the following sections, we will use either "objects" or "vertices" for objects in ODGs. We will also use "call relationships" or "edges" for dependencies of objects.

We begin the modeling of the ODG with A_{oo} to denote the event of the originator o being compromised for its cloud-originated vulnerability, and p_{oo} as the probability of A_{oo} to occur, also known as the cloud-originated vulnerability: $p_{oo} = P\{A_{oo}\}$. Let m denote the index of the path the attack takes from o to the target object d, and let r_{od}^{m} denote that path. Let A_{od}^{m} be the event that o has incurred a cloud-originated attack and the attack has propagated to d along r_{od}^{m} such that

$$A_{od}^m = \begin{cases} 1, & if\ an\ attack\ propagates\ on\ the\ path\ r_{od}^m \\ 0, & if\ \ no\ propagation\ found\ on\ the\ path\ r_{od}^m \end{cases} \quad (1)$$

Let p_{od}^m denote the probability of A_{od}^m to occur. For r_{od}^m, A_{od}^m, and p_{od}^m, m can be removed if there is only one path from o to d. Then they are denoted by r_{od}, A_{od}, and p_{od}. Also let e denote an edge in the ODG and p_e be its propagated vulnerability. We suppose the vulnerability propagation is only related to the features of the edge itself. Then the event of the vulnerability propagation along each edge is independent Hence we have the propagated vulnerability along m. The propagated vulnerabilities need to be calculated based on observed vulnerabilities and the topology of the ODG. The calculation method will be shown in the following sections. The results can be reused if duplicated objects are found.

$$p_{od}^m = P\{A_{od}^m\} = P\{A_{od}^m | A_{oo}\}P\{A_{oo}\} = p_{oo} \prod_{e \in r_{od}^m} p_e. \quad (2)$$

Given Equation (2) above, and let E_d denote the event that d is compromised for any reason, so that $P\{E_d\}$ can denote the object vulnerability of d. The propagated vulnerability of d, denoted by μ_d, is the probability that an attack originates from any object $o \in \mathbb{V} - d$ and then propagates to d:

$$\mu_d = P\{E_d | \overline{A_{dd}}\} = P\left\{ \bigcup_{o \in (\mathbb{V} - d)} A_{od} \right\}. \quad (3)$$

The object vulnerability is denoted by π_d, where

$$\pi_d = P\{E_d\} = P\left\{ \bigcup_{o \in V} A_{od} \right\}. \quad (4)$$

A. Vulnerability Propagation Model in ODGs

In this section, an analytical model is to be built to evaluate the vulnerability of an arbitrary vertex in a general ODG. Before we start, without loss of generality, the following assumptions are made for calculation simplicity:

- Each object $o \in \mathbb{V}$ has the same cloud-originated vulnerability $p_{oo} = \alpha$;
- Each relationship $e \in \mathbb{E}$ has the same probability $p_e = \beta$ to propagate an attack.

Refer to the method in [20] that was used in reliability theory, we define the two sets below:

- *Minimal path sets*, denoted by $R_{od} = (r_{od}^1, r_{od}^2, \dots, r_{od}^{M_{od}})$, is the set of all minimal paths. A minimal path is a set of edges that comprise a path, but the removal of any one edge will cause the resulting set not to be a path. In other words, if all the edges in a minimal path are compromised while all other edges are working properly, the target object will be compromised. If any one of the edges in the minimal path subsequently is not compromised, the target object will not be compromised because of this path. Fig. 4 shows an example of the minimal path set. The four minimal paths are illustrated by different line dashes. The minimal path sets can be found via a depth-first search (DFS) that traverses all minimal paths.
- *Minimal cut sets,* denoted by $C_{od} = (C_{od}^1, C_{od}^2, \dots, C_{od}^{D_{od}})$, is the minimal sets of edges whose failure of vulnerability propagation ensure the failure of the attack propagating from o to d. D_{od} is defined as the number of minimal cut sets. Again, Fig. 4 shows an example of the minimal cut set. The minimal cut sets can be found via the CARA algorithm [21] originally used in the fault tree.

According to Equation (1) and (2), the probability of the vulnerability propagation from o to d along a specific path m can be given by the function below:

$$p_{od}^m = P\{A_{oo}\}P\{A_{od}^m = 1\} \quad (5)$$

Since A_{od}^m is a Bernoulli random variable, we may also compute p_{od}^m by taking its expectation. That is,

$$p_{od}^m = P\{A_{oo}\}E[A_{od}^m] \quad (6)$$

Suppose there are M_{od} minimal paths between o and d, then $0 < m \le M_{od}$. Let p_{od} be the probability that o is compromised and the vulnerability is propagated by any minimal path from o to d. Hence

$$p_{od} = P\{A_{oo}\}E\left[\bigcup_{m=1}^{M_{od}} A_{od}^m \right] \quad (7)$$

Now we use an example to show the calculation of propagated vulnerability. An ODG with 4 minimal paths from o to d is shown in Fig. 4. We calculate p_{od} below for this ODG:

$$\begin{aligned} p_{od} = \ & P\{A_{oo}\}\{1 - E[(1 - A_{ox}A_{xd})(1 - A_{oy}A_{yd})(1 \\ & - A_{ox}A_{xy}A_{yd})(1 - A_{oy}A_{xy}A_{xd})]\} \end{aligned} \quad (8)$$

Based on the fact that all events are Bernoulli random variables, we have $A_{yd}^2 = A_{yd}$. Replacing the propagated vulnerability of each edge with α and β, we have

$$\begin{aligned} p_{od} = \ & P\{A_{oo}\}\big(E[A_{ox}A_{xd}A_{oy}A_{yd}A_{xy}] - E[A_{ox}A_{xd}A_{yd}A_{xy}] \\ & + E[A_{ox}A_{xd}A_{oy}A_{yd}] + E[A_{ox}A_{xd}] \\ & - E[A_{ox}A_{oy}A_{yd}A_{xy}] + E[A_{ox}A_{yd}A_{xy}] \\ & + E[A_{oy}A_{yd}]\big) \\ = \ & P\{A_{oo}\}\big(E[A_{ox}]E[A_{xd}]E[A_{oy}]E[A_{yd}]E[A_{xy}] \\ & - E[A_{ox}]E[A_{xd}]E[A_{yd}]E[A_{xy}] \\ & + E[A_{ox}]E[A_{xd}]E[A_{oy}]E[A_{yd}] \\ & + E[A_{ox}]E[A_{xd}] \\ & - E[A_{ox}]E[A_{oy}]E[A_{yd}]E[A_{xy}] \\ & + E[A_{ox}]E[A_{yd}]E[A_{xy}] \\ & + E[A_{oy}]E[A_{yd}]\big) \\ = \ & \alpha(\beta^5 - \beta^4 + \beta^4 + \beta^2 - \beta^4 + \beta^3 + \beta^2) \\ = \ & 2\alpha\beta^2 + \alpha\beta^3 - \alpha\beta^4 + \alpha\beta^5 \end{aligned} \quad (9)$$

As we can see above, the calculation of the propagated vulnerability is tedious especially when more paths exist between two objects. It would be useful if we obtain the bounds of propagated vulnerabilities instead.

Recall the definition of p_e in Equation (2). Let i be the index of the current minimal cut set in C_{od}. Let j be the index of the current minimal path set in R_{od}. Let e be an edge in the current minimal cut set C_{od}^i or minimal path set R_{od}^j.

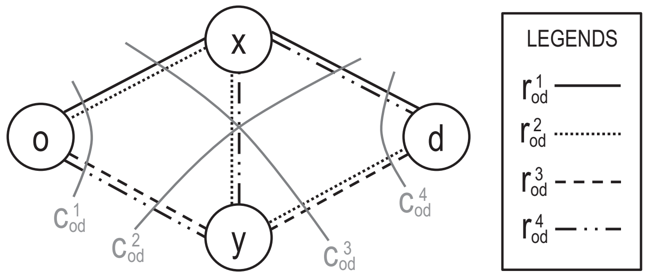


Fig. 4. An ODG with $M_{od} = 4$. As an example of the minimal path sets, it has four minimal paths: $R_{od} = (r_{od}^1, r_{od}^2, r_{od}^3, r_{od}^4)$, distinguished by different line dashes. If consider the graph as an example of the minimal cut sets, there are 4 minimal cuts in grey lines: $C_{od} = (c_{od}^1, c_{od}^2, c_{od}^3, c_{od}^4)$.

With C_{od} we can infer that any successful minimal cut set shall block the propagation of vulnerabilities. So the fact that all minimal cut sets have failed would guarantee a successful attack. Therefore p_{od} should be greater than or equal to the probability that no minimum cut set in C_{od} is satisfied. For any minimal cut set C_{od}^i, the probability of the event that all of the edges in C_{od}^i have failed to propagate the vulnerability is $\prod_{e\epsilon C_{od}^i}(1-p_e)$. Then $\left[1-\prod_{e\epsilon C_{od}^i}(1-p_e)\right]$ is the probability that at least one edge in C_{od}^i can propagate the vulnerability. Define the events $U_{od}^1, U_{od}^2, \dots, U_{od}^{D_{od}}$ by

$$U_{od}^i = \{at\ least\ one\ edge\ in\ C_{od}^i\ can\ propagate\} \tag{10}$$

Hence the probability of U_{od}^i is

$$P\{U_{od}^i\} = 1 - \prod_{e\epsilon C_{od}^i}(1-p_e) \tag{11}$$

Since the vulnerability will propagate if and only if all of the events U_{od}^i occur, we have

$$\begin{aligned} p_{od} &= P\left\{U_{od}^1 U_{od}^2 \dots U_{od}^{D_{od}}\right\} \\ &= P\{U_{od}^1\}P\{U_{od}^2|U_{od}^1\}\dots P\{U_{od}^{D_{od}}|U_{od}^1\dots U_{od}^{D_{od}-1}\} \\ &\geq \prod_{i=1}^{D_{od}} P\{U_{od}^i\} \end{aligned} \tag{12}$$

The last inequality above indicates the lower bound of p_{od}. Replace $P\{U_{od}^i\}$ according to Equation (11), we have

$$p_{od} \geq \prod_{i=1}^{D_{od}}\left[1-\prod_{e\epsilon C_{od}^i}(1-p_e)\right] \tag{13}$$

On the other hand, with R_{od} we can infer that at least one minimal path set in R_{od} should be satisfied to ensure the propagation. For any minimal path set R_{od}^j, the probability of the event that R_{od}^j can propagate the vulnerabilities is $\prod_{e\epsilon R_{od}^j} p_e$. So the probability that at least one edge in R_{od}^j cannot propagate is $\left(1-\prod_{e\epsilon R_{od}^j} p_e\right)$. Define the events $W_{od}^1, W_{od}^2, \dots, W_{od}^{M_{od}}$ by

$$W_{od}^j = \{at\ least\ one\ edge\ in\ R_{od}^j\ cannot\ propagate\} \tag{14}$$

Hence the probability of W_{od}^j is

$$P\{W_{od}^j\} = 1 - \prod_{e\epsilon R_{od}^j}(1-p_e) \tag{15}$$

Since d will not be attacked by o if and only if all of the events W_{od}^j occur, we have

$$\begin{aligned} 1-p_{od} &= P\left\{W_{od}^1 W_{od}^2 \dots W_{od}^{M_{od}}\right\} \\ &= P\{W_{od}^1\}P\{W_{od}^2|W_{od}^1\}\dots P\{W_{od}^{M_{od}}|W_{od}^1\dots W_{od}^{M_{od}-1}\} \\ &\geq \prod_{j=1}^{M_{od}} P\{W_{od}^j\} \end{aligned} \tag{16}$$

Equivalently,

$$p_{od} \leq 1 - \prod_{j=1}^{M_{od}} P\{W_{od}^j\} \tag{17}$$

The last inequality above indicates the upper bound of p_{od}. Replace $P\{W_{od}^j\}$ according to Equation (16), we have

$$p_{od} \leq 1 - \prod_{j=1}^{M_{od}}\left(1-\prod_{e\epsilon R_{od}^j} p_e\right) \tag{18}$$

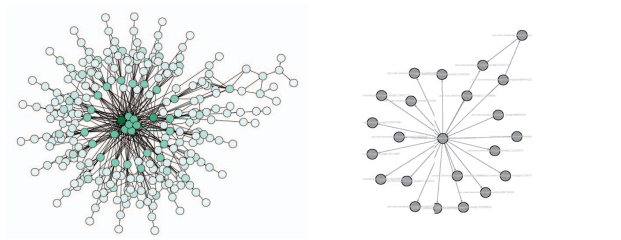
Considering the lower bound and the upper bound, we have the following bounds for the function of propagated vulnerability:

$$\prod_{i=1}^{D_{od}}\left[1-\prod_{e\epsilon C_{od}^i}(1-p_e)\right] \leq p_{od} \leq 1 - \prod_{j=1}^{M_{od}}\left(1-\prod_{e\epsilon R_{od}^j} p_e\right) \tag{19}$$

By listing Algorithm 1, we have implemented the function to calculate the bounds of propagated vulnerabilities. It takes the topology of the ODG as the input. Then it uses standard algorithms to find R_{od} and C_{od}. After that we consider the vulnerabilities of all edges of all paths in R_{od} and C_{od} for the bounds Algorithm 2 outputs. The output value is the lower and upper bounds of the joint vulnerability of a path. We choose the upper bound during the calculation to estimate the path vulnerability aggressively. And then we finally get the propagated vulnerability between two objects.

B. The impact of offloading on object vulnerability

Now that the vulnerability of each object is available, it becomes easier to answer which objects should be kept locally and which can be offloaded. Simply keeping an object with highest vulnerability on the local device does not necessarily make it safer. Staying locally for an object reduces its cloud-originated vulnerability because no eyes would watch it directly from the cloud. But it does not reduce the propagated vulnerability. Meanwhile, separating objects working closely together may cause significant communication cost.



(a) The ODG after the first piece of news has been retrieved, including both original objects and Android APIs. The vertices colored with darker green have larger degrees.

(b) The ODG only displaying *original objects*. It simplifies the scale of the problem greatly but can still identify vulnerable objects.

Fig. 5. An Android application called *TrendCraw* fetching news feeds from Internet. The figures show ODGs of all objects and only original ones, respectively.

We choose to keep objects generating the greatest impact on other nodes. Those originators are kept on mobile devices rather than in the cloud. To calculate the impacts of each object o , we first calculate the difference between the vulnerability of each object with and without o in the ODG, which is called the *absolute impact*. If the propagated vulnerability of the object d is μ_d^o with o and $\mu_d^{\bar{o}}$ without o, the absolute impact from o to d, denoted by δ_d^o, is then

$$\delta_d^o = \mu_d^o - \mu_d^{\bar{o}}. \tag{20}$$

Then we can calculate the ratio o accounts for the vulnerability of d because the difference is solely caused by the removal of o. We name the ratio the *relative impact* from o to d, denoted by γ_d^o, where $\gamma_d^o = \delta_{od}/\pi_d$. Finally, we average the relative impact of each node over all nodes and denote the result by Δ_o, also known as the *impact factor* for the originator. Given that $|\mathbb{V}|$ is the total number of object in the ODG, $\Delta_o$can be calculated with the equation below:

$$\Delta_o = \frac{1}{|\mathbb{V}|}\sum_{d\in\mathbb{V}-o}\gamma_d^o = \frac{1}{|\mathbb{V}|}\sum_{d\in\mathbb{V}-o}\frac{\delta_d^o}{\pi_{od}} = \frac{1}{|\mathbb{V}|}\sum_{d\in\mathbb{V}-o}\frac{\mu_d^o - \mu_d^{\bar{o}}}{\pi_{od}}. \tag{21}$$

Algorithm 1 Propagated Vulnerability Algorithm (PVA)

```
function pva(o, d):
    upperBound = 1
    for each path p1 in $R_{od}$ do
        t1 ← 1
        for each edge e1 in p1 do
            t1 ← t1 × p(e1)
        end for
        upperBound ← upperBound × (1 – t1)
    end for
    upperBound ← 1 - upperBound
    lowerBound ← 1
    for each path p2 in $C_{od}$ do
        t2 ← 1
        for each edge e2 in p2 do
            t2 ← t2 × (1 - p(e2))
        end for
        lowerBound ← lowerBound × (1 – t2)
    end for
    result ← (lowerBound, uppererBound)
    return result
end function
```

Table 1. Parameters of the sample apps

App Name	From Original Packages			Imported Packages
	Packages	Classes	Activities	
TrendCraw	2	9	3	183
MyExpense	3	24	11	194
iMetro	22	145	20	192

The impact factor reflects what the vulnerability of a single object could bring to the ODG. Objects with larger impact factors should be kept running on the local mobile device so as to avoid its cloud-originated vulnerability.

The security impact factor reflects the vulnerability an object can bring to the system. Objects with larger security impact factors should be kept on the local device to avoid its cloud-originated vulnerability.

V. Evaluation and Numerical Results

The ODG model has been evaluated with real apps to verify if vulnerable objects can be correctly identified. We try our best to keep our results general. In our current experiments, we pick three open-source sample applications from different categories from Google Play store and third-party stores. ReThey are listed as follows:

- *TrendCraw* fetches news feeds from the Internet periodically, and then displays the contents to users.
- *MyExpense* manages the daily expense of the user. It does not have Network-related actions.
- *iMetro* provides subway maps and station schedules of cities all over the world. It downloads subway information from the Internet according to users' selection.

Table 1 illustrates parameters of the sample apps. The packages written specifically for the apps (rather than imported) are called *original packages*, while those imported from other libraries are called *imported packages*. From the table we notice the significant difference between the numbers of original packages and those of imported ones. We assume that impacts of imported packages are merged into cloud-originated vulnerabilities so that we can focus on a relatively smaller number of objects in this paper. Fig. 5 compares the differences of the two methods. This simplification is for illustration purpose only and it does not compromise the generality of our approach. We treat the vulnerability analysis of imported packages as separated tasks. After the vulnerabilities of all imported packages are known, we are able to calculate the vulnerability of the package that imports those packages. Moreover, we build a database storing the vulnerabilities of imported packages. The vulnerabilities of the widely-used packages only needs to be computed once. The later requests can directly use the existing results of vulnerabilities of the imported packages.

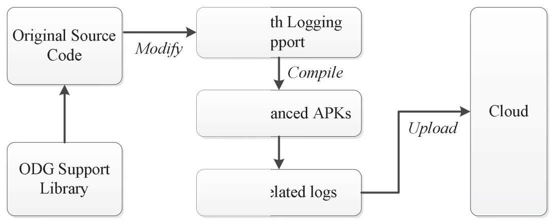
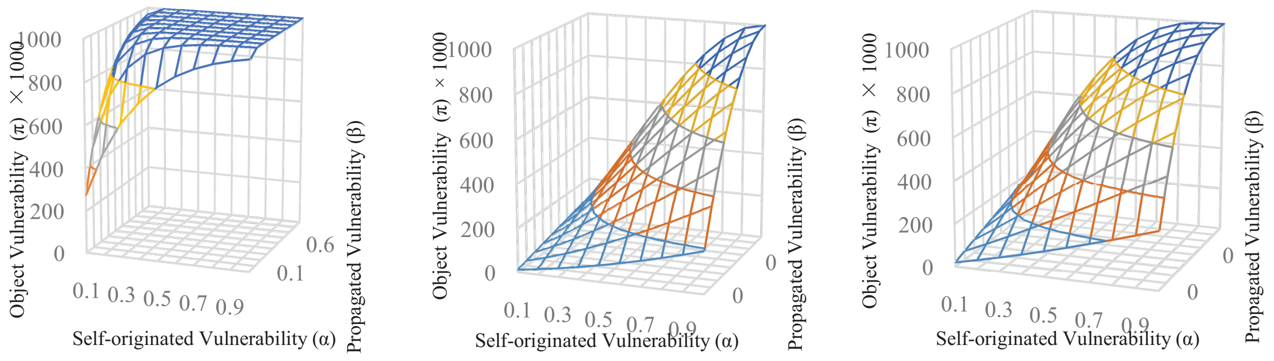


Fig. 6. Standard process of experiments on existing open-source Android apps. With the ODG support library, the apks can be converted to support ODG-related logging. After uploading the logs to the cloud, the analysis will return the offloading decision to the smartphone.



`org.ametro.render.RenderProgram` `org.ametro.render.RenderStation` `org.ametro.render.RenderStationName`

Fig. 7. The vulnerabilities of three objects under different cloud-originated vulnerabilities (α) and propagated vulnerabilities (β). The object with class type `org.ametro.render.RenderProgram` has a faster increasing speed of vulnerability when α and β rise.

A. Standard Process of Experiments

In our experiments, we import both the source code of the apps and our library supporting the ODG model. Minor changes are made to the original source code to enable logging the object calls. Then we recompile all apps to generate the ODG-enhanced Android installation package files in format of .APK, and copy them to smartphones. Our library enables the app to log every object and the call relationships it creates dynamically at runtime, and to save the file in its external storage card. The flow chart of the standard process is shown in Fig. 6.

There are two conditions to make the experiments practical. One is to keep the generality of the object call relationship logging method. The other is to ensure the acceptable complexity of the offloading decision algorithm. Our solution satisfies the conditions as below:

To keep the generality of our process so that fewer changes are required to apply the ODG model to a new app, we introduce the aspect-oriented programming (AOP) paradigm [22] to enable logging right before the occurrence of ODG-related events, such as method calls and object creations. The AOP intercepts the built-in lifecycles of the app rather than its specific working flow. Therefore it is easy to implement automated analysis for apps with the ODG model.

B. Experimental Results

In order to evaluate the offloading results of our ODG model, we implement two other offloading analytical models: CloneCloud (CCD) [3] and ThinkAir (TAR) [4]. The execution time, energy cost and security levels of the three models are compared.

1) Offloading Algorithm Execution Time

In our analysis on the sample apps, it takes more time than the CCD and TAR mechanisms to perform one round of offloading decisions with the strategies above applied, which is shown in Table 2.

Table 2. Time taken for analyzing apps with the three offloading mechanisms

App Name	Offloading Analysis Time (ms)		
	CloneCloud	ThinkAir	ODG
TrendCraw	20937	18561	22428
MyExpense	37115	33753	40756
iMetro	61512	54917	65970

2) Impact of System-wide Vulnerability Change

When all objects are running in the cloud, we wonder which objects are more sensitive to system-wide vulnerability hike, because cloud-originated vulnerabilities change jointly when they run in different clouds. Propagated vulnerabilities also change together depending on the strength of attacks. As a panorama of the response to system-wide vulnerability change, Figure 9 shows the vulnerabilities of objects in the sample app *iMetro* with different α and β. The vulnerabilities are categorized by the types of objects, i.e., the classes of objects..

From the figure we can conclude that the first object `org.ametro.render.RenderProgram` has its vulnerability increased much faster than the other two objects. That indicates its higher sensitivity to the system-wide vulnerability change. Checking the causes of the fact, we have noticed different numbers of neighbors for the listed objects. The objects with larger number of neighbors, including one- and two-hop neighbors, tend to have higher vulnerabilities because they accumulate more propagated vulnerability. Combine Fig. 9 with Fig. 8, we can infer the positive correlation between the sum of one-/two-hop neighbors of one object and its propagated vulnerability.

To demonstrate the impact of object vulnerabilities to offloading decisions in the ODG model, the same numbers of objects are offloaded in each of the three models for comparison. We show results for the energy cost on smartphones and the number of sensitive APIs called in the cloud below.

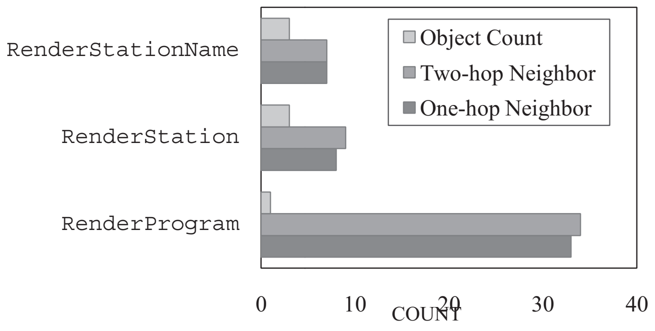


Fig. 8. The number of one- and two-hop neighbors may impact the sensitivity of objects to system-wide vulnerability change. Object `RenderProgram` has about 3 times more neighbors.

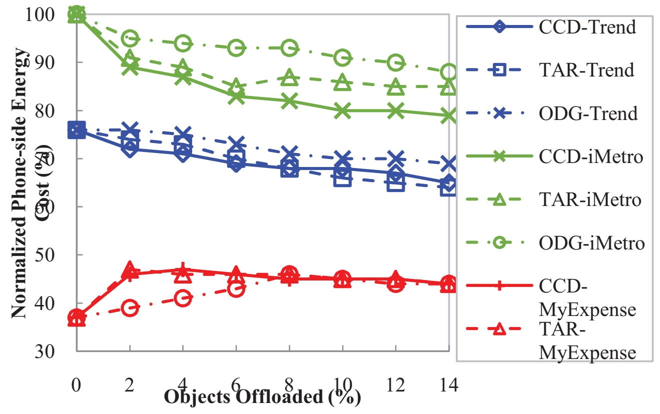


Fig. 9. Normalized phone-side energy cost with the three offloading schemes under different percentages of objects offloaded. The ODG model costs slightly more energy at the smartphone side comparing with the other two mechanisms. But it can still save around 10% of the energy with 14% of the objects offloaded.

3) *Phone-side Energy Cost*

As one of the primary goals of offloading, the energy costs of the three apps are measured. Since the goal of energy saving and secure offloading are conflicting, we choose the security factor as priority in our ODG model: the object with the least security impact Δ shall be offloaded first. When the security impact of the objects is the same, we choose the one that can save most energy for offloading. Meanwhile, the energy factor is taken as the ending condition in our experiments: for all offloading mechanisms, at most 14% of the total objects are offloaded. We choose the model PowerTutor [22] as the measurement tool of energy cost at the smartphone side. After normalizing the energy cost data, the changes of costs are illustrated in Fig. 9. We notice that the ODG model consumes more energy at the smartphone side comparing with the other two mechanisms. But it can still save around 10% of the energy with 14% of the objects offloaded.

4) *Sensitive APIs Accessible by the Cloud*

We choose to trace the running locations of sensitive APIs related to retrieving user's personal information, phone identities and geographical data, because they are related to security issues of mobile apps. We take the approach from [23] to locate the sensitive APIs and analyze the offloading results from the same experiment groups conducted in last section. In our case, an API will be marked as vulnerable either when they run in the cloud or when they can be accessed from the cloud due to object dependencies. From Fig. 10 the ODG model exposes significantly less sensitive APIs to the cloud for all three sample apps.

VI. Future Work

Our future work will first focus on quantifying the cloud-originated vulnerabilities. The calculation of propagated vulnerabilities depends on the cloud-originated vulnerabilities. Thus it directly affects accuracy of the ODG-based system. Like some intrusion detection systems (IDS), we can employ machine learning technology to find features that could impact cloud-originated vulnerabilities.

Besides, we need an optimization model to determine the objects to be offloaded once all vulnerabilities have been estimated. The model outputs the optimal offloading results and is responsible for all consequent communication costs and delay.

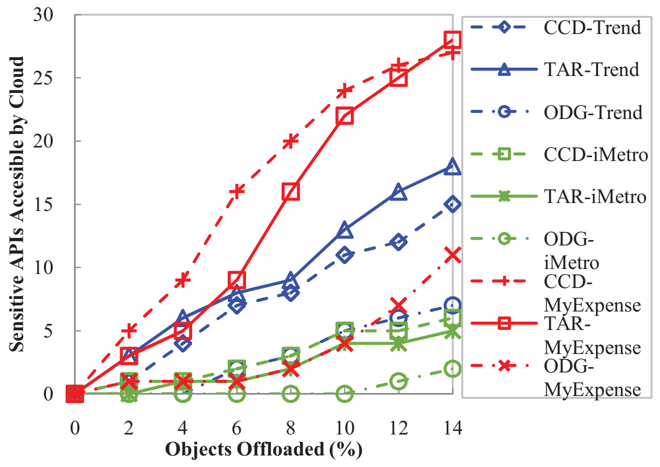


Fig. 10. The numbers of sensitive APIs accessible by the cloud with the three offloading schemes under different percentages of objects offloaded. The ODG model shows significant less sensitive APIs exposed to the cloud.

VII. Conclusion

In this paper, we focus on offloading apps securely without putting vulnerable parts in the cloud. A model named the object dependency graph (ODG) is proposed to analyze the security of mobile apps according to objects they have created at runtime. With the knowledge of cloud-originated and propagated vulnerabilities of runtime objects, the objects' vulnerabilities are calculated taking into account the multipath vulnerability propagation. We have applied our model to real Android apps and have shown that compared to other two existing offloading mechanisms, our offloading approach yields more secure results.

REFERENCES

[1] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE,* vol. 8, pp. 14--23, 2009.

[2] B. G. Chun and P. Maniatis, "Augmented smartphone applications through clone cloud execution," 2009.

[3] B. G. Chun, S. Ihm, P. Maniatis, M. Naik and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," 2011.

[4] S. Kosta, A. Aucinas, P. Hui, R. Mortier and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," 2012.

[5] C. Cachin, I. Keidar and A. Shraer, "Trusting the cloud," *ACM SIGACT News,* vol. 40, pp. 81--86, 2009.

[6] T. Ristenpart, E. Tromer, H. Shacham and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," 2009.

[7] Y. Chen and R. Sion, "On securing untrusted clouds with cryptography," 2010.

[8] X. Zhang, J. Schiffman, S. Gibbs, A. Kunjithapatham and S. Jeong, "Securing elastic applications on mobile devices for cloud computing," 2009.

[9] B.-G. Chun and P. Maniatis, "Dynamically partitioning applications between weak devices and clouds," 2010.

[10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, "MAUI: making smartphones last longer with code offload," 2010.

[11] H. Liang, D. Huang, L. X. Cai, X. Shen and D. Peng, "Resource allocation for security services in mobile cloud computing," 2011.

[12] Y. Wen, W. Zhang and H. Luo, "Energy-optimal mobile application execution: Taming resource-poor mobile devices with cloud clones," 2012.

[13] P. Resnick, K. Kuwabara, R. Zeckhauser and E. Friedman, "Reputation systems," *Communications of the ACM,* vol. 43, pp. 45--48, 2000.

[14] R. L. Biddle, E. Tempero and N. Wellington, Understanding OOP language support for reusability, Department of Computer Science, Victoria University of Wellington, 1995.

[15] Y. Khmelevsky and V. Voytenko, "Cloud computing infrastructure prototype for university education and research," 2010.

[16] S. Buchegger and J.-Y. Le Boudec, "A Robust Reputation System for P2P and M bile Ad hc Netw rks P2P and Mobile Ad-hoc Networks," 2004.

[17] J. Golbeck and J. Hendler, "Reputation network analysis for email filtering," 2004.

[18] P. Resnick and R. Zeckhauser, "Trust among strangers in Internet transactions: Empirical analysis of eBays reputation system," 2002.

[19] J. D. Work, A. Blue and R. Hoffman, *Method and system for reputation evaluation of online users in a social networking scheme,* Google Patents,

[20] S. M. Ross, Introduction to probability models, Academic press, 2009.

[21] L. Rosenberg, "Algorithm for finding minimal cut sets in a fault tree," *Reliability Engineering & System Safety,* vol. 53, pp. 67--71, 1996.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, "Aspect-oriented programming," *ECOOP,* pp. 220--242, 1997.

[23] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," 2010.

[24] W. Enck, D. Octeau, P. McDaniel and S. Chaudhuri, "A study of android application security," 2011.