# Harvesting Developer Credentials in Android Apps

Yajin Zhou[†], Lei Wu[†], Zhi Wang[‡], Xuxian Jiang[⋆]
[†]North Carolina State University [‡]Florida State University [⋆]Qihoo 360
{yajin_zhou, lwu4}@ncsu.edu, zwang@cs.fsu.edu, jiangxuxian@360.cn

## ABSTRACT

Developers often integrate third-party services into their apps. To access a service, an app must authenticate itself to the service with a credential. However, credentials in apps are often not properly or adequately protected, and might be easily extracted by attackers. A leaked credential could pose serious privacy and security threats to both the app developer and app users.

In this paper, we propose *CredMiner* to systematically study the prevalence of unsafe developer credential uses in Android apps. CredMiner can programmatically identify and recover (obfuscated) developer credentials unsafely embedded in Android apps. Specifically, it leverages data flow analysis to identify the raw form of the embedded credential, and selectively executes the part of the program that builds the credential to recover it. We applied CredMiner to 36,561 apps collected from various Android markets to study the use of free email services and Amazon AWS. There were 237 and 196 apps that used these two services, respectively. CredMiner discovered that 51.5% (121/237) and 67.3% (132/196) of them were vulnerable. In total, CredMiner recovered 302 unique email login credentials and 58 unique Amazon AWS credentials, and verified that 252 and 28 of these credentials were still valid at the time of the experiments, respectively.

## Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

## Keywords

CredMiner; Information Flow; Static Analysis; Amazon AWS

## 1. INTRODUCTION

In recent years, mobile apps have become hugely popular with millions of apps in every major app store and tens of billions of accumulated downloads. However, building an app is still a complicated task that demands an intuitive UI design and a reliable and scalable back-end infrastructure. To that end, developers often integrate third-party services into their apps. For example, Amazon S3 (Simple Storage Service) is a popular cloud storage solution for developers to store user data in the cloud without maintaining their own infrastructures. In addition, free email services are frequently used to send crash reports or customer feedback to app developers. To facilitate the integration, these services conveniently provide developers with free SDKs [5] and ready-to-use libraries [10], as well as recipe-style sample code for them to follow [4].

```
.method public static SendMailInBackground
 new-instance v3, Lcom/pompeiicity/funpic/Email;
 const-string v7, "fitt*****@126.com"
 const-string v8, "jed****"
 invoke-direct {v3,v7,v8},Lcom/pompeiicity/funpic/Email;->
             <init>(Ljava/lang/String;Ljava/lang/String;)V
 ...
.end method
```

Figure 1: An Example of Embedded Plaintext Credential

To use a service, the developer often needs to sign up and obtain a developer credential to authenticate to the service. The credential uniquely identifies the developer (or the app) for accounting and access-control purposes. Cryptographic keys and username/password pairs are the two most common forms of developer credentials. To authenticate an app, a simple but utterly unsafe solution is to directly embed the plaintext credential in the app. Figure 1 shows a real world example of this neglectful practice we discovered during our experiments. More cautious developers apply simple transformation or obfuscation to hide their credentials in the apps. Unfortunately, such transformation or obfuscation can be readily reversed, manually or even programmatically. For example, some developers encrypt their credentials with a symmetric-key cryptographic algorithm (e.g., AES), but leave the plaintext keys in their apps.

A leaked credential can compromise all the data protected by it, posing serious threats to the security and privacy of both the developer and the users. For example, a leaked Gmail credential can be used to log into many Google services, such as Gmail, Google Driver, Google Docs, Calendar, Google Analytics, etc. It can also be used to send spam and phishing emails spoofed from the developer. Moreover, security-insensitive developers often reuse their usernames and passwords across different services, giving the attacker an opportunity to compromise those services as well. Similarly, leaked cloud storage credentials allow the attacker to access sensitive user data, in certain cases, the data for *all* the users. The consequences are further exacerbated by the often large installation bases of vulnerable apps.

In this paper, we propose a system called `CredMiner` to systematically study the prevalence of unsafe developer credential uses. CredMiner can identify the insecure handling of developer credentials and automatically recover the embedded ones. It first pre-

*WiSec'15,* June 24–26, 2015, New York, NY, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3623-9/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2766498.2766499.

filters the apps to exclude those that do not contain interesting libraries (we consider a library to be interesting if it contains functions that accept plaintext credentials. Likewise, we call these functions the interesting functions.) For each selected app, CredMiner locates the calls to the interesting functions, and searches backwards for the sources of the credential using the static, backward program slicing [28, 38]. The resulting program slice contains all the Dalvik instructions (i.e., the bytecode of an Android app) necessary to reconstruct the credential from its sources. CredMiner then uses an execution engine to run the program slice *forwardly* to automatically recover the credential.

CredMiner's backward slicing algorithm works on the disassembled Android bytecode. More specifically, we first manually identify every method in the interesting libraries that takes a credential in the parameter. We call these methods the `sink methods`. For example, to send an email with the JavaMail library, the developer needs to instantiate, with the username and password, the `javax.mail.PasswordAuthentication` class. Accordingly, the constructor of this class is a sink method. The parameters for this method thus contain the plaintext credential at *run-time*. From each call site to a sink method, CredMiner recursively backtracks the instructions that touch these parameters. It starts with the registers for the interesting parameters [1], and reversely follows the def-use chains to add new registers and instructions to the tracking set. This process stops when it reaches the instructions that supply the raw form of the credential. We call these instructions the source instructions, and the enclosing method the source method. For example, the `constant-string` instructions at line 3 and 4 in Figure 1 are the source instructions, and `sendMailInBackground` is the source method. After locating all the source and sink methods, CredMiner generates a program slice from the source methods to the sink methods. The slice consists of all the necessary Dalvik instructions (in the Smali format [13]) to reconstruct the credential. CredMiner has an execution engine that can interpret Smali instructions and emulate selected API functions of Android system libraries (e.g., `java.lang.String.append(char)`). To automatically reveal the credential, we execute the generated program slice forwardly with the execution engine. The engine returns the final inputs to the sink methods, which contain the plaintext developer credential. For example, it returns the strings in Line 3 and 4 of Figure 1 as the credential because the intermediate instructions in the slice do not change their values. Finally, we validate whether the recovered credential is still valid or not.

We have implemented a prototype of CredMiner and used it to study the usage of developer credentials in 36,561 apps collected from the official and alternative Android app stores. We focused our study on two types of credentials, email accounts and Amazon AWS/S3 credentials. CredMiner could be easily extended to support other types of credentials. After pre-filtering, the sets of candidate apps consisted of 237 and 196 apps for these two types of credentials, respectively. CredMiner further discovered that 51.5% (121/237) and 67.3% (132/196) of these candidate apps were vulnerable. Some of the vulnerable apps are hugely popular with more than 10,000,000 downloads from the official Google Play store. Additionally, CredMiner extracted 302 unique email login credentials (an app may use several email accounts) and 58 unique Amazon AWS credentials in total. Our validation showed that 252 and 28 of these credentials were still valid at the time of the experiments, respectively.

Our further analysis of the vulnerable apps revealed the worrisome practice of inadequate developer credential protection in Android apps. *First*, nearly 79.5% of the vulnerable apps that use Amazon AWS do not employ *any* protection of their credentials. The credentials are simply embedded in the apps in the plaintext. We suspect this disconcerting failure is the result of developers following the sample code in the (old) AWS SDK, which uses the same insecure practice. We even found several vulnerable apps that seem to have copy-and-pasted the insecure sample code directly. Therefore, it is important to demonstrate the best practice in the sample code, instead of the quick-and-dirty ways to use the API. *Second*, some developers protect their credentials with simple transformation (e.g., string encoding) or weak obfuscation that can be easily reverse-engineered. *Third*, even though some services provide solutions to secure the credentials, these solutions are often not used or are used improperly. For example, Amazon supports a mechanism called token vending machine (TVM) to securely authenticate apps [3]. Unfortunately, the temporary user credentials obtained through TVM are often improperly configured and have much higher privileges than necessary.

In summary, this paper makes the following contributions:

- We propose CredMiner, a system that can identify vulnerable Android apps that have inadequate developer credential protection, and automatically recover these (obfuscated) credentials.
- We have implemented a prototype of CredMiner and applied it to 36,561 apps downloaded from the official and third-party Android app stores. Our experiments showed that CredMiner is effective in harvesting developer credentials in Android apps.
- We systematically study the prevalence of unsafe developer credential uses, and analyze the potential causes of these insecure practices.

The rest of this paper is structured as the following: we first introduce the necessary background information and the threat model in Section 2, and present the design and evaluation of CredMiner in Section 3 and 4, respectively. We then discuss potential improvements to CredMiner in Section 5, followed by the related work in Section 6. Finally, we summarize the paper in Section 7.

## 2. BACKGROUND AND THREAT MODEL

In this section, we briefly introduce the key concepts of Amazon AWS and the JavaMail library to provide necessary background information for CredMiner, and then present the threat model.

### 2.1 Amazon AWS

Amazon AWS offers several services to developers, such as storage, database, and analytics. We use the storage service, Amazon S3 (Simple Storage Service), to illustrate the key concepts of the Amazon AWS credential and user authentication. Amazon S3 is a cloud storage infrastructure to store and retrieve flexible amount of data. Developers interact with Amazon S3 through well-defined APIs and do not need to worry about details of the data storage. Data in Amazon S3 is organized into buckets (similar to disk partitions). The developer can further create folders and files in a bucket. To access the data stored in Amazon S3, an app needs to authenticate itself with an AWS account.

Each customer of Amazon AWS (i.e., the app developer in this case) has a root account, a privileged account that can access all the subscribed AWS services. The root account can create constrained subordinate accounts, called IAM user credentials, through the IAM (Identity and Access Management) service. The developer

[1] Unlike the stack-based standard Java bytecode, the Dalvik bytecode is register-based.

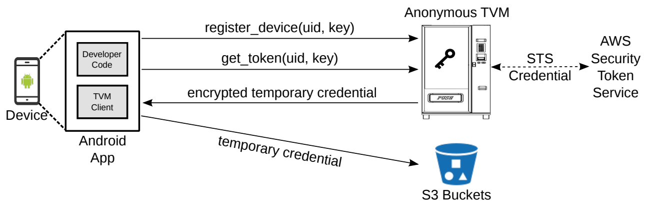


Figure 2: Anonymous TVM Architecture and Interactions

can constrain the access rights for IAM credentials to the minimal permissions necessary for the app. Nevertheless, it is unsafe and inflexible to embed the root account or IAM user credentials in an app: *first*, they can be easily extracted from the app as demonstrated by CredMiner. The leaked credential immediately gives the attacker all the permissions granted to the account. This includes the right to access the app users' stored data and potentially more. *Second*, all the app users share the same credential without proper isolation between them. This allows the users to access each other's data. *Third*, it is difficult to revoke an embedded root or IAM credential should it be compromised, the developer has to release a new version of the app and wait for all the users to upgrade. These issues are applicable to other similar services as well.

To address those issues, Amazon provides AWS Security Token Service that can create temporary security credentials derived from an AWS account (for brevity, we call the parent account the STS credential.) A temporary credential consists of an access key ID, a secret access key, a session token, and an expiration time. The maximum lifetime of a temporary credential is currently 36 hours. By default, a temporary credential inherits the permissions of the parent STS credential. It is thus critical to confine the permissions of the STS credential. The best practice is to create a constrained IAM credential as the STS credential, and never use the root credential for it. AWS Security Token Service can be used in many different ways. Amazon proposes the token vending machine (TVM) architecture to address the aforementioned issues.

In essence, TVM introduces another layer of indirection so that the long-term developer credentials (i.e., the root and IAM credentials) are not exposed to the app. As shown in Figure 2, *the developer* runs a TVM server in the cloud. Each app first registers its device to the TVM server. A device is uniquely identified by a UID and a key. To obtain a temporary credential, the app sends a `get_token` command to the TVM server together with its UID and key. The TVM server then requests the AWS Security Token Service to create a temporary credential and returns it to the app. The temporary credential is tied to the TVM's STS credential. This temporary credential can then be used to access the Amazon S3 service, such as to read files in a bucket. A temporary credential has limited lifetime. The app has to request a new one if it expires. AWS supports both anonymous TVM and identity TVM. The major difference is that identity TVM requires the extra steps for the app user to register and login to the TVM service. Identity TVM allows customizing permissions for individual users and keeping track of the app usage (for simplicity, we mainly use anonymous TVM as the example in this paper.) TVM can address all the aforementioned issues: *first*, the STS credential and other developer credentials are never exposed to the app. It is only used by the TVM server to communicate with Amazon AWS. Moreover, a temporary credential has limited lifetime. Even if leaked, it is much less risky than a compromised developer credential. *Second*, the TVM server can attach policy objects to a temporary credential to further control its permissions. For example, TVM can confine each app instance to its own subdirectory in a bucket, preventing it from accessing other apps' data. *Third*, a temporary credential expires in a few hours.

```
public class Mail extends javax.mail.Authenticator {
  public Mail(String user, String password) {
    this();
    _user = user;
    _password = password;
  }

  public PasswordAuthentication getPasswordAuthentication(){
    return new PasswordAuthentication(_user, _password);
  }
}
```

Figure 3: Suggested Usage of JavaMail in Android Apps

There is no need to release a new version of the app and wait for users to upgrade in order to replace the compromised credential. In addition, each app user uses a unique temporary credential. It is much easier to revoke individual credentials.

TVM is a major improvement over embedding plaintext or obfuscated developer credentials in the app. It represents one viable solution to securely handle developer credentials. However, it is more complicated than those simple-but-insecure practices. Our experiments with a large number of apps show that developers often make mistakes in the deployment of TVMs. A common pitfall is that the STS credential has more privileges than necessary. These excessive permissions are inherited by temporary credentials. For example, CredMiner discovered several cases where a temporary credential can be misused to access *other users' data*. A related issue is that developers often fail to attach (correct) policy objects to temporary credentials in order to strictly separate the data from different users.

## 2.2 JavaMail Library

JavaMail is a popular framework to build email clients in Java. It has been ported to the Android platform [10]. App developers often use this library to send emails in the background. To authenticate to a mail service, the app extends the `javax.mail.Authenticator` class and overrides its method `getPasswordAuthentication`, as shown in Figure 3. This method returns an object of `javax.mail.PasswordAuthentication` initialized by the account name and the password . After authentication, the app can send emails using the SMTP protocol. This is the suggested way to use JavaMail in an Android app [12].

## 2.3 Threat Model

We assume an adversary model where an attacker can download a large number of Android apps. There exist multiple open-source crawlers that can automatically download apps from both Google Play and alternative Android markets [2, 36]. After downloading these apps, the attacker aims at recovering developer credentials embedded in these apps for further attacks (e.g., to collect user data stored in the cloud). In particular, the attacker can use an Android emulator (Google provides one based on QEMU, the ubiquitous emulator) to monitor every executed instruction and memory access by the app to recover the developer credentials. In this paper, we focus on studying the prevalence of unsafe developer credential handling, and consider concrete follow-up attacks out of the scope of this work.

## 3. SYSTEM DESIGN

In this section, we describe the design of CredMiner, specifically, how to select candidate apps, how to identify the source and sink methods of the credentials, and how to recover and validate the embedded credentials.

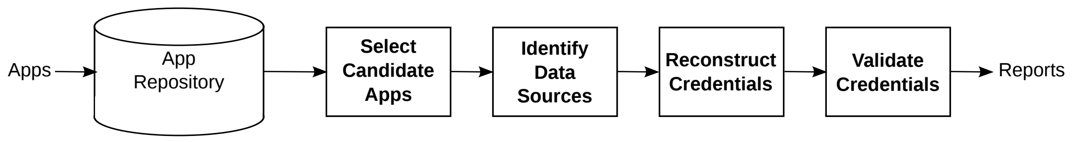


Figure 4: CredMiner Work Flow

## 3.1 Overview

CredMiner aims at recovering developer credentials from Android apps. Figure 4 shows the overall work flow of CredMiner. Given a number of apps in the repository, CredMiner first selects the candidate apps that use one of the interesting libraries (i.e., libraries that use plaintext credentials). For each candidate app, CredMiner performs a data flow analysis to identify the sinks and the sources of the credentials. A program slice is generated connecting the sources to the sinks. CredMiner then tries to reconstruct those credentials by selectively executing the program slice. At last, we manually verify whether the credentials are still valid. CredMiner works on the disassembled Android apps in the Smali format [13]. As such, it does not require the source code of the app, and is compatible with apps downloaded from Google Play and alternative Android app stores. In the rest of this section, we illustrate each step in detail.

## 3.2 Select Candidate Apps

We pre-filter the app repository to exclude apps that do not contain interesting libraries. It seems that this can be achieved by simply extracting the package and class names from the app and compare them to those in an interesting library. However, this simple approach leads to false negatives because of code obfuscation. For example, ProGuard [11] is recommended by Amazon to obfuscate the AWS SDK [15]. Even though an obfuscator cannot change names in the system libraries, it can safely mingle the package/class names of third-party libraries. One possible solution is to use bytecode similarity to decide whether a package of the app matches one of the interesting libraries [26, 31]. However, this process is time consuming since it involves pair-wise comparison between the interesting libraries and all the apps in the repository. Instead, CredMiner employs the following heuristics: popular obfuscators like ProGuard do not change constant strings. These strings give us a strong hint of the original package. However, this heuristic will miss apps obfuscated by tools capable of encoding strings (e.g., DexGuard [8]). We use the package's structural layout to handle this case. Specifically, the number of classes inside a package and their hierarchy are quickly compared to each interesting library for a potential match. We then leverage the slow-but-precise bytecode similarity to determine whether the match is real or not. The string comparison is quite effective. We seldom need to use the more complicated similarity comparison. For each identified (obfuscated) library, we further map its classes and methods to the original ones through package similarity. Note that we only need to compare the similarity for the identified apps. Scalability is not an issue here since the number of such apps is far less than the total number of apps. By de-obfuscating the app, we can locate call sites of the sink methods in the app (Section 3.3). CredMiner takes the following two steps to map obfuscated classes and methods, respectively.

**Mapping Obfuscated Classes:** this step de-obfuscates the classes of a package in the app by locating their counterparts in the original library. We use bytecode similarity for this purpose [26, 31, 44]. Specifically, we construct a string representation for each class (we call it the class signature) and compare it to those of the interesting libraries. A quality class signature should retain the semantics of the class and resist to code obfuscation. In CredMiner, the class signature consists of the number of class fields and the method signatures. A method signature in turn consists of the number and types of parameters, the return type, and its instruction stream. To tolerate differences in the compiler's register allocation and obfuscated class names, we exclude register names and the names of non-system classes in the signature. However, system class names are included because they contain rich semantics and obfuscators often keep them intact. For instance, instruction `invoke-direct{v10, v12},Ljavax/mail/internet/InternetAddress;-><init>(Ljava/lang/String;)V` will be reduced to `invoke-direct{2},(Ljava/lang/String;)V`. Non-system class name `javax.mail.internet.InternetAddress` is excluded from the string representation (with obfuscation, the name will be converted to a non-descriptive one like `La/b/c/d`), while system class name `java/lang/String` is retained. After generating class signatures, we use n-gram to calculate the similarity between classes in the package and the identified library. Matched class pairs have the highest similarity. Since this process has been discussed thoroughly in the previous work [26, 31, 44], we do not elaborate it further.

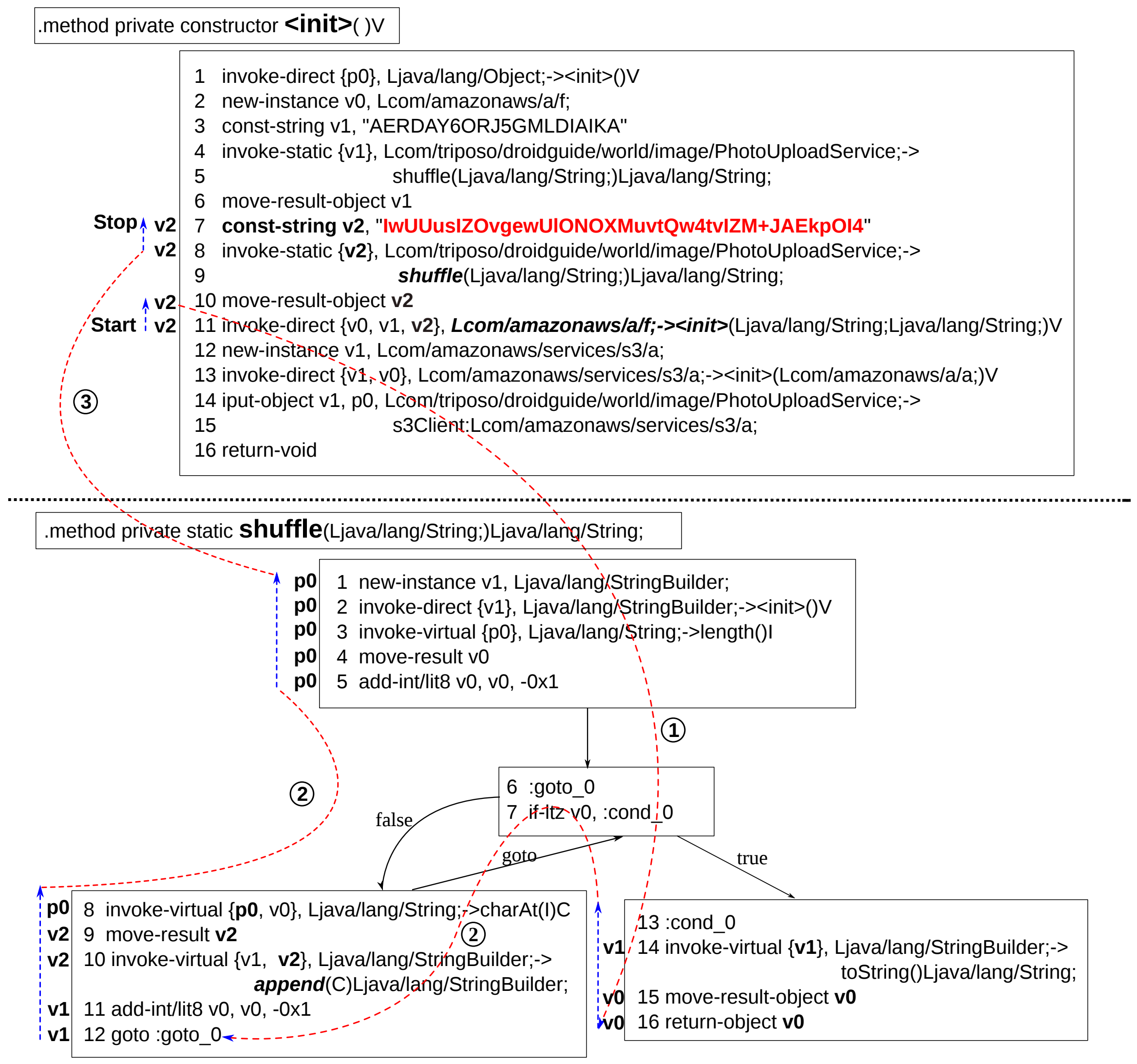
**Mapping Obfuscated Methods:** In the second step, we further de-obfuscate the methods of matched classes. We first try to determine the mapping by comparing constant strings in the methods. Our experience shows that this simple approach is very effective as constant strings are usually not obfuscated. If it fails to determine a match, we compare method signatures with the n-gram similarity, and leave the more precise control-flow-graph-based similarity comparison as a backup [31]. If a mapping still cannot be decided, we preform the manual analysis as the last resort. This rarely happens in practice.

After the above two steps, we can reliably identify whether an app contains an interesting library, and, if so the mapping between the obfuscated classes/methods and the original ones. In the rest of this paper, we assume that classes and methods have been de-obfuscated and only use their original names.

## 3.3 Identify Data Sources

After selecting the candidate apps, CredMiner uses static backward program slicing to locate the sources for an embedded credential and generates a program slice which contains all the instructions affected by those sources. The nature of a source can help us to decide whether the app is vulnerable or not. For example, if the credential originates from user inputs (e.g., the `EditText` widget), the app likely is not vulnerable since the credential is not embedded in the app.

**Locating Sink Methods** For each interesting library, we first identify sink methods in the library, i.e., the methods that accept a plaintext credential in the parameters. For example, the constructor of `javax.mail.PasswordAuthentication` is a sink method for JavaMail because it accepts the email account name and password. Similarly, `com.amazonaws.auth.BasicAWSCredentials`'s constructor is also a sink method because it accepts the Amazon AWS access key ID and secret access key. Note that this process requires manual efforts and domain knowledge of the interesting libraries. After identifying sink methods, we search for invocations of those methods in the app. That is, we locate where those methods are



```
.method private constructor <init>( )V

1  invoke-direct {p0}, Ljava/lang/Object;-><init>()V
2  new-instance v0, Lcom/amazonaws/a/f;
3  const-string v1, "AERDAY6ORJ5GMLDIAIKA"
4  invoke-static {v1}, Lcom/triposo/droidguide/world/image/PhotoUploadService;->
5                        shuffle(Ljava/lang/String;)Ljava/lang/String;
6  move-result-object v1
7  const-string v2, "IwUUusIZOvgewUlONOXMuvtQw4tvIZM+JAEkpOI4"
8  invoke-static {v2}, Lcom/triposo/droidguide/world/image/PhotoUploadService;->
9                        shuffle(Ljava/lang/String;)Ljava/lang/String;
10 move-result-object v2
11 invoke-direct {v0, v1, v2}, Lcom/amazonaws/a/f;-><init>(Ljava/lang/String;Ljava/lang/String;)V
12 new-instance v1, Lcom/amazonaws/services/s3/a;
13 invoke-direct {v1, v0}, Lcom/amazonaws/services/s3/a;-><init>(Lcom/amazonaws/a/a;)V
14 iput-object v1, p0, Lcom/triposo/droidguide/world/image/PhotoUploadService;->
15                       s3Client:Lcom/amazonaws/services/s3/a;
16 return-void
```

```
.method private static shuffle(Ljava/lang/String;)Ljava/lang/String;

1  new-instance v1, Ljava/lang/StringBuilder;
2  invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V
3  invoke-virtual {p0}, Ljava/lang/String;->length()I
4  move-result v0
5  add-int/lit8 v0, v0, -0x1

6  :goto_0
7  if-ltz v0, :cond_0

8  invoke-virtual {p0, v0}, Ljava/lang/String;->charAt(I)C
9  move-result v2
10 invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
                       append(C)Ljava/lang/StringBuilder;
11 add-int/lit8 v0, v0, -0x1
12 goto :goto_0

13 :cond_0
14 invoke-virtual {v1}, Ljava/lang/StringBuilder;->
                       toString()Ljava/lang/String;
15 move-result-object v0
16 return-object v0
```

Figure 5: An Example of Backtracking (Tracked Registers are Annotated to the Left of Interesting Instructions.)

called. At *run-time*, the inputs to a sink method contain plaintext credentials. If the credential is obfuscated, it must have already been reversed when it reaches the sink methods at *run-time*. Consequently, we could backtrack from call sites of a sink method to locate sources of the credential.

**Backtracking Credentials** After locating a call site of a sink method, we perform a static backward program slicing to backtrack the credential to its sources [38]. Figure 5 uses an example to illustrate this process. At each step, CredMiner maintains a set of registers that are currently being tracked (we call them the tracked registers.) The tracking process starts from the inputs to the sink method. For example, register $v_2$ on line 11 of the `<init>` method is one of the tracked registers (the other one is register $v_1$ on the same line). It is an input to the constructor of `com/amazonaws/a/f`. This class is obfuscated from `com/amazonaws/auth/BasicAWSCredentials`, whose constructor is a sink method that accepts Amazon AWS credentials. The class name has been internally reversed to the unobfuscated form during the candidate app selection (Section 3.2). Our backtracking algorithm recursively follows the use-def chain over the app's control flow graph. It first continues in the current basic block. When it reaches the boundary of the basic block, it follows to all the basic blocks that can reach the current one (② in Figure 5). If the tracked register is the return value of a method invocation, CredMiner follows into the callee (①). When the algorithm reaches the entry point of a method (line 1 of the `shuffle` method), CredMiner finds all the call sites of this method and continues the process in the callers if any parameter of the callee is in the tracked register set (③). During this process, the registers that are currently being tracked are updated according to instruction types. In the following, we describe how this works for different types of instructions.

- Irrelevant instructions: if an instruction does not affect the tracked registers, CredMiner simple skips it and moves backward.

- Instructions that overwrite tracked registers: for an instruction that overwrites one or more tracked registers, CredMiner adds the source registers of this instruction to the set of tracked register, and removes the destination registers from it. For example, instruction `move` $p_1$, $v_x$ moves the value from register $v_x$ to register $p_1$. If $p_1$ is being tracked, CredMiner adds $v_x$ to the set and remove $p_1$ from it. An interesting case is the `iget-object` instruction, which retrieves the value of a class field to the destination register. For example, the `iget-object`,$p_1$,$v_x$,`myClass;->`

`myField` instruction moves $v_x$`->myField` into the register $p_1$. $v_x$ is an object of class `myClass`. In this instruction, there is no source registers that CredMiner can add to the set since $v_x$`->myField` is an object field. To address that, for each `iget-object` instruction, CredMiner finds its corresponding `iput-object` instructions and adds their source registers to the set. To continue the previous example, register $v_y$ will be added if CredMiner finds the following instruction in the app: `iput-object,` $v_y$`,`$v_x$`, myClass; ->myField`. In this sense, our slicing algorithm can be considered as field-sensitive.

- Method invocation instructions: if the return value of an invoked method (the callee) is in the tracked register set, this method invocation instruction should be processed. Specifically, CredMiner follows into the callee, and continues backward slicing from the return instruction. Figure 5 contains an example of this case. The return value of the `shuffle` method is in the tracked register set at line 10 of the `<init>` method. CredMiner follows into `shuffle` and continues backward slicing from the `return-object` instruction (line 16) of this method. However, CredMiner does not track into a callee if it is a method of a system class. Instead, CredMiner models the transformation from the return value to the parameters of the method. For example, the instruction on line 15 of the `shuffle` method invokes the system method (`Ljava/lang/StringBuilder->toString`), we model the data flow of this method and directly propagate the return value to the parameter ($v_1$ in line 14).

**Finding Source Methods** A source method is a method where backtracking stops. In other words, the algorithm has found the instructions that provide the raw forms of the credential. Specifically, backtracking stops when one of the following conditions holds. *First*, it meets a `constant-string` instruction and the destination register of this instruction is in the set (e.g., line 7 of the `<init>` method in Figure 5). *Second*, the return value of certain system methods is stored to a tracked register. These methods usually return data from external sources, such as a file, a network connection, and the keyboard input. For example, if the return value of the `android.widget.EditText.EditText.getText` method is stored to a tracked register, we know that the credential originates from user inputs and stop backtracking.

## 3.4 Reconstruct Credentials

After locating the raw credential, CredMiner automatically reconstructs the credential if it originates from a constant string. This is particularly useful if the credential is encoded or encrypted since we do not need to write an app-specific tool to reverse-engineer the credential. CredMiner uses an execution engine to execute the program slice. The engine maintains an internal representation of Java objects, and interprets the Android bytecode on them. There are several challenges need to be addressed.

First, the program slice is an incomplete subset of the app code. It may rely on objects created out of the slice. We solve this problem by creating mock objects on demand. Specifically, CredMiner creates an internal representation for the object, and interprets its constructor on the object. We use simple default values for the parameters (e.g., an empty string for a string parameter).

Second, even though the engine can create objects on demand, there are still some cases it cannot handle, especially when the class of the object is deeply coupled with the Android framework. For example, the `android.content.context` class is too complicated for the engine to handle. Fortunately, the engine focuses on reconstructing credentials, instead of executing arbitrary slices.

```
1  def StringBuilder_append_C_Ljava_lang_StringBuilder(arg_regs):
2    #get the register value. Register names are in the
3    #parameter arg_regs, v1, v2 for instance
4    arg1_value = vm.get_reg_value(arg_regs.split(",")[0].strip())
5    arg2_value = vm.get_reg_value(arg_regs.split(",")[1].strip())
6    #check the type
7    if type(arg1_value) != str or type(arg2_value) != str:
8      print ("[x] [%d] we expect an (string, string) here. But"
9             "the actual type is (%s,%s)" % (get_linenumber(),
10            str(type(arg1_value)), str(type(arg2_value))))
11     return False
12   #emulate java.lang.StringBuilder.append(char)
13   arg1_value += arg2_value
14   vm.put_reg_value(arg1, arg1_value)
15   return True
```

Figure 6: Emulated `java.lang.StringBuilder.append(char)`

Most common instructions in our program slices transform and manipulate strings.

Third, how to handle system libraries is another challenging problem. In CredMiner, the execution engine has built-in knowledge to emulate the semantics of selected system methods. Figure 6 shows how the `java.lang.StringBuilder.append(char)` method is emulated in Python (the engine is implemented in this language.) CredMiner keeps the Dalvik VM registers in a dictionary, indexed by the register names. A Java string object is represented by a Python string. The code in Figure 6 first retrieves two registers from the register file (line 4 and 5). It then checks their types to ensure that they both are strings. If the type check is passed, CredMiner emulates the `append(char)` method by concatenating two strings together (line 13). Finally, the result is stored in the destination register.

Last but not least, the semantics of the Dalvik instructions should be maintained during the interpretation. In particular, the execution engine takes the following actions according to instruction types. *a*) control flow transfer instructions: for an unconditional control flow transfer instruction (e.g., `goto`), the target is encoded in the instruction as a label. CredMiner extracts this label and changes the execution flow to the target instruction. For a conditional control flow transfer instruction (e.g., `if-ne`), CredMiner first tests the condition and only jumps to the target if the condition satisfies. *b*) ALU instructions: ALU instructions are straightforward to handle. CredMiner performs the ALU operations and saves the results into destination registers. *c*) object field manipulation instructions: these instructions are probably the most common instructions in an app. As mentioned earlier, the related Java object may not exist in the program slice when the instruction is executed. CredMiner creates a mock object on demand and then manipulates its fields. *d*) method invocation instructions: for a method invocation instruction, CredMiner resolves the callee and invokes it. If the callee is a non-static method, the `this` object may not exist. CredMiner creates a mock object for `this` if necessary. Moreover, each method has its own register file. CredMiner allocates a new Python dictionary for the register file when entering a method, and discards it when the method returns.

## 3.5 Validate Credentials

The final step of CredMiner is to check whether the recovered credentials are still valid. To this end, we leverage the original app as much as possible. Specifically, we run the app in an Android emulator and monitor its execution. We set a watch point at the sink methods of the interesting libraries, and compare the *run-time* parameters, which contain the actual plaintext credentials, to those recovered by CredMiner. We also monitor the interaction between the app and the remote server to check whether the authentication succeeds or not. If it succeeds, the credential is still valid. In some cases, we control the emulator to drive the app down to a selected path. For example, some apps embed a large number of email accounts, and randomly select one for use. We control the returned random number to force the app to try each email account. Overall, this manual process is rather time consuming (without CredMiner's static analysis, it cannot scale to tens of thousands apps in our repository.), but it has virtually no negative impact to the security and privacy of the developer's account. The validation of Amazon TVM credentials are more involved than this and is described in Section 4.

## 4. EVALUATION

To evaluate the effectiveness of CredMiner, we collected a number of apps from Google Play and several alternative Android app stores in Asia. The apps from Google Play were downloaded from February to June 2014, and the apps from other stores were downloaded in June 2014. Table 1 shows the statistics of these apps. In total, there were 36,561 distinct apps (we consider apps with different SHA1 hash to be different.) All the results reported in this section were based on the experiments conducted in July 2014, unless otherwise stated.

Table 1: Distribution of Collected Apps

| | Google Play | Alternative Markets |
|---|---|---|
| # of Apps | 21,092 | 15,479 |
| Percentage | 57.67% | 42.33% |
| # of Total Apps | 36,571 | |
| # of Distinct Apps | 36,561 | |

### 4.1 Overall Results

CredMiner mainly focuses on two types of credentials, email accounts and Amazon AWS accounts. Among the apps we collected, there are 237 and 196 candidate apps that use the JavaMail library and the Amazon AWS SDK, respectively. CredMiner reveals that 51.5% (121/237) and 67.3% (132/196) of these candidate apps are vulnerable. Table 2 shows the distribution of the vulnerable apps among different app stores. Interestingly, 96.9% (128/132) of the vulnerable apps that leak Amazon AWS credentials come from Google Play. This is probably because Amazon AWS has a low market share in Asia. We have reported our findings to the developers of these vulnerable apps. Most of these apps do not have an official channel to report vulnerabilities. In this case, we tried to contact them through the customer support emails listed on the app store. We only received very few responses from the developers. In two exceptional cases, the developers were able to acknowledge the issues within 24 hours with fixes planned in the near future. Because of the scarcity of responses, we omit the vulnerable app names and obfuscate other details in the paper to protect user privacy.

Table 2: Distribution of Vulnerable Apps

| Category of Credentials | Google Play | Other Stores | Total |
|---|---|---|---|
| Email Credentials | 65 | 56 | 121 |
| Amazon AWS Credentials | 128 | 4 | 132 |

### 4.2 Email Credentials

The first category of the studied apps uses free email services. Our data set has 237 candidate apps in this category, and 121 (51.1%) of them were found to be vulnerable. In particular, 65 of these vulnerable apps were downloaded from Google Play with 11 apps having more than 50,000 downloads.

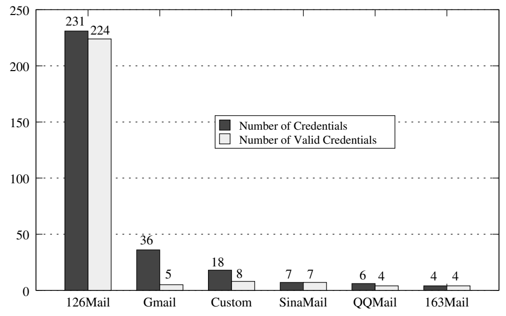
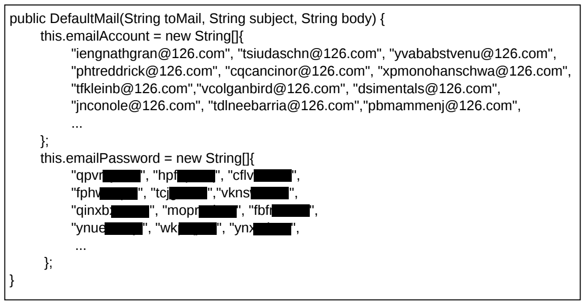


Figure 7: Leaked Credentials from Different Providers

There are a number of reasons why some candidate apps are not vulnerable. First, the imported JavaMail library is not used in the app. It may have been included as the testing code, but the developer forgets to remove it before releasing the app. Second, the library is not used to send emails but for other purposes. For example, the `javax.mail.internet.InternetAddress.validate` method checks whether an email address is in a valid format or not. We observed several apps in this case. Third, the credential comes from user inputs, not the app itself. This is the most common reason that a candidate app is not vulnerable.

**Credentials Breakdown** CredMiner recovered 302 unique email credentials. This number is larger than that of the vulnerable apps because an app may include several email credentials. In fact, we found one app that had 130 embedded email accounts. At the time of our experiments, 252 of these 302 accounts were still valid. Attackers could use these credentials to access all the stored emails, and to send spam or phishing emails as the developers. It's worth noting that email credentials can often login into other services from the same provider. For example, most Google services use the Gmail account to authenticate a user. Consequently, the data stored in those services are in jeopardy due to the leaked passwords. We notice that there is a trend for service providers to deploy 2-step verification. This significantly reduces the risk of leaked passwords. Though, it is questionable whether the same developer that embeds plaintext passwords in the app would care to use it. Figure 7 shows the number of total and still-valid credentials ordered by email providers. Most of these accounts belong to `126.com` (a free email provider in China), from which 96.9% accounts were still valid. We also recovered 18 Gmail credentials, but only 5 of them (13.8%) were still valid. In addition to free email services, we detected 18 email accounts that were using their own servers, including Amazon Simple Email Service.

**Vulnerable Malware** When analyzing these apps, we came up with an interesting thought to see whether we can recover email credentials from malware samples. We expect malware authors to be more careful in protecting their email accounts since these accounts may reveal their real identities, expose network traces, and provide evidences to the law enforcement. We collected 1,404 malware samples from two sources, Android Malware Genome Project [46] and a mobile malware sharing website [7]. Surprisingly, CredMiner recovered 8 email credentials and 1 of them was still valid after around 3 years. This sample came from Android Malware Genome Project, which contains Android malware collected from August 2010 to October 2011. In addition, only 2 of these vulnerable samples protect their accounts with custom string encoding schemes. The main purpose for malware to use email services is to send the stolen private information, such as SMS messages, recorded background voices, and contacts, to the attacker.



```
public DefaultMail(String toMail, String subject, String body) {
        this.emailAccount = new String[]{
                "iengnathgran@126.com", "tsiudaschn@126.com", "yvababstvenu@126.com",
                "phtreddrick@126.com", "cqcancinor@126.com", "xpmonohanschwa@126.com",
                "tfkleinb@126.com","vcolganbird@126.com", "dsimentals@126.com",
                "jnconole@126.com", "tdlneebarria@126.com","pbmammenj@126.com",
                ...
        };
        this.emailPassword = new String[]{
                "qpv[redacted]", "hpf[redacted]", "cflv[redacted]",
                "fph[redacted]", "tcj[redacted]","vkns[redacted]",
                "qinxb[redacted]", "mopr[redacted]", "fbf[redacted]",
                "ynue[redacted]", "wk[redacted]", "ynx[redacted]",
                 ...
         };
}
```

Figure 8: Some of the 130 Email Credentials in Case Study III

**Case Study I** This is a popular app from Google Play with more than 1,000,000 downloads. Its main function is to securely hide user pictures with a user-provided pin. If the user forgets the pin, a link will be sent to his/her email to recover the pin. JavaMail is used to send this email. The email credential for this purpose is obfuscated through encryption and encoding. Specifically, it is first encrypted using AES in the CTR mode. The encrypted credential is then encoded again in Based64, and the resulting blob (`K/JFSGE54oU=`) is embedded in the app. However, the key to decrypt the blob is only encoded in Base64, and can be easily retrieved. In addition, the initial vector for AES is simply a constant string directly embedded in the app. After decoding and decrypting the blob, CredMiner shows that the plaintext password is `abc123**`. It is sadly interesting to see that the developer spent such efforts to protect a very weak password. In addition, we also found a plaintext email credential (username: `handy****@gmail.com`, password: `key_******`) from the app. However, this credential is not used by the app. We suspect that it might have been used by the previous versions of the app, until the developer decided to protect the credential. Another app from the same developer employs a similar strategy to protect its email credentials.

**Case Study II** This is a spyware downloaded from Google Play in March 2014. It has since been removed from Google Play. The app stealthily tracks the SMS messages received and sent by the victim. It uses Amazon Simple Email Service to send them to a specific email account. The Amazon AWS credential to send those emails is directly placed in the app, without any protection. It was still valid at the time of our experiments, even after the app had been removed from Google Play.

**Case Study III** We downloaded this app from an alternative Android app store in June 2014. The app includes a feature to collect user feedback through emails. Instead of using the default email app for this purpose (the developer might consider this hurting the user experience since it requires to switch to the default email app and back), the app sends these emails in the background using an embedded account. Surprisingly, we discovered 130 different email accounts from this app, all in the plaintext. They all belong to the same free email service provider, `126.com`, and are still valid then. The app randomly picks one to use. We suspect this is the developer's effort to circumvent the service provider's quote on how many emails one account can send in a period. Figure 8 shows some of those email accounts.

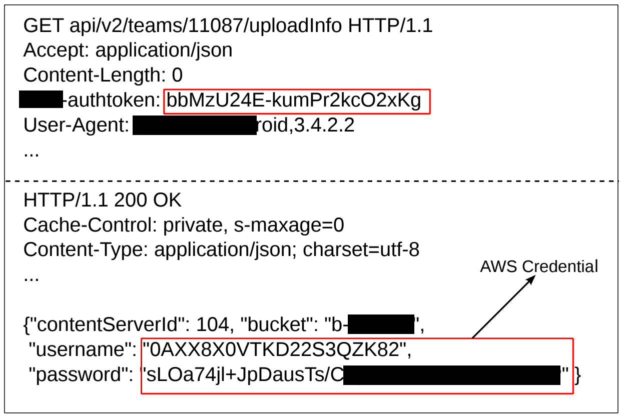
## 4.3 Amazon AWS Credentials

The second category of the studied apps uses Amazon AWS credentials. An Amazon AWS credential consists of an access key ID and the related secret access key (they are semantically similar to a username and password.) It may be shared by multiple services from Amazon, depending on the subscription and the configuration. There are 196 candidate apps in our data set using its SDK, and 132 (67.3%) of them are vulnerable. Most invulnerable candidate apps import the SDK but never use it. There are also a small number of apps that employ the recommended secure practice and thus are not vulnerable. However, the majority of these candidate apps are vulnerable. Most of these apps (96.9%) came from Google Play. Particularly, 24% of the vulnerable apps (32 apps) have more than 50,000 downloads. In total, CredMiner recovered 58 unique AWS credentials, and 28 of them are still valid during our experiments.

**Comparison to PlayDrone** A system called PlayDrone [36] can also detect AWS credentials in Android apps. It uses simple string matching to locate plaintext AWS credentials which are "pairs of strings matching `AKIA[0-9A-Z]{16}` and `[0-9a-zA-Z/+]{40}` that are at most 5 lines apart." This simple approach suffers from both false negatives (e.g., encoded/obfuscated credentials) and false positives (e.g., strings that accidentally match the criteria). In comparison, CredMiner's approach is more generic and reliable. For example, CredMiner can automatically reconstruct certain encoded/obfuscated credentials, and it works on other types of credentials as well. In this experiment, we compare the effectiveness of PlayDrone and CredMiner. Specifically, we implemented a prototype of PlayDrone, and applied it to our candidate apps. PlayDrone successfully recovered credentials from 105 apps, while CredMiner recovered credentials from 126 apps, a 20% improvement. All the apps detected by PlayDrone were also detected by CredMiner. It's worth noting that both CredMiner and PlayDrone missed 6 vulnerable apps. Five of these apps employ the anonymous token vending machine (Section 2.1), and the other one uses a custom protocol to protect the credential. We will discuss these apps in detail in the next paragraph.

**Anonymous Token Vending Machine** Amazon advises developers to protect their credentials with the token vending machine (TVM) [3]. Instead of using an embedded credential, the app can obtain a temporary credential from a remote TVM server. This architecture protects the long-term developer credentials from being leaked. However, our experiments show that developers often mis-configure the TVM servers, leaving their apps vulnerable. Particularly, developers often place the URL to the remote TVM server directly in their apps. An attacker can send a request to this URL to obtain a temporary credential for his/her malicious purposes. The privilege of this credential is inherited from the developer's STS credential (Section 2.1). If the developer fails to constrain the privilege of the STS credential, the attacker's credential has more power than necessary. For example, we configured our own TVM server by following the (insecure) tutorial provided by Amazon [14]. It turns out that temporary credentials generated by our server could be used to fully access the Amazon S3 service (e.g., to read/write all the files in our buckets). We suspect some candidate apps that use TVM have similar issues because developers are known to copy and paste the sample code. To that end, we manually analyzed some popular candidate apps that use anonymous TVM. The results are not optimistic. We will describe one of the examples in the following.

This particular app from Google play is very popular. It has accumulated more than 5,000,000 downloads. The app backs up the user data to the Amazon S3 cloud using a temporary credential obtained from an anonymous TVM server. For each user, it creates a unique UID, and stores his/her data under a directory associated to this UID. If the temporary credential is not properly confined, the attacker might be able to enumerate the UIDs and access the app users' files. To confirm whether this is the case or not, we created two testing accounts using the app, and retrieved their UIDs from the app's sqlite database. We then obtained a temporary credential for one of the testing accounts. This credential allowed us to access



```
GET api/v2/teams/11087/uploadInfo HTTP/1.1
Accept: application/json
Content-Length: 0
[redacted]-authtoken: bbMzU24E-kumPr2kcO2xKg
User-Agent: [redacted]roid,3.4.2.2
...
-----------------------------------------------
HTTP/1.1 200 OK
Cache-Control: private, s-maxage=0
Content-Type: application/json; charset=utf-8
...

{"contentServerId": 104, "bucket": "b-[redacted]",
 "username": "0AXX8X0VTKD22S3QZK82",
 "password": "sLOa74jl+JpDausTs/C[redacted]"}
```

Figure 9: HTTPS Request and Response to Get Amazon Credential

the other account's files. This experiment demonstrated that the developer failed to properly configure the TVM server to strictly isolate different app users. Considering the huge popularity of this app, this vulnerability poses serious threats to the user privacy. In short, even though Amazon TVM has a reasonably secure design, it is not widely used, let alone used securely.

**Case Study I** This is a popular travel guide app. It has been downloaded in Google Play for more than 100,000 times. The app uses Amazon S3 to store its users' pictures. To protect its AWS credential, the app encodes the credential by reversing its order. For example, the access key ID `AKIAIDLMG5JRO6******` is encoded to `******6ORJ5GMLDIAIKA`. CredMiner successfully identified and recovered the encoded credential. We reported this issue to the app's developers. They promptly acknowledged the issue and pointed out that the credential can only be used to upload user pictures and they are monitoring the credential for abuse.

**Case Study II** This app allows sports teams to upload, share, and study training videos. The user has to register an account on the developer's website to use the app, or signs in with a demo account to test-drive the app. When the demo account is used, the app retrieves an Amazon credential from its server. This credential can be easily obtained and misused by an attacker. CredMiner flags this app as suspicious because the credential comes from the Internet. We manually analyzed this app. When a request is sent to a special URL bearing the demo account's authentication token, the server returns a JSON document that contains the AWS credential. Figure 9 shows the request sent to the server and the returned credential. Fortunately the developer has fixed this vulnerability in the latest update.

## 4.4 App Developer's Security Practice

In this section we try to categorize app developers' insecure practices in protecting their credentials, and discuss potential reasons behind these practices.

*First*, many developers do not protect their credentials at all. Their credentials are directly embedded in the app without any protection. Among the 132 vulnerable apps that use Amazon AWS, 105 (79.5%) apps belong to this category. *Second*, some developers do understand the risk and take precautions to protect their credentials through encryption or encoding. However, the keys to reverse the process are stored in the app, often in plaintext. This is only marginally better than the first practice because it is really straightforward to reverse-engineer Android apps. Another 21 vulnerable apps that use Amazon AWS belong to this category. *Third*, some developers follow the best practice recommended by service providers to protect their credentials. However, these mechanisms are often complicated and the developers lack the necessary knowledge to deploy them in a secure way. For example, developers who use Amazon TVM often fail to specify the least privilege for temporary credentials.

We further speculate the potential reasons for the developers' insecure practices. *First*, insecure sample code provided by service providers can mislead developers. For example, Amazon used to provide sample code that directly embeds the Amazon AWS credential. Developers often copy-and-paste the sample code into their own apps without giving it a second thought, even though there is a warning saying that the sample code is not secure. We discovered several such apps whose packages have the same variable names, classes names, and configuration file names as the sample code. It is a relief to see that Amazon has removed the insecure sample code from the recent releases of the SDK (version `2.0.5`). *Second*, there is no clear instructions about how to use the SDKs securely. One example is the Amazon token vending machine (TVM). The app developer could follow the instructions given by Amazon to set the privilege for temporary credentials [14]. Unfortunately, the sample policy given in that document is not secure – it allows a temporary credential to access all the resources. Simple, clean, and secure document would help developers to secure their systems. For complicated systems like Amazon TVM, it would also help if the providers could include them as a default configuration of their services (instead of asking developers to run their own TVM servers).

## 4.5 Other Cloud Services

Although we focused our study on two types of credentials, CredMiner works for other types of credentials as well. In the following, we briefly describe the evaluation results with the Microsoft mobile service [9] and the Aliyun OSS [1]. Both services are similar to Amazon S3. Apps use them to synchronize user data to the cloud. However, these two services are not popular in our data set with less than 10 apps in total. Six of these apps are vulnerable. For example, one app stores the user's email address and Facebook authentication token in the Azure cloud storage. The insecurely embedded Azure credential allows an attacker to retrieve these data. In particular, the Facebook token can be used to access the user's Facebook account without further authentication.

## 5. DISCUSSION

CredMiner systematically mines developer credentials in Android apps to study the prevalence of improperly handled credentials. In this section, we discuss possible improvements to CredMiner.

First, CredMiner uses call graphs and control flow graphs to compute the sources of credentials. However, due to the object-oriented nature of the Java programming language and the event-driven design of Android apps, it's challenging to construct these graphs precisely. For example, the actual class of a virtually invoked method or the exact type of a Java object sometimes can only be determined at run-time. Our system employs a simple but conservative class-hierarchy based points-to analysis to generate approximate call graphs and control flow graphs. This simple analysis works for our system, but may not perform well for more complicated cases. A precise points-to analysis can be integrated into our system to improve the overall preciseness of CredMiner. Moreover, the event-driven design of Android apps and the Java reflection could break call graphs. Similar to the previous systems [24,30,45], we manually connect some of these methods, such as `Thread.start` and `Thread.run`. CredMiner also partially supports Java reflection by resolving class and method names that are simple strings.

Second, CredMiner does not handle native libraries in an app. It thus will miss credentials hidden inside native libraries. We did a quick measurement of how many apps may belong to this category. Our experiment is based on the observation that the native code eventually needs to call methods in the SDK to access the service, especially for the services that do not provide a native SDK. We can identify such apps by searching their native code for the class/methods names of the SDK and of other Java classes that call the SDK (they can be used to indirectly call the SDK.) There are only 3 apps in our data set having the native code that use the Amazon AWS service. This is an underestimation because we cannot handle obfuscated class/method names in the native code. Our current prototype cannot handle these apps.

Third, our execution engine selectively executes the program slice to reconstruct credentials. This proves to be rather convenient, especially if the credentials are encoded or encrypted, since we don't need to write app-specific tools for this task. For our purpose, there is no need to build a complete, generic execution engine. The current prototype is tailored to credential reconstruction. Manual efforts may still be necessary, especially if the program slice contains complicated system classes which our prototype cannot handle. Our experience with a large number of apps shows that operations in these program slices are mostly related to string manipulation and transformation, which can be readily handled by our prototype.

Finally, how to securely protect developer credentials in apps is a challenging problem. Most obfuscation techniques will eventually be reverse-engineered. In the following, we try to propose several possible mitigation measures to alleviate this issue: *a*) App store operators could deploy CredMiner to detect vulnerable apps and warn the developer when such an app is uploaded to the store. This does not fundamentally fix these vulnerabilities, but at least pushes the developer to pay more attention to this issue. *b*) Sample code released by vendors should demonstrate only secure and correct ways to use their SDKs, and avoid misleading developers with quick-and-dirty code. For example, an early version of the sample code for the Amazon AWS SDK embeds the credential directly in the app. CredMiner detected a few apps that seemed to copy-and-paste the insecure sample code. It is encouraging to see that the insecure sample code has been removed from the recent versions of the SDK. *c*) Architectures like Amazon TVM provide reasonably secure ways to authenticate users. Their correct deployment should be documented and demonstrated simply and cleanly. It would also help if they can be packaged as turnkey solutions with minimal configuration and maintenance from the developers. *d*) Developers should consider alternative and more secure ways to implement some features. For example, instead of using free email services to send crash reports, they can use third-party crash analysis libraries [6] or send the reports with the default email app.

## 6. RELATED WORK

**Static Analysis of Android Apps** Our system leverages static analysis to detect vulnerable Android apps. Static analysis has been widely used by previous systems for various purposes: first, it has been applied to detect malicious apps. DroidRanger [49] and RiskRanker [25] are two such systems. DroidRanger uses app permissions and behavior footprinting to detect new instances of known malware. RiskRanker instead relies on reachability analysis to detect unknown malware. One aspect shared by CredMiner and RiskRanker is that both systems use backward program slicing to locate source methods. Second, static analysis has been used to detect capability leaks or confused-deputy vulnerabilities in third-party or pre-installed apps [27]. Such systems include ComDroid [17], Woodpecker [24], CHEX [30], Amandroid [37], and SEFA [39]. For example, CHEX statically detects component hijacking vulnerabilities that can be exploited to gain unauthorized access to protected or private resources. A system that can automatically patch such vulnerable apps has also been proposed [41]. The most related work in this category is probably SAAF [28], a static analysis framework for Android that supports program slicing. CredMiner uses a similar data flow analysis, but it also has an execution engine to selectively execute program slices in order to reconstruct credentials.

**Similarity Comparison of Android Apps** CredMiner uses code similarity to de-obfuscate interesting libraries. Code similarity has been employed to detect repackaged Android apps, such as DroidMOSS [44], Juxtapp [26], PiggyApp [43], AnDarwin [18], and DexDiff [31]. For example, DroidMOSS generates the fingerprint for an app with fuzzy hash. Juxtapp [26] computes the app similarity based on the hashes for k-grams of the opcodes. CredMiner uses not only opcodes but also other components of classes and methods to calculate the similarity. DexDiff compares the similarity of the AOSP (the Android Open Source Project) apps and the pre-installed apps in a third-party phone to systematically audit the vendor customization. CredMiner uses a similar signature as DexDiff. However, we do not include class and method names in the signature since they may have been changed by code obfuscation. Overall, the basic ideas of Android code similarity are similar but differ in their purposes and details.

**Detection of Vulnerable Apps** Many Android apps have security issues due to developers' poor security practices. First, user data could be leaked to open interfaces in Android, such as content providers. ContentScope [45] is a system that can systematically detect content leak and content pollution vulnerabilities. Second, an empirical study [19] of Android apps shows that app developers often do not use cryptographic APIs securely. About 88% of the studied apps make at least one mistake. Our experience also confirms this observation. For example, we find some apps that use constant numbers as the initial vector of AES. Third, MalloDroid [22] reveals that the SSL/TLS library is frequently improperly used by Android apps, leaving these apps vulnerable to the man-in-the-middle attack. SMV-HUNTER [35] is proposed to detect SSL/TLS man-in-the-middle vulnerabilities using static and dynamic analysis. CredMiner also aims at exposing vulnerable apps. However, we focus on detecting apps that leak developer credentials. A recent system called PlayDrone [36] performs a large scale survey of Android apps in Google Play. It can detect embedded plaintext Amazon AWS credentials with simple string matching. CredMiner uses a more comprehensive approach. It can automatically detect and recover transformed or obfuscated credentials. Our evaluation shows that CredMiner can detect about 20% more vulnerable apps than PlayDrone in our data set.

**Detection of Privacy Leaking Apps** Smartphone privacy has attracted lots of attention nowadays. Earlier research discovered that user privacy could be leaked through third-party apps on the Android and iOS platforms. For example, TaintDroid [21] uses dynamic taint analysis to detect privacy leaks in Android apps, while PiOS [20] targets the iOS platform. Several systems [16, 29, 32, 40, 42, 47, 48, 48, 50] have been proposed to mitigate this threat. They either extend the Android framework to provide fine-grained permission control, or repackage apps to avoid changing the framework. For example, Aurasium interposes the interaction between apps and system libraries to regulate the access to the private user data [40]. In-app ad libraries have also been reported to leak private information [23]. To address that, AdDroid separates the advertisement functionality from the host app [33] , while AdSplit moves the advertisement code into another process [34]. CredMiner differs from these efforts in identifying vulnerabilities that may leak developer credentials.

## 7. SUMMARY

In this paper, we aim at studying the prevalence of unsafe developer credential uses. Leaked credentials could compromise the privacy of the user data and even disrupt the apps totally. We have presented the design, prototype, and evaluation of CredMiner. CredMiner leverages static backward program slicing to identify the raw formats of the credentials and then uses an execution engine to automatically reconstruct the embedded credentials. We have applied our system to 36,561 apps. The results showed that 51.1% (121 of 237) of the apps that use free email services, and 67.3% of the apps (132 of 196) that use the Amazon AWS service were vulnerable. CredMiners recovered 302 and 58 unique email and Amazon AWS credentials, respectively. Among them, 252 and 28 credentials were still valid at the time of experiments. The existence of a large number of vulnerable apps as well as the poor security practices by the app developers reflect the severity of this threat and the urgent need to address it.

**Acknowledgements** We would like to thank our shepherd, David Barrera, and the anonymous reviewers for their insightful comments that greatly helped improve the presentation of this paper. We also want to thank Yu Wang, Huaguo Shi, Chiachih Wu for the helpful discussion. This work was partially supported by Qihoo 360 when the first author was an intern there. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the company.

## 8. REFERENCES

[1] Aliyun OSS. `http://www.aliyun.com/product/oss/`.

[2] Android Apps Crawler. `https://github.com/mssun/android-apps-crawler/`.

[3] Authenticating Users of AWS Mobile Applications with a Token Vending Machine . `https://aws.amazon.com/articles/4611615499399490`.

[4] aws-sdk-android-samples. `https://github.com/awslabs/aws-sdk-android-samples`.

[5] AWS SDK for Android. `http://aws.amazon.com/sdkforandroid/`.

[6] BugSense. `https://www.bugsense.com/`.

[7] contagiominidump. `http://contagiominidump.blogspot.com/`.

[8] DexGuard. `https://www.saikoa.com/dexguard`.

[9] Get started with Mobile Services. `http://azure.microsoft.com/en-us/documentation/articles/mobile-services-android-get-started/`.

[10] javamail-android. `https://code.google.com/p/javamail-android/`.

[11] ProGuard. `http://developer.android.com/tools/help/proguard.html`.

[12] Sending Emails without User Intervention (no Intents) in Android. `http://www.jondev.net/articles/Sending_Emails_without_User_Intervention_(no_Intents)_in_Android`.

[13] Smali. `https://code.google.com/p/smali/`.

[14] Token Vending Machine for Anonymous Registration - Sample Java Web Application. `http://aws.amazon.com/code/8872061742402990`.

[15] Using ProGuard with the AWS SDK for Android. `http://mobile.awsblog.com/post/Tx2OC71PFCTC63E/Using-ProGuard-with-the-AWS-SDK-for-Android`.

[16] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th International Workshop on Mobile Computing System and Applications*, 2011.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services*, 2011.

[18] J. Crussell, C. Gibler, and H. Chen. Andarwin: Scalable detection of semantically similar android applications. In *Proceedings of the 18th European Symposium on Research in Computer Security*, 2013.

[19] M. Egele, D. Brumley, and C. K. Yanick Fratantonio. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[20] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security*, NDSS, 2011.

[21] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[22] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgartner, and B. Freisleben. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[23] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

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

[25] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services*, MobiSys, 2012.

[26] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th DIMVA*, 2012.

[27] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22, October 1998.

[28] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[29] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[31] M. Mitchell, G. Tian, and Z. Wang. Systematic Audit of Third-Party Android Phones. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2012.

[32] M. Nauman, S. Khan, M. Alam, and X. Zhang. Extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[33] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[34] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the 21th USENIX Security Symposium*, 2012.

[35] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.

[36] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proceedings of the 14th ACM International Conference on Measurement and Modeling of Computer Systems*, 2014.

[37] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[38] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*, 1981.

[39] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[40] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21th USENIX Security Symposium*, 2012.

[41] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[42] M. Zhang and H. Yin. Efficient, Context-Aware Privacy Leakage Confinment for Android Applications without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, 2014.

[43] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of 'piggybacked' mobile applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, 2013.

[44] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, 2012.

[45] Y. Zhou and X. Jian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[46] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[47] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang. Hybrid User-level Sandboxing of Third-party Android Apps . In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[48] Y. Zhou, K. Singh, and X. Jiang. Owner-centric Protection of Unstructured Data on Smartphones . In *Proceedings of the 7th International Conference on Trust and Trustworthy Computing*, 2014.

[49] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS, 2012.

[50] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, 2011.