sPECTRA: a Precise framEwork for analyzing CrypTographic vulneRabilities in Android apps

Jyoti Gajrani*, Meenakshi Tripathi*, Vijay Laxmi*, M.S. Gaur*, Mauro Conti†, Muttukrishnan Rajarajan‡
*MNIT, Jaipur, INDIA, {2014rcp9542,mtripathi.cse,vlaxmi,gaurms}@mnit.ac.in
†University of Padua, Italy, conti@math.unipd.it
‡City University London, R.Muttukrishnan@city.ac.uk

***Abstract*—The majority of Android applications (apps) deals with user's personal data. Users trust these apps and allow them to access all sensitive data. Cryptography, when employed in an appropriate way, can be used to prevent misuse of data. Unfortunately, cryptographic libraries also include vulnerable cryptographic services. Since Android app developers may not be cryptographic experts, this makes apps become the target of various attacks due to cryptographic vulnerabilities.**

In this work, we present sPECTRA: an automated framework for analyzing wide range of cryptographic vulnerabilities in Android apps at large scale. sPECTRA is more precise and accurate in comparison to state-of-the-art approaches as it reduces both false negatives and false positives. The inclusion of Intelligent UI exploration during dynamic analysis makes sPECTRA deployable to analyze apps at large scale. Moreover, sPECTRA works on apk files without the need of any source code.

We evaluate sPECTRA on 7,000 apps collected from 7 most popular Android app stores. Results indicate that 90% of apps are exploitable because of cryptographic vulnerabilities. We made sPECTRA available as an open source[1].

***Index Terms*—cryptographic, APIs, vulnerabilities, Android, attacks.**

I. Introduction

With the rapid growth of smartphone technology, our daily life is becoming more dependent on smartphones. Among various smartphone technologies, Android share is worth 84.1% by Q1 2016 [1]. Users trust mobile apps and grant them access to their personal information. Therefore, all the private data that is taken by these apps either for storage on the device or for transmission out of the device must be secured with strong cryptographic services.

Cryptographic providers like Java Cryptographic Architecture (JCA) [7], BouncyCastle [14] and SpongyCastle [12] provide a set of cryptographic APIs (Application Program Interface) for developers. These cryptographic APIs and associated parameters must be used in the correct way to provide strong security guarantees otherwise the incorrect way may lead to attacks such as Man-in-the-mobile (MitMo) attacks, Brute-force attacks, and Dictionary attacks. In [19], researchers shown the first key recovery attack on full AES-128 with computational complexity $2^{126.1}$.

Android has become the preferred target for financial malware due to large market coverage and plenty of reported attacks. Cryptographers know what are the most secure parameters to be used with these APIs in the way to ensure strong security guarantee however, may not be the same for software developers. Developers may invoke a wrong API function, set incorrect parameters, and check the return values improperly and so on. Therefore, not only the developers but app distributors must check for vulnerabilities in apps before publishing them to markets to prevent any loss of end-user. Correct usage of cryptographic primitives (low-level cryptographic algorithms) such as strong encryption algorithms, random keys, key-length, padding with block ciphers, validation of SSL/TLS certificates, digital signature algorithms, salts, and iteration count ensures resilience against cryptographic exploits.

In this paper, we propose **sPECTRA**, an automated framework using hybrid analysis for detection of such vulnerabilities in Android apps to provide high-security guarantees to app users. The main contributions of sPECTRA are summarized as follows:

- sPECTRA analyzes a wide range of cryptographic vulnerabilities in comparison to state-of-the-art approaches.
- sPECTRA includes intelligent techniques to enable automated vulnerability analysis at large scale. We show the efficacy of sPECTRA by analyzing of 7000 Android apps collected from 7 different app repositories. The results show that almost 90% of the apps using cryptographic features are vulnerable.
- sPECTRA includes lightweight approaches to speedup the analysis.

We release sPECTRA as open source[1] to drive research in this direction. sPECTRA will be made available as web-based analysis service to benefit developers and app stores.

II. Cryptographic Primitives and Associated Vulnerabilities

This section briefly covers cryptographic primitives and the inappropriate usage of these primitives which makes the apps vulnerable to various cryptographic attacks.

A. Cryptographic APIs

Listing 1 shows the code that implements encryption of `IMEI number` using Password-Based Encryption (PBE). This code contains a set of cryptographic vulnerabilities.

[1]URL: https://bitbucket.org/spectra2016/sourcecode/src/32021e83cec2

TABLE I: Cryptographic Primitives and Associated Vulnerabilities

Primitive	Use/Focus	Vulnerabilities	Implemented Attacks	Few Relevant APIs Examples

Symmetric Encryption	Securely storing data/keys.
	Secure transmission of sensitive data.
		DES Algo (key 56 bits) [5]
		AES in ECB Mode
		AES key size <=128 bit [18]
		AES CBC/CTR Mode with Static IV [21]
			Chosen-plaintext attacks.
			Brute-force attacks
				`IvParameterSpec.init()`
				`KeyGenerator.init()`
				`Cipher.getInstance()`

Digital Signature	Digitally Signing certificates	Signing Algorithm SHA1withRSA [13]	Hash Collision	`Signature.getInstance()`
Padding	Randomizing the ciphers	NoPadding [21]	Padding oracle attack	`Cipher.getInstance()`
Message Digest	Hashes of user credentials. Data Integrity	Algorithms - MD4, MD5,SHA-1 [15]	Length Extension attacks. Brute-force attacks	`MessageDigest.getInstance()`
Password-Based Encryption	Encyption based on password.	Iteration Count <1000 [11] Salt - Static	Dictionary-based attacks. Collision-based attacks	`PBEKeySpec.init()` `SecretKeyFactory.getInstance()`
Key Derivation	Random Key generation using secure PRNG	Static key Material	Brute-force attacks.	`SecretKeySpec.init()` `SecretKeyFactory.generateSecret()`
Pseduo-Random Number Generation (PRNG)	Random Salt Initialization Vector(IV) Nonces, OTPs	Constant Seed	Chosen-plaintext attacks. Collision-based attacks. Replay attacks.	`SecureRandom.setSeed()`
SSL/TLS protocol	Communication Security	HostnameVerifier, TrustManager	MitMo attacks.	-
On-Device Storage	Storing data in shared storage	writing to "`/sdcard/*`"	Covert/Overt channel attacks	`File.init()`

`SecureRandom` class provides a cryptographically strong pseudo-random number generator unfortunately, this is seeded with constant Seed (Line 4). Use of static value for seeding may completely replace the cryptographically strong default seed causing it to generate an anticipated sequence of salts (Line 7) which are unfit for secure use. Random salt restricts the attackers from pre-computing a dictionary of derived keys.

```
1 //Random-Number Generation
2 SecureRandom random=new SecureRandom();
3 byte[] seed = password.getBytes("UTF-8");
4 random.setSeed(seed);
5 //Salt Generation
6 byte[] salt=new byte[8];
7 random.nextBytes(salt);
8 //Key Generation
9 KeySpec ks = new PBEKeySpec(password, salt, 256, 128);
10 SecretKeyFactory f = SecretKeyFactory.getInstance("
      PBKDF2WithHmacSHA1");
11 SecretKey t = f.generateSecret(ks);
12 SecretKey sec = new SecretKeySpec(t.getEncoded(), "AES");
13 //Encrypt the message
14 Cipher c = Cipher.getInstance("AES/ECB/NoPadding");
15 c.init(Cipher.ENCRYPT_MODE, sec);
16 byte[] encrypted = c.doFinal((imei).getBytes());
```

Listing 1: Vulnerable use of cryptographic Primitives

`PBEKeySpec` (Line 9) used for generating `KeySpec` is vulnerable due to use of static salt (second parameter of `PBEKeySpec` API) and iteration count (IC) (third parameter of `PBEKeySpec` API) with value 256 (must be minimum 1000 [11]). Larger IC complicate the key derivation function and increases the difficulty of brute force attack. Further, it performs encryption using AES algorithm in ECB mode (Line 14) which has proven vulnerable. This is because this mode will generate the same cipher-text if the same plain-text is encrypted with the same key. The code specifies `NoPadding` as Padding Scheme (Line 14). However, Padding must be present with ciphers to prevent the cryptanalyst in predicting plain-text message length.

Table I summarizes various cryptographic primitives used by Android apps, associated vulnerabilities based on National Institute of Standards and Technology (NIST) and Federal Office for Information Security (BSI) recommendations, and the attacks that can exploit these vulnerabilities. To prevent Android apps from these attacks, all the cryptographic APIs must be critically analyzed. For each primitive mentioned in Table I, column 1, we exhaustively identified all APIs provided by all three libraries SunJCE, BouncyCastle and SpoungyCastle. Further, we analyzed all vulnerable values for each of these APIs and prepared vulnerability database. Table I, last column include some of the these APIs from SunJCE.

B. SSL/TLS Connection Validation

Apps use SSL/TLS protocols with the goal to securely transmit sensitive data to the server. During SSL connection establishment, two conditions are validated:

- Hostname of the server must match CommonName mentioned in certificate presented by server.
- There must exist trust chain between the certificate presented by the server and the root CA certificates preinstalled on mobile.

Two JCA classes `HostnameVerifier` and `X509TrustManager` are used to validate above conditions respectively but vulnerability in custom implementation of validation methods may lead to MitMo attacks. Listings 2-3 shows code snippet with `HostnameVerifier` vulnerability. `HostnameVerifier`'s `verify()` method always returns `true` as shown in Listing 2 (Line 3) which means even if Hostname does not match CommonName, it returns true. This makes the code vulnerable. In Listing 3, `SSLSocketFactory` create SSL socket but it allows all hostnames (Line 2) through `setHostnameVerifier` API.

```
1 HostnameVerifier allHostValid = new HostnameVerifier(){
2  public boolean verify(String hname, SSLSession s){
3   return true; }};
4 URL url = new URL("https://www.server.com/");
5 HttpsURLConnection con = (HttpsURLConnection)url.
      openConnection();
6 con.setDefaultHostnameVerifier(allHostValid);}
```

Listing 2: Vulnerable HostnameVerifier

```
1 SSLSocketFactory s=new AndroidSSLSocketFactory(Keystore);
2 s.setHostnameVerifier(SSLSocketFactory.
      ALLOW_ALL_HOSTNAME_VERIFIER);
```

Listing 3: Vulnerable HostnameVerifier

Listing 4 shows code snippet with `TrustManager` vulnerability. Here, `SSLSocketFactory` accept all certificates irrespective of its signer as shown by blank overridden `checkServerTrusted()` method without any exception.

```
private static SSLSocketFactory vs() {
 SSLContext c = SSLContext.getInstance("TLS");
```

```
TrustManager tm = new X509TrustManager() {
 @Override
 public void checkServerTrusted(X509Certificate[] chain,
     String authType) {} };
c.init(null, new TrustManager[] { tm }, null);
return c.getSocketFactory();}
```

Listing 4: Vulnerable TrustManager

C. **Vulnerable On-device Storage**

External storage is shared in Android and must not be used for storing any private data of an app. Even encrypted storage is also vulnerable through covert and overt channels.

III. Methodology and Design

Figure 1 shows overall work-flow of framework. sPECTRA is designed to work in two phases. Phase 1 prepares the set of all cryptographic APIs used by app. The app using cryptographic primitives is potentially **sensitive app**. Section III.1 covers the detail of this phase. Phase 2 performs precise vulnerability analysis of only sensitive apps. This phase comprises four major functions: SSL/TLS Vulnerability Identification, App Hooking and Repackaging, Intelligent UI Exploration and Log Parsing. Section III.2 covers the details of Phase 2.

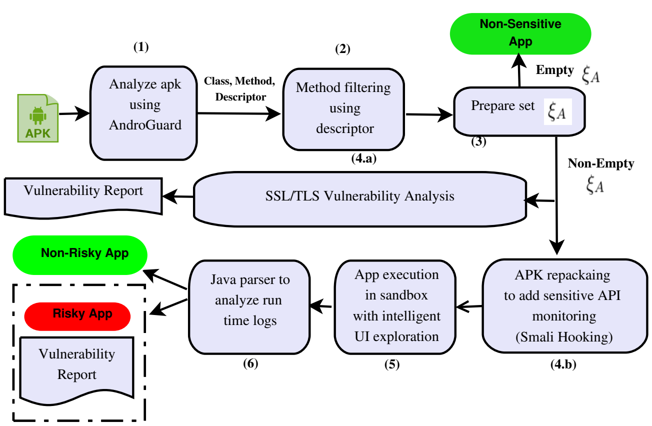

Fig. 1: sPECTRA WorkFlow

1. Phase 1

Implementation of this phase is done using Androguard framework [2]. A **method descriptor** represents the type of parameters that the method takes and the value that it returns. Method descriptors extracted from app using Androguard (`get_descriptors()` utility), are used to filter sensitive apps. The four primary packages, relevant to crypto APIs are `javax.crypto`, `javax.crypto.spec`, `java.security` and `javax.net.ssl`. Therefore, all methods that make use of cryptographic APIs have patterns "crypto", "security" or "ssl", in their descriptors. All methods having any of above descriptor pattern are filtered in Step 2, Figure 1 (using `get_methods()` utility). Next step constructs the set $\xi_\mathbf{A}$ (the set of all cryptographic sensitive APIs used by the app A) using filtered methods (Step 3, Figure 1). It is done by finding actual APIs used in source code, with the help of `get_source()` method of Androguard.

As shown in Equation 1, empty set ξ_A indicates that the app is Non-Sensitive app. App with Non-Empty set ξ_A is marked as a potentially sensitive app.

$$App = \begin{cases} Non-Sensitive, & \text{if } \xi_A = \emptyset \\ Sensitive, & \text{otherwise} \end{cases} \tag{1}$$

Listing 5 shows the result of Phase 1 analysis for Mobikwik app[2]. As shown in result, few of the arguments can be checked for vulnerability directly but for others, vulnerability identification require actual run-time values. Moreover, it is observed that for obfuscated apps, the vulnerable arguments can not be inferred from phase 1. The parameter declaration, definition and API using it may all be distributed over different components. The asynchronous nature and presence of multiple components in Android makes static backward slicing imprecise to find arguments. Therefore, to obtain vulnerabilities precisely, sensitive app is further analyzed by Phase 2. The set ξ_A is input for next phase.

```
//VULNERABLE !! -> ITERATION COUNT 4
new javax.crypto.spec.PBEKeySpec(new StringBuilder().
    append(p7).append(p8).toString().toCharArray(), com.
    mobikwik_new.helper.ab.b, 4, 128);
//VULNERABLE !! -> SIGNING ALGORITHM SHA1withRSA
java.security.Signature.getInstance("SHA1withRSA");
//VULNERABLE !! -> AES in ECB mode
javax.crypto.Cipher.getInstance("AES/ECB/PKCS7Padding");
//MAY BE VULNERABLE !! -> CHECK KEY MUST NOT BE STATIC
javax.crypto.spec.SecretKeySpec(v0.getBytes(), "AES");
//MAY BE VULNERABLE !! -> CHECK PARAMETER
javax.crypto.spec.IvParameterSpec(com.mobikwik.helper.ab.
    a);
```

Listing 5: Phase 1 result for Mobikwik app

2. Phase 2

The details of four major modules of this phase are as follows:

A. **SSL/TLS Certificate Validation**

The module identifies vulnerable implementations of SSL/TLS certificate validation using static analysis. The analysis is implemented on top of Soot library [23]. Soot's tagging feature is employed to tag apps as "Vulnerable" or "Non-Vulnerable". Initially, tags are set to "Non-Vulnerable". The analysis utilizes Soot's Points-To analysis, Control Flow Graph (CFG) and Data-Flow analysis features. Point-To analysis is a static analysis techniques that aims to find objects, a pointer/reference may point during execution of program. For e.g. if `p=&x; p=&y;` then `p` may points to `x` or `y` during execution. Therefore Points-To analysis of `p`, gives set `Points-To(p)` = {`x`,`y`}. sPECTRA first generate intermediate representations (IRs) in form of CFG, Points-To set and then apply following checks on IRs:

a) The module analyze exit nodes in CFG of class containing `HostnameVerifier` interface for return value. Vulnerability is reported if it always returns a `true` value. Listing 6 shows the source code snippet. Whenever `verify` method of `HostnameVerifier` class is called (Line 4), `UnitGraph` is constructed for the class (Line 5). The state

[2]https://play.google.com/store/apps/details?id=com.mobikwik_new&hl=en

is marked as VULNERABLE if all the tails (Line 6) of graph return value of 1 (True) (Line 12).

b) For listing 3, to find whether `HostnameVerifier` is vulnerable or not, Points-To set of `SSLSocketFactory` is calculated. If the set contains `AllowAllHostnameVerifier` then it is marked as vulnerable.

c) sPECTRA mark the absence of an exception in custom implementations of `X509TrustManager` as vulnerable. The absence of exception means not generating alerts in the case of non-validation of signing authority.

In all above cases, before reporting vulnerability, sPECTRA confirms that vulnerable instantiations of `HostnameVerifier` and `TrustManager` are used in any SSL connection using Data-Flow analysis. For e.g. vulnerabilities are reported for Listing 2 and 4 because vulnerable instantiation of `HostnameVerifier` in Listing 2-Line 1 is used at Line 6 and vulnerable `TrustManager` in Listing 4-Line 3 is used at Line 6. The Data-Flow analysis approach helps sPECTRA in addressing **false positives**.

```
1 finalstate = "Non-VULNERABLE"; allstatestrue=1;
2 InvokeExpr exprm = stmt.getInvokeExpr();
3 SootMethod m = exprm.getMethod();
4  if(m.toString().contains("verify")){
5   UnitGraph graph = new BriefUnitGraph(b);
6   for (Unit u : graph.getTails()) {
7    if (u instanceof ReturnStmt) {
8     ReturnStmt rs = (ReturnStmt) u;
9     if (rs.getOp().equals(IntConstant.v(0))) {
10     allstatestrue = 0;
11     break;}}}
12  if (allstatestrue == 1)
13   finalstate = "VULNERABLE";
```

Listing 6: Code to Check HostnameVerifier Vulnerability

B. **App Hooking and Repackaging**

The sensitive app is repackaged to add monitoring code for the API set ξ_A with the help of APIMonitor [4] (Step 4(b), Figure 1). APIMonitor provides the feature of configuring APIs, those need monitoring at run time. The monitoring code logs the run-time parameters and returned values on the invocation of APIs from set ξ_A during execution. The repackaged app is then installed and executed in an fresh emulator.

C. **Intelligent UI Exploration**

In dynamic analysis based system, a critical step in detecting a vulnerability is to generate the vulnerable behavior by simulating the user interaction expected by the app. Android's provides Monkey tool [8] and MonkeyRunner [9] as default exploration tools. However, experiments show their unsuitability for large-scale analysis. A large number of crashes are reported for MonkeyRunner in literature [28]. Also, at the crash, it does not generate any error trace. MonkeyRunner testing needs the object's coordinates to perform the touch and drag actions. Hence, a minor change in location of view will require the test cases to be re-written. It requires the position of view to be known in advance which restrain to use it as generic exploration system. However, sPECTRA is a generalized system which does not depend on object's position on screen. Invalid inputs by Monkey for text based UI elements lead to crash or stop the application.

To address the issues, we propose and develop a more complete and systematic UI Exploration approach. The implementation uses APIs from Robotium framework [3] which provides APIs for writing user interface tests for Android applications. Exploration starts with launcher activity. All views (view is an object that draws something on the screen for user interaction) are divided in four major categories: Input views, Click views, Scrollable views, and Zoom views. Views of current activity are retrieved and maintained in arraylist when the activity is loaded first time. For an already traversed activity, actions (click, zoom, scroll etc.) will be performed on unexplored views in depth-first order. The method continues till all activities/views are covered at least once. Flags are maintained for explored activities/views.

Algorithm 1 covers the approach in more detail. UI Exploration begins by first finding all activities in the app A (app to test) and setting the corresponding flag as unexplored (Line 1). Exploration starts with Launcher Activity (Line 2). It finds all type of views in current activity if this activity is being loaded the first time and set the respective flags as false initially (Lines 7-11). Next, it performs zoom, scroll and click events on respective views in Depth-First order with all input type of views intelligently filled. An input value is provided based on the nearest placeholder of view. E.g., a placeholder with values "Number", "Contact Number", "Mobile", "cell", is given a valid mobile number as input. After performing event on any view, the flag associated with that view is set. A delay is added to load the activity fully (Lines 13-23), and the process is repeated with loaded activity. Once all views of activity are explored, the activity's explored flag is set (Lines 25-26) and ensured that this activity is not considered again (Line 6). App Exploration continues in this case by loading previous activity. It ensures that previous activity is not loaded if the current activity is Launcher Activity (Lines 25-31).

D. **Log Parsing**

The module exhaustively finds the vulnerabilities from the logs collected during app execution in the emulator. The logs collection is made using `logcat` utility of `adb` in parallel of UI Exploration by running both in separate subprocesses. sPECTRA maintains a database of cryptographic APIs along with vulnerable arguments and return values of all APIs of 3 crypto libraries. After the app exploration finishes, Parser module intelligently processes the collected logs. The module identifies vulnerabilities by testing each argument and the return value of the API against the database. If the app uses primitives like salt, key-material then the app exploration and log collection modules are executed second time. From processing both the logs it is ensured that multiple runs of app use the different values.

IV. EVALUATION

sPECTRA's evaluation is done on three fronts:

```
Input  : App A
Output: UI Exploration of App A
Set explored_activity(i)=false for all activities i in A;
Set Launcher Activity as CurrentActivity;
function Explore()
If explored_activity(i) is true for all activities i in A
    Stop App Exploration;
i = Get CurrentActivity with explored_activity(i)==false;
if i is loaded first time then
    I(i)=List of all InputType Views in current Activity;
    C(i)=List of all click, scroll, zoom views in current
    Activity;
    clicked_view(j)=false for all views j in C(i);
end
allviews_explored(i)=true;
for j in all views in C(i) where do
    if clicked_view(j)==false then
        Provide intelligent user-input to views I(i);
        Set clicked_view(j) = true;
        allviews_explored(i)=false;
        Perform a click on view j ;
        if launched_activity <> currentactivity then
            wait to load Activity;
            Explore();
        end
    end
end
if allviews_explored(i)==true then
    set explored_activity(i)=true;
    if i <> launcher_activity then
        Load previous activity using GoBack() utility;
    end
    Explore();
end
end function
```

Algorithm 1: sPECTRA approach for UI Exploration

Vulnerability Coverage sPECTRA covers wide range of vulnerabilities compared to earlier work as shown in Table II.

TABLE II: Comparison with Cryptographic Analysis Frameworks - Covered(✓), Uncovered(✗)

Vulnerability	Framework			
⇓	sPECTRA	CryptoLint [22]	SMV-HUNTER [24]	CMA [27]
Symmetric Encryption (Algo and Mode)	✓	✓	✗	✓
Digital Signature	✓	✗	✗	✗
Padding	✓	✗	✗	✓
Message Digest	✓	✗	✗	✓
Salt	✓	✓	✗	✗
Key-material	✓	✓	✗	✗
Initialization Vector	✓	✓	✗	✓
Seed	✓	✓	✗	✗
Key-Size	✓	✗	✗	✓
Iteration Count	✓	✓	✗	✗
SSL/TLS	✓	✗	✓	✗
On-Device Storage	✓	✗	✗	✗
Features ⇓				
Scalability	✓	✓	✓	✗
Open-Source	✓	✗	✓	✗
Web based service	✓	✗	✗	✗

Code Coverage Android's default provided Monkey and MonkeyRunner are not suitable for large-scale analysis due to the reasons covered in Section III.2.C. sPECTRA's UI automation module improves code coverage, when compared to the monkey tool due to the use of complete and deterministic exploration. EMMA [6], a code coverage measurement tool is used to obtain code coverage of both sPECTRA's Intelligent UI Exploration module and Android's Monkey. EMMA requires source code to find code coverage of Monkey. Therefore, we downloaded 40 apps belonging to various categories from F-Droid[3] and modified them by adding code coverage code. EMMA generates % code coverage in terms of Class, Method, Block and Line. Figure 2 shows the mean coverage (Class, Method, Block and Line) for both sPECTRA and Monkey for 14 representative apps (out of 40 measured). In the experiments, the Monkey is set to execute 5000 events that is quite a large number. Results for all 40 apps confirm that sPECTRA performs better than Monkey. This is attributed to following reasons:

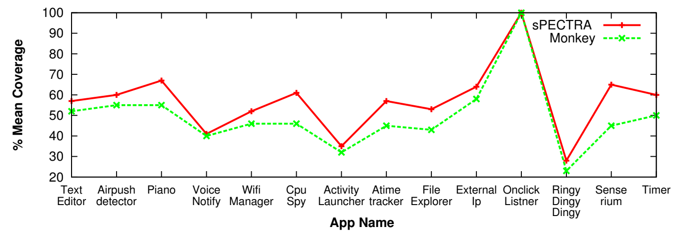

Fig. 2: UI Exploration Comparison

1) sPECTRA includes context aware input generation for TextViews. A set of predefined inputs is maintained for different placeholders. E.g., a placeholder with values "Number," "Contact Number," etc. is given a valid mobile number as input. In this way, sPECTRA address the problem of Monkey which terminates the app on invalid input.
2) sPECTRA handles advanced UI elements like swipes, Long Press, tabs, spinners, etc. which are missing in Monkey.
3) The systematic handling of explored and unexplored views in each activity makes sPECTRA more complete system for code coverage.

Not only code coverage but sPECTRA also improves over Monkey and MonkeyRunner in other regards as mentioned in Table III.

TABLE III: Comparison with Android's default UI frameworks

Property	sPECTRA	Monkey	Property	sPECTRA	MonkeyRunner
Repeatable Events	✗	✓	Crash Handling	✓	✗
Intelligent Text Input	✓	✗	Scalability	✓	✗
Effective Exploration	✓	✗	Coordinate based UI Interaction	✗	✓
Code Coverage	High	Low			

False Negatives SMV-HUNTER [24] propose a hybrid approach for detection of SSL/TLS vulnerabilities where static analysis first marks the app as vulnerable if overriding of default validation methods is done. Then the dynamic phase performs actual Man-in-the-middle (MITM) attack for the marked app to confirm the vulnerability. But, due to reported crashes of MITM proxies in processing large number of requests, sPECTRA develop static analysis approach as detailed in Section III.2.A. For HostnameVerifier vulnerability, sPECTRA considers both the cases of Listing 2 and 3 while SMV-HUNTER only considers the case of Listing 3. This

[3]https://f-droid.org/repository/browse/

leads to considerable number of false negatives by SMV-HUNTER. The popular playstore apps like BuzzWidget, SMS Blocker, OneDrive are found to be vulnerable by sPECTRA while SMV-HUNTER does not report the same.

In dynamic analysis based approach, a critical step in reducing false negatives is to trigger the vulnerable behavior by simulating the user interaction that leads to vulnerability. sPECTRA's handling of advance views like tabs, long presses, spinners, valid inputs for textviews reduces the false negatives.

V. Results

Table IV shows the market-wise statistics of analyzed apps. We use Google-Play citegcrawler and Third-Party crawlers [10] to collect the samples for analysis. Out of these apps, 107 apps failed during analysis. Some of the apps failed as Soot was not able to analyze them, some failed during repackaging and some during exploration (not compatible with emulator).

TABLE IV: Sample's Statistics

Domain/Market-Name	#Apps	Domain/Market-Name	#Apps
google playstore	800	gfan	6000
nduoa	700	androidpur	366
mobomarket	58	appsapk	82
apkfun	31		

Vulnerability report (partial) for mobikwik app after Phase 2 is as shown in Listing 7. The Listing shows that same salt (Vulnerable 1), same key-material (Vulnerable 2) and same IV (Vulnerable 3) is used in two runs. Iteration count value is only 4 as shown in vulnerability 1. `HostnameVerifier` vulnerability is also present. It uses `AES` algorithm in `ECB` mode and signing algorithm is `SHA1withRSA` (Phase 1 report).

```
//Vulnerable 1 -> Same Password and Salt is used in
      multiple (4) runs. Low Iteration Count
PBEKeySpec;-><init> : 4
A1 (Password) {c, o, m, ., m, o, b, i, k, w, i, k, _, n,
      e, w, j, y, o, t, i, ., g, a, j, r, a, n, i, @, m, n,
      i, t, ., a, c, ., i, n
A2 (Salt) {-46, 90, 68, -128, -103, 57, -74, -64, 51, 88,
      -95, -45, 77, -117, -36, -113, -11, 32, -64, 89}
A3 (Iteration-Count) { I=4 }
//Vulnerable 2 -> Same Key material used in 4 instances
      for generation of secret key
SecretKeySpec;-><init> : 4
([B={-52, 51, -68, -121, -44, -114, -59, -20, -79, 22,
      34, -77, -48, -75, 45, 93}, "AES")
//Vulnerable 3 -> Same Initialization Vector 2 times
IvParameterSpec;-><init> : 2
([B={16, 74, 71, -80, 32, 101, -47, 72, 117, -14, 0, -29,
      70, 65, -12, 74})
//VULNERABLE 4 -> HostnameVerifier always returning true
Class Path : org/apache/cordova/filetransfer
final class FileTransfer$2  implements HostnameVerifier{
public boolean verify(String paramString, SSLSession
      paramSSLSession) {return true; }}
```

Listing 7: Precise Report of Analysis for Mobikwik App

Table V shows detailed results for some of the highest downloaded apps of Google-playstore from various categories. The name of banking app is not disclosed for privacy reasons. The analysis shows that even the most popular Android apps are vulnerable due to improper use of cryptographic primitives.

Out of 7000 apps analyzed, 4529 apps are found to be using cryptographic primitives. Out of these 4529 apps, 3877 apps are vulnerable with at least one factor. Figure 3 shows

TABLE V: Analysis Results

App/Category/Download in 10^6	Vulnerabilities Identified
Banking App/Banking/1-5	MessageDigest algorithm is SHA-1
	Static Key Material for Secret-Key Generation
	AES in ECB Mode
	Signature Algorithm is SHA1withRSA
	HostnameVerifier always return TRUE
	TrustManager does not throw any exception
Mobikwik/LifeStyle/10-50	Iteration Count in PBE is 4
	Static Key Material for Secret-Key Generation
	Static Salt
	MessageDigest algorithm is MD5
	Initialization vector in AES/CBC mode is static
	AES in ECB mode
	Signature Algorithm is SHA1withRSA
	SSL TrustManager Vulnerability
Password Notes/Tools/.5-1	Iteration Count in PBE is 100
	Static Salt
	AES in ECB mode
	Key Length (32 bit) in PBE
TaxiForSure/Transport/1-5	AES in ECB mode
	MessageDigest algorithm is SHA-1
	SSL TrustManager vulnerable
Snapdeal/Shopping/10-50	AES in ECB mode
Signal/Communication/1-5	Iteration Count in PBE is 100

the number of apps (out of 7000 analyzed) using the specific cryptographic primitive and out of those apps, the apps which are vulnerable. The results indicate that nearly 90% (out of using crypto) of apps are vulnerable. More than 50% of the apps are vulnerable to Hash collision attacks due to signing algorithm vulnerability. Still, these apps use `SHA1WithRSA` as the signature algorithm that is declared vulnerable [13]. 75% of apps still use DES encryption algorithm having the 56-bit key length that makes them vulnerable to brute-force attacks. Moreover, the vulnerable apps also belong to critical categories like banking, finance, shopping, or education.

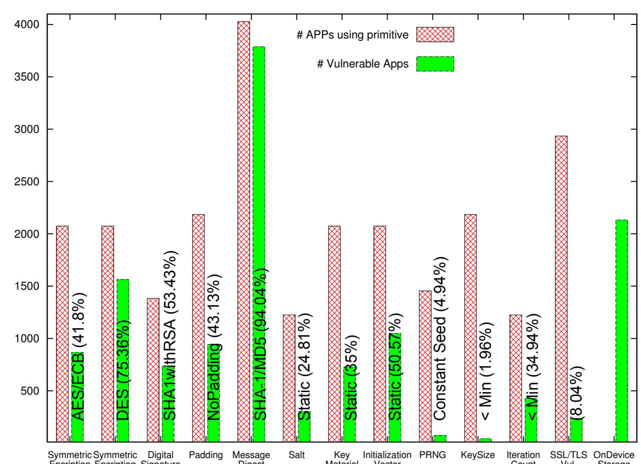

Fig. 3: Vulnerability by Primitive

VI. Related Work

Static Analysis Egele et al. developed CryptoLint to statically analyze cryptographic vulnerabilities based on misuse of symmetric and PBE in Google playstore apps [22]. Additionally, sPECTRA analyze other vulnerabilities as detailed in Table II. The critical observations show that CryptoLint being the pure static approach, may not determine vulnerability for obfuscated, run-time dependent or logical condition based parameters. Moreover, the security prerequisites put necessity on using non-unique and non-predictable values for critical security parameters like Initialization Vector, Salts, etc. Static analysis can only infer that these critical parameters are derived from static components or not while sPECTRA reports the vulnerability on the use of same values for critical parameters in different executions. The comparison of results of sPECTRA is not done with CryptoLint due to it's unavailability.

Hybrid Analysis CMA uses hybrid approach for analysis [27]. However, the approach relies on **manual analysis** which limits it's scalability for large app stores. Specifically, the aim of sPECTRA is to enable the automatic large scale analysis. Moreover, sPECTRA covers the wider range of vulnerabilities compared to CMA as shown in Table II. Overall, sPECTRA's approach makes it a **lightweight framework** compared to CMA. Mauro et al. proposed a light weight, system MITHYS for protecting against SSL vulnerabilities [20]. Steven et al. propose OpenCCE system which provides developers with cryptographically correct code blueprint based on their requirement [16]. However, our focus is to verify the apps after development.

Automated Analysis Dynodroid [25] is an automatic input generation system that instruments the Android SDK for capturing system events. On average, it achieves a code coverage of 55%. Dynodroid's results show that monkey also performs comparable code coverage, but Monkey requires nearly 20X more input events for same code coverage. However, Dynodroid is only supported for Android version 2.3 while sPECTRA is tested till version 5.1.1. Appsplayground's [26] UI exploration is closely related to sPECTRA. However, it works on modified Android software stack while sPECTRA works on unmodified software stack. This restricts Appsplayground current implementation applicable only for single API level. The critical issue of emulator fails in loading snapshot with exception "**savevm: unable to load section RAM**" is observed during experiments with Appsplayground.

A^3E [17] constructs a high-level CFG that captures legal transitions among activities (app screens). This graph is then used to develop an exploration strategy. The time of exploration modules are more than hour which is very high and create the problem in scaling. It does not handle multi-touch gestures such as pinching and zooming and only tested for Android version 2.3.4.

VII. Conclusions

sPECTRA is an automated and lightweight framework to precisely analyze cryptographic vulnerabilities in Android apps at large scale. The aim is to prevent exploitation of user's private information by Android apps. Our results show that even popular apps available at Google playstore are vulnerable to cryptographic attacks. The important feature of sPECTRA is that it does not require any root access, source code, and works without any alteration to Android source code. sPECTRA also works for obscured/obfuscated apps due to run-time analysis. sPECTRA currently includes analysis for 12 categories of vulnerabilities. We are working on analyzing more vulnerabilities. sPECTRA is currently unable to explore custom views, and system events. We are also working on these to increase code coverage.

References

[1] http://www.gartner.com/newsroom/id/3323017.

[2] Androguard. https://code.google.com/p/androguard/.

[3] Android user interface testing with Robotium. http://www.vogella.com/tutorials/Robotium/article.html.

[4] APIMonitor. https://code.google.com/p/droidbox/wiki/APIMonitor.

[5] BLOCK CIPHERS: Approved Algorithms. http://csrc.nist.gov/groups/ST/toolkit/block_ciphers.html.

[6] EMMA: a free Java code coverage tool. http://emma.sourceforge.net/.

[7] Java Cryptography Architecture(JCA). http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.

[8] Monkey. http://developer.android.com/tools/help/monkey.html.

[9] monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[10] Mssun. https://github.com/mssun/android-apps-crawler.

[11] Password-Based Cryptography. http://www.rfc-base.org/txt/rfc-2898.txt.

[12] Spongy Castle. https://rtyley.github.io/spongycastle/.

[13] The Hacker News : Security in a serious way. http://thehackernews.com/2014/02/98-of-ssl-enabled-websites-still-using.html.

[14] The Legion of the Bouncy Castle. https://www.bouncycastle.org/.

[15] Vulnerability Note. http://www.kb.cert.org/vuls/id/836068.

[16] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 2015.

[17] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices*, volume 48, pages 641–660. ACM, 2013.

[18] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 800:131A, 2011.

[19] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique cryptanalysis of the full aes. In *Advances in Cryptology–ASIACRYPT 2011*, pages 344–371. Springer, 2011.

[20] Mauro Conti, Nicola Dragoni, and Sebastiano Gottardo. Mithys: Mind the hand you shake-protecting mobile devices from ssl usage vulnerabilities. In *Security and Trust Management*. Springer, 2013.

[21] Morris Dworkin. Recommendation for block cipher modes of operation. methods and techniques. Technical report, DTIC Document, 2001.

[22] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[23] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark*, 2008.

[24] David Sounthiraraj Justin Sahs Garret Greenwood and Zhiqiang Lin Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. 2014.

[25] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[26] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[27] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, pages 75–80. IEEE, 2014.

[28] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153, 2015.