Orchestration or Automation: Authentication Flaw Detection in Android Apps

Siqi Ma, Juanru Li, Surya Nepal, Diethelm Ostry, David Lo, Sanjay Jha *Member, IEEE,* Robert H. Deng *Fellow, IEEE,* and Elisa Bertino *Fellow, IEEE*

Abstract—Passwords are pervasively used to authenticate users' identities in mobile apps. To secure passwords against attacks, protection is applied to the password authentication protocol (PAP). The implementation of the protection scheme becomes an important factor in protecting PAP against attacks. We focus on two basic protection in Android, i.e., SSL/TLS-based PAP and timestamp-based PAP. Previously, we proposed an automated tool, GLACIATE, to detect authentication flaws. We were curious whether orchestration (i.e., involving manual-effort) works better than automation. To answer this question, we propose an orchestrated approach, AUTHEXPLOIT in this paper and compare its effectiveness GLACIATE. We study requirements for correct implementation of PAP and then apply GLACIATE to identify protection enhancements automatically. Through dependency analysis, GLACIATE matches the implementations against the abstracted flaws to recognise defective apps. To evaluate AUTHEXPLOIT, we collected 1,200 Android apps from Google Play. We compared AUTHEXPLOIT with the automation tool, GLACIATE, and two other orchestration tools, MalloDroid and SMV-Hunter. The results demonstrated that orchestration tools detect flaws more precisely although the F1 score of GLACIATE is higher than AUTHEXPLOIT. Further analysis of the results reveals that highly popular apps and e-commerce apps are not more secure than other apps.

Index Terms—Vulnerability Detection, Password Authentication, Mobile Security

1 INTRODUCTION

Mobile devices have become ubiquitous in our lives, and a large number of apps have been developed for mobile devices to support a wide range of domains such as banking, social networking and transportation. We have grown so accustomed to using apps that it is hard to imagine daily life without them. Hence, for our security it is essential to assess the implementation of these apps.

In this paper, we analyze the implementations of password authentication protocols in mobile apps for the following specific reasons. First, most mobile apps use online services (e.g., Facebook and twitter) which employ user authentication as the first line of defense in protecting users' private data (e.g., account details, interaction history, private messages). Second, user authentication has been regarded as one of the weakest links in security. Vulnerabilities have been identified in authentication schemes used in non-mobile online apps, and we believe that similar vulnerabilities exist in mobile apps. Our analysis focuses on Android apps as Android is the most widely used mobile app platform, running on more than 80% of mobile phones and tablets [1].

- *Siqi Ma, Surya Nepal, and Diethelm Ostry are with Data61, CSIRO, Australia.*
  E-mail: {siqi.ma, surya.nepal, diet.ostry}@csiro.au
- *Juanru Li is work with Shanghai Jiaotong University, Shanghai, China. Email: roman@sjtusec.com*
- *Robert H. Deng and David Lo are with Singapore Management University, Singapore. Email: {robertdeng, davidlo}@smu.edu.sg*
- *Sanjay Jha is with University of New South Wales, Sydney, Australia. Email: sanjay.jha@unsw.edu.au*
- *Elisa Bertino is with Purdue University, West Lafayette, USA. Email: bertino@purdue.edu*

Passwords have been the main user authentication method since the advent of computers. Due to the advantages of using a password, such as it being simple and inexpensive to create, use and revoke, most developers typically use a password authentication protocol [2] in mobile apps. In the basic password authentication protocol (BPAP), a user sends a combination of username and password in plaintext to a server through a client app. The server replies with an authentication-acknowledgement if the received password is valid. However, BPAP is vulnerable to eavesdropping and replay attacks. To protect against such attacks, developers commonly use two countermeasures: 1) Secure Socket Layer (SSL) / Transport Layer Security (TLS)-based password authentication [3]; and 2) nonce-based password authentication protocols [4]. In SSL/TLS-based password authentication, SSL first authenticates the server and sets up a secure connection between the client and the server [5]. Then the client sends the username and password over the secure connection in order to authenticate the user to the server. For nonce-based password authentication, a client uses the user's password as a secret key to compute a cryptographic function on a nonce value. It is crucial that the nonce value is not predictable. Nonce-based password authentication protocols are of two types: challenge-response password authentication protocol, in which the nonce value is a random number generated by a server, and timestamp-based password authentication protocol, in which the nonce value is the current time at a client.

Although various protection schemes have been proposed to secure these password-based authentication protocols, a variety of attacks have been devised to intercept/modify passwords ([3], [6], [7], [8]). The password interception attack [3] targets the cryptographic algorithm implemented in the SSL/TLS protocol. Attackers attempt to break block ciphers in the Cipher Block Chaining (CBC) mode implemented in the SSL/TLS protocol. Passwords of Internet Message Access Protocol (IMAP) accounts are then intercepted. Password stealing and reuse attacks [7] are crafted to attack two-factor and three-factor authentications. Since users often use simple passwords and reuse the same passwords for different accounts, attackers perform phishing attacks or install keyloggers to steal the passwords. Various techniques have been proposed to counter these attacks, such as deploying more sophisticated protocols [9], using an additional software or hardware infrastructure [10], and designing stronger cryptographic algorithms [11].

Existing tools that have been developed for similar goals in other ways, such as detecting vulnerabilities in SSL/TLS implementations, suffer from several drawbacks: (a) lack of accuracy due to the analysis of limited features (e.g., only network APIs), and inability to extract inter-components methods [12]; (b) inability to identify the portion of the app code where the password authentication protocol is implemented, and how to locate all password authentication protocols that are implemented in an app (for apps that use multiple protocols) [13]; (c) inability to recognize new implementation defects because only known attacks are used to identify a vulnerability [14].

In this paper, we are not concerned with designing secure protocols or cryptographic algorithms for password-based authentication. Rather, our goal is to carry out an analysis of existing Android apps to identify implementation defects in their password authentication protocols and we assume that the design of the protocols is secure.

By extracting the requirements for implementing each password authentication protocol securely from official documents [15] and websites[1], we summarize three common types of implementation flaws: *Flaw 1* - plain password transmission; *Flaw 2* - insufficient SSL/TLS validation; *Flaw 3* - incomplete timestamp format. *Flaw 1* makes the authentication protocol vulnerable to password eavesdropping and replay attacks. *Flaw 2* makes the authentication protocol vulnerable to server impersonation and man-in-the-middle (MITM) attacks. In SSL/TLS-based password authentication protocol, the client app is required to verify the validation of the server's certificate and check its hostname to ensure that the client is communicating with the correct server. Finally, *Flaw 3* makes the authentication protocol vulnerable to replay attacks.

To identify these flaws, we have designed and implemented a fully-automated , GLACIATE [16], to assess the implementation of password authentication protocols.

In automation approach GLACIATE, the templates used to detect authentication flaws are learnt by a machine learning algorithm. Such a detection scheme reduces the manual effort of summarizing general patterns. However, an interesting issue is whether machine learning techniques are "smarter" than humans in generating such patterns. Therefore, we designed an orchestration tool, AUTHEXPLOIT, to identify authentication flaws from app implementations. In AUTHEXPLOIT, the authentication flaw patterns are first identified manually, and then apply the automated analysis to detect the flow. The automation part in AUTHEXPLOIT has two components: *App Preprocessing* and *Flaw Detection*. As first step, AUTHEXPLOIT converts each Android app into an intermediate representation (i.e., Jimple code in our paper) and creates its super call graph by applying intra- and inter-procedural analysis. It then performs flaw detection by using the super call graph to further statically analyze the dependencies among functions and variables. AUTHEXPLOIT finally identifies whether the password authentication protocol implemented in an app matches any of the three implementation flaws listed above which are identified and coded manually.

In order to assess the effectiveness of AUTHEXPLOIT, we created a ground–truth dataset by manually analyzing 1,200 Android apps with respect to the three flaws discussed above. we also compared the results of AUTHEXPLOIT and GLACIATE with other two orchestration tools (where some form of manual efforts are involved), which are two state-of-the-art SSL/TLS certificate validation tools, MalloDroid [12] and SMV-Hunter [17], against our dataset. AUTHEXPLOIT performs better than both MalloDroid and SMV-Hunter by successfully identifying 627 authentication flaws. Although GLACIATE has a lower Recall value than AUTHEXPLOIT, AUTHEXPLOIT detects flaws more accurate with the Precision value of 93.9%.

Contributions: Overall, our contributions are

- We manually extract the requirements for correct implementations of secure password authentication protocols from official documents and secure apps. For two commonly used protocols, i.e., SSL/TLS-based password authentication protocol and timestamp-based password authentication protocol, we identify three common flaws that may affect app security.
- We develop an orchestrated detection tool, AUTHEXPLOIT, that aim to detect authentication flaws. We then compare the effectiveness of our tool with two state-of-the-art orchestration tools, and demonstrate that our tool performs better than them. Finally,we compare the effectiveness of AUTHEXPLOIT with automation tool, GLACIATE, to check whether orchestration approach is better than automation approach.
- We have carried out several analyses to investigate the correlations between implementation flaws and different app features (e.g,. popularity and domain categories).

Organization: The rest of this paper is organized as follows. Section 2.3 introduces background information of the secure password authentication protocols used in Android and their correct implementations. We illustrate the challenges of examining authentication in an app in Section 3.2. In Section 4.2.3, we introduce AUTHEXPLOIT in detail. In Section 5.3, we evaluate the effectiveness of AUTHEXPLOIT by applying it to real-world Android apps and investigate the widespread authentication implementation. We discuss related works in Section 6.3. Section 7 concludes the paper and outlines directions for future work.

1. System Approach: https://book.systemsapproach.org/security/authentication.html

2 Background

In this section, we present the relevant background about the techniques used for identifying vulnerable implementations of password authentication protocols in Android apps. First, we introduce the most commonly used program analysis techniques for static analysis in Section 2.1. We then present the two most commonly used password authentication protocols and describe their security requirements in Section 2.2. Finally, in Section 2.3 we list three common implementation flaws and describe how they are exploited by attackers.

2.1 Static Program Analysis

Different from dynamic program analysis, static program analysis identifies flaws in a program without running the program. Therefore, static program analysis locates the exact code snippet with flaws, and also examines code that is rarely reached during execution. Generally, control flow analysis and data flow analysis are applied to extract the program execution process. Details are elaborated below.

Control Flow Analysis. Control flow analysis [18] is a technique for determining the execution order of program statements. The control flow is represented as a control flow graph (CFG). In a CFG, an entry block and an exit block are defined, representing the entry and exit of an execution, respectively. Nodes in the CFG represent a number of basic blocks, which are a linear sequences of program instructions. A CFG also includes a number of basic blocks which are linear sequences of program instructions. An edge between blocks b and b' indicates that the execution flows from b to b'. Therefore edges in a CFG represent all possible execution paths in the program including infeasible paths and dead code. The set of execution traces can further be predicted and specified via the CFG.

Data Flow Analysis. From the created CFG, data flow analysis is used to trace variable changes when the program is executed. Data flow analysis gathers information about the variables of interests and then traces their possible locations in a program. From each basic block in the CFG, data flow analysis derives the effects of each operation, and its corresponding inputs and outputs. This analysis also optimizes the CFG by removing the infeasible paths and dead code, because variables in these statements are not data dependent on any other variables.

2.2 Password Authentication Protocols

A simple password authentication protocol is a weak but convenient authentication scheme. In the basic password authentication protocol (BPAP), a client sends a combination of username and password in plaintext to an authentication server and the server accepts the user only if the password is valid. BPAP is efficient in that it requires only one message from the client to the server without requiring the execution of cryptographic operations. However, BPAP was originally intended for users to be authenticated to a local computer or to a remote server over a closed network; the protocol is vulnerable to eavesdropping and replay attacks when it is used over an open network [7], [8], [19].

Two general approaches are proposed to protect BPAP against attacks: SSL/TLS-based password authentication protocol and nonce-based password authentication protocol.

SSL/TLS-based password authentication protocol. In this protocol, a secure channel is built by using SSL/TLS to protect the communication between the client and the server. All transmitted messages through the secure channel are encrypted, including the username and password of each client.

To build a secure connection, the following verification steps **MUST** be executed:

1) A client sends a message to the server to initiate SSL/TLS communication.
2) The server then sends back a public key certificate.
3) After receiving the certificate from the server, the client verifies whether the certificate is valid. If the certificate is valid, the client creates and sends an encrypted key back to the server. Otherwise, the communication fails.
4) The server decrypts the key and delivers a digitally signed acknowledgement to start an SSL/TLS encrypted connection.

In the verification steps, verifying the public key certificate received from the server is the most essential step. A valid certificate is required to be signed by a Certificate Authority (CA). The client **MUST** validate each certificate by checking:

1) Whether the `subjectAltname` field and the host portion of the server's URL *match* [12];
2) Whether the CA that signed the certificate is *trusted* [20];
3) Whether the signature of the certificate is *unexpired* [21].

Nonce-based password authentication protocol. Another mitigation to counter password eavesdropping and replay attacks [22] is the nonce-based password authentication protocol, which relies on an arbitrary number (i.e., a nonce). The nonce is only used once in a cryptographic communication, i.e., the communication between the client and the server in our paper. According to the particular approach for generating the nonce, the protocol is classified as challenge-response or timestamp-based.

For the challenge-response protocol, a server first generates a nonce by using a pseudo-random number generator and sends it to the client as a challenge. The client then uses his/her own password as a secret key to encrypt the nonce and sends the ciphertext to the server for identity verification. Besides using a pseudo-random number, using a timestamp as a nonce (i.e., the timestamp-based password authentication protocol) is another way to label each unique message.

The construction of a secure nonce-based password authentication protocol **MUST** comply with the following requirements:

1) The pseudo-random number used for the challenge-response password authentication protocol should be *unpredictable*.

2) The timestamp involved in the timestamp-based password authentication protocol should have the format of *Year/Month/Day/Hour/Minute/Second*.

Note that we cannot determine the randomness of the pseudo-random number generated by the server without access to the source code; thus we only consider implementation flaws in the timestamp-based password authentication protocol in this paper.

2.3 Flaws in Protocol Implementations

A password authentication protocol is secure only if it complies with the above mentioned requirements. However, implementing a password authentication protocol correctly is not easy. We discovered the following flaws that make implementations non compliant with the above requirements.

***Flaw 1*: Plain password transmission.** This flaw is an implementation of BPAP over an insecure network without any protection, i.e. users' passwords are transmitted in plaintext. It is vulnerable to eavesdropping and replay attacks.

***Flaw 2*: Insufficient SSL/TLS validation.** This is a common flaw in the implementations of secure password authentication protocols. Unless all the validation requirements listed in Section 2.2 are satisfied, Android apps may trust invalid certificates (i.e., certificates signed by untrusted CAs, self-signed certificates, and expired certificates) or accept invalid hostnames. This allows an attacker to connect to apps by providing a forged certificate to steal users' passwords. Additionally, a malicious counterfeit server can connect with an app by using someone else's valid certificate if the app accepts all hostnames.

***Flaw 3*: Incomplete timestamp format.** This flaw makes the timestamp used for authentication repeatable. For example, a timestamp in the format of "Minute/Second" allows the protocol message to be repeated every hour with the same minute and second.

3 Example

In this section, we first illustrate an implementation defect by using a running example of code with *Flaw 2* (i.e., insufficient SSL/TLS validation), and then highlight challenges in detecting flaws in apps.

3.1 Code Example

For illustration purpose, we consider an example from a highly popular communication app with ratings $\geq$ 4.5 and downloaded $\geq$ 100,000 times. The example (in Jimple code) given in Listing 1 is the code that allows the app to connect with any server without verifying the hostname.

The example includes three functions in two classes: `trustAllHosts` (lines 10-26) and `getTrustedVerifier` (lines 27-39) defined in `class1`, and `verify` (line 51-60) defined in `class2`. Function `setHostnameVerifier` in line 23 takes as input of a hostname, which is assigned in line 22. To make a secure connection, the assigned hostname must be verified by function `getTrustedVerifier`. Although the validation function `verify` is invoked, it is set to return 1 (TRUE) without executing the hostname verification, which suffers from *Flaw 2*.

Listing 1. An Example of *Flaw 2*

```
1   // An Example of Flaw 2: Insufficient SSL/TLS
        validation
2   public class class1 extends java.lang.Object
3   {
4    public void <init>()
5    {
6      class1 $r0;
7      $r0 := @this: class1;
8      specialinvoke $r0.<java.lang.Object: void <init>()
          >();
9    }
10   public class1 trustAllHosts ()
11   {
12     class1 $r0;
13     java.net.HttpURLConnection $r1;
14     boolean $z0;
15     javax.net.ssl.HttpsURLConnection $r2;
16     javax.net. sll .HostnameVerifier $r3;
17     $r0 := @this: class1;
18     $r1 = virtualinvoke $r0.<class1:java.net.
          HttpURLConnection getConnection()>();
19     $z0 = $r1 instanceof javax.net.ssl.
          HttpsURLConnection;
20      if $z0 == 0 goto label1;
21     $r2 = (javax.net.ssl.HttpsURLConnection) $r1;
22     $r3 = staticinvoke <class1: javax.net.ssl.
          HostnameVerifier getTrustedVerifier()>();
23      virtualinvoke $r2.<javax.net.ssl.
          HttpsURLConnection: void
          setHostnameVerifier(javax.net.ssl.
          HostnameVerifier)>($r3);
24    label1:
25     return $r0;
26   }
27   private static javax.net.ssl.HostnameVerifier
        getTrustedVerifier()
28   {
29     javax.net.ssl.HostnameVerifier $r0;
30     class2 $r1;
31     $r0 = <class1: javax.net.ssl.HostnameVerifier
          TRUSTED_VERIFIER>;
32      if $r0 != null goto label1;
33     $r1 = new class2;
34     specialinvoke $r1.<class2: void <init>()>();
35     <class1: javax.net.ssl.HostnameVerifier
          TRUSTED_VERIFIER> = $r1;
36    label1:
37     $r0 = <class1: javax.net.ssl.HostnameVerifier
          TRUSTED_VERIFIER>;
38     return $r0;
39   }
40  }
41
42  final class class2 extends java.lang.Object
        implements javax.net.ssl.HostnameVerifier
43  {
44   void <init>()
45   {
46     class2 $r0;
47     $r0 := @this: class2;
48     specialinvoke $r0.<java.lang.Object: void <init>()
          >();
49     return;
50   }
51   public boolean verify(java.lang.String, javax.net.ssl
        .SSLSession)
    {
      class2 $r0;
      java.lang.String $r1;
      javax.net.ssl.SSLSession $r2;
      $r0 := @this: class2;
      $r1 := @parameter0: java.lang.String;
      $r2 := @parameter1: javax.net.ssl.SSLSession;
      return 1;
    }
  }
```

3.2 Challenges

Our detailed analysis of the flaw in Listing 1 clearly shows the major challenges in the systematic identification of flaws from the implementation:

Challenge 1: How to match specific implementation flaws against code? In theory, the design of a password authentication protocol is simple and well-defined; however, in practice, implementations are complicated. As developers have various coding styles, the identification of code fragments that implement password authentication protocols is challenging. In order to check whether there are any flaws (described in Section 2.3) in an implementation, we need to convert each flaw into a flexible intermediate representative template that can be applied to different coding styles, while at the same time returning few/no false positives.

Challenge 2: How to efficiently and accurately identify the implementation flaws? To detect an implementation flaw, we must analyze whether the implementation of the password authentication protocol suffers from any of proposed the three flaws. As the related implementation code is a code snippet, extracting all program dependencies structure of an app is quite expensive, and in addition most of those structures are not relevant to our analysis[2].

Challenge 3: How to extract a complete protection scheme for the password authentication protocol? Just analyzing the code snippet of password authentication is insufficient for assessing whether the protocol is secure. A secure password authentication protocol is supported by at least one protection scheme that protects either the connection between the server and the client by using SSL/TLS or the transmitted password by using cryptographic primitives. How to identify the implemented scheme(s) used to support the password authentication protocol is equally important.

4 Design

For GLACIATE, it uses a machine learning approach to learn flaw patterns from vulnerable apps. However, developers usually name the customized functions variously. It is imprecise by using the machine learning approach to determine whether a function is relevant to authentication. To address the issue of GLACIATE, we construct an orchestrated approach, AUTHEXPLOIT, to by using expert predefined templates (human expert is involved). Having the defined templates, it takes the following steps to automatically detect implementation flaws in Android apps (see Figure 1).

- **Step 1: Select BPAP.** This checks whether an app has implemented the BPAP.
- **Step 2: Recognize protection schemes.** For the apps using BPAP, this determines what protection schemes are implemented to protect BPAP.
- **Step 3: Discover flaws.** According to the implementation requirements of each protection scheme, this step matches the known requirements with the implementations.

4.1 App Preprocessing

The preprocessing operation is organized in several steps described below.

4.1.1 App Decompilation.

AUTHEXPLOIT is built on top of Soot [23] and works directly on Dalvik bytecode. Applied to Android apps, it translates lowlevel Android bytecode into its intermediate representation (IR), namely, Jimple code. In Jimple code [24], Soot represents low-level code as `Scene`, `SootClass`, `SootMethod`, `SootField`, and `Body`.

4.1.2 Super Call Graph Generation.

From the Jimple code, AUTHEXPLOIT creates a super call graph for each app. Each node in the super call graph is a function, represented in the format {*Statement*, *Method Name*, *Class Name*}. The statement is the current line of statement. The method name and the class name separately represent the `SootMehod` and the `SootClass` where the statement belongs to. A directed edge between two nodes represent three types of relationships: function calls, control flows, and data flows.

AUTHEXPLOIT takes the following steps to construct a super control flow graph.

1) It builds a control flow graph (CFG) for each `SootMethod`. Following the execution sequence, it then conducts an intra-procedural analysis [25] and connects two statements by adding directed edges.
2) It extracts correlations among functions by identifying function calls in each `SootMethod`. According to each function call, it adds an edge directed from the caller to the callee.
3) It collects data flows by analyzing the CFG built in Step 1). Starting from the input parameters and declared variables in each `SootMethod`, it extracts data dependencies between two nodes. A data flow edge is created between two nodes if a node is data dependent on another node; the edge is labeled with the correlated parameter/variable. The edge represents that the value of an input parameter or a variable flows from the parent node to the child node.
4) It constructs a super call graph by combining the edges generated in Steps 1), 2), and 3).

Note that dead code may have been included in the super call graph as an independent graph. Therefore, we

2. On average, if the size of an application is larger than 10K, the number of Jimple code files is more than 10,000. If the size is 5k - 10k, the number of Jimple code files is around 6,000.

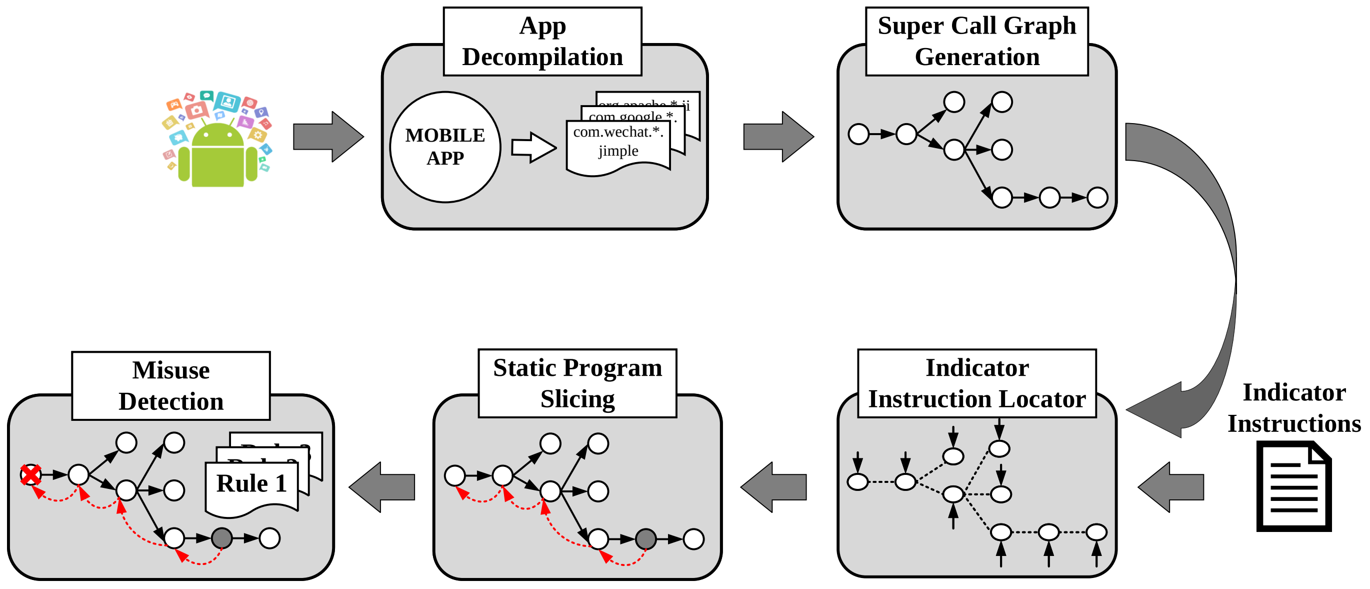

Fig. 1. Overview of AUTHEXPLOIT

remove these independent graphs to improve the efficiency of AUTHEXPLOIT because they are irrelevant to any other code snippets.

4.2 Flaw Location

By taking as input the Jimple code of each app and the constructed super call graph, AUTHEXPLOIT outputs a detection report including the type of flaw and its location. First, AUTHEXPLOIT identifies whether the BPAP is implemented in an app. It then determines the protection scheme designed for the BPAP. It finally checks whether the implemented protection scheme complies with the security requirements.

4.2.1 Indicator Instruction Locator

Instead of creating flaw patterns automatically, AUTHEXPLOIT requires that some manual-observed *indicator instructions* to locate the BPAP code and the relevant protection schemes. An indicator instruction is a function that is related to the implementation of the password authentication protocol in Android. If any of these indicator instructions is invoked in an app, it is then labeled as BPAP-related.

To collect the APIs that **MUST** be invoked in the implementation of password authentication protocols, we manually learn the correct implementations from official documents in AndroidDeveloper[3] and code from Stack Overflow[4] – this addresses Challenge 1 mentioned in Section 3.2. Finally, 16 indicator instructions that have selected based on the manual analysis are listed in Table 1. Since these indicator instructions are the official APIs provided by Java programming languages, we assume that the authentication procedure implemented in the Android apps must invoke at least one of them to achieve PAP. Details about each indicator instruction are introduced below.

3. Android Developer: https://developer.android.com/
4. Stack Overflow: https://stackoverflow.com/search?q=password+authentication

BPAP. AUTHEXPLOIT relies on indicator instructions 1 and 2 to identify whether a BPAP is implemented in an app. Function `getPassword` is called to return a user's password. A combination of the username and the password is returned by function `requestPasswordAuthentication`.

SSL/TLS. Indicator instructions 3 - 12 are commonly used in the implementation of SSL/TLS protocol. Indicator instructions 3 - 5 are used to create a socket connection. Function `createSocket` creates a connection with a specified remote host at a specified port. Function `getInstance` in class `javax.net.ssl.SSLContext` can specify which protocol to use (e.g., SSL, TLS). A function call of `getSession` tries to set up a handshake session through the socket. Since SSL/TLS-based password authentication protocol requires a correct certificate validation, indicator instructions 6 - 9 may be called. Function `getLocalCertificates` returns the certificate that is sent to the peer. While using `TrustManager`, function `getInstance` in class `javax.net.ssl.TrustManagerFactory` is invoked. Functions `verify` and `checkValidity` are certificate validation methods. Indicator instructions 10 - 12 are used in the implementation of hostname verifier. Function `verify` in class `javax.net.ssl.HostnameVerifier` has to be invoked To verify a hostname. In both classes of `javax.net.ssl.HttpsURLConnection` and `org.apache.http.conn.ssl.SSLSocketFactory`, each specifies a verification function `setHostnameVerifier` to accept the valid hostname for secure connection construction.

Timestamp. Indicator instructions 13 - 16 are timestamp-related functions. Functions `currentTimeMillis()` and `nanoTime()` generate unique timestamps in the format of "YYYY-MM-DD hh:mm:ss". Functions `init` and `toMillis` from class `java.util.Date` and class `java.util.concurrent.TimeUnit` generate

TABLE 1
Indication Instructions

Password Authentication Protocol	Instruction #	Indicator Instruction
BPAP	1	java.net.PasswordAuthentication char[] getPassword();
	2	java.net.Authenticator java.net.PasswordAuthentication requestPasswordAuthentication(java.lang.String, java.net.InetAddress, int, java.lang.String, java.lang.String, java.lang.String, java.net.Authenticator$RequestorType)
SSL-based Password Authentication Protocol	3	javax.net.ssl.SSLSocketFactory java.net.Socket createSocket(java.lang.String, int);
	4	javax.net.ssl.SSLContext javax.net.ssl.SSLConext getInstance (java.lang.String);
	5	javax.net.ssl.SSLSocket javax.net.ssl.SSLSession getSession();
	6	javax.net.ssl.SSLSession java.security.cert.Certificate[] getLocalCertificates()
	7	javax.net.ssl.TrustManagerFactory javax.net.ssl.TrustManagerFactory getInstance(java.lang.String);
	8	java.Security.cert.X509Certificate void verify(java.security.PublicKey)
	9	java.security.cert.X509Certificate: void checkValidity();
	10	javax.net.ssl.HostnameVerifier boolean verify(java.lang.String, java.net.ssl.SSLSesstion);
	11	javax.net.ssl.HttpsURLConnection void setHostnameVerifier(javax.net.ssl.HostnameVerifier);
	12	org.apache.http.conn.ssl.SSLSocketFactory void setHostnameVerifier(org.apache.http.conn.ssl.X509HostnameVerifier);
Timestamp-Based Password Authentication Protocol	13	java.lang.System long currentTimeMillis();
	14	java.lang.System long nanoTime();
	15	java.util.Date void init();
	16	java.util.concurrent.TimeUnit long toMillis(long);

timestamps as "Days, Hours, Minutes, Seconds". It is important to note that we only analyze the code snippets that invoke both the timestamp instructions and BPAP instructions to ensure the effectiveness and efficiency of our detection.

Given the indicator instructions, AUTHEXPLOIT matches nodes in the super call graph of each app to identify the BPAP. For the apps labeled as BPAP-related, AUTHEXPLOIT further performs static program slicing to analyze its implementation in detail.

4.2.2 Static Program Slicing

To learn the root cause of each flaw, AUTHEXPLOIT applies program slicing [26] (forward [27] and backward [28] program slicing). By analyzing a specific subset of the behavior defined in a given program, AUTHEXPLOIT extracts correlations between the BPAP and the protection schemes in an app to determine whether they are relevant to each other.

According to the matched nodes in the super call graph, AUTHEXPLOIT locates the code snippet with BPAP. It further identifies SSL/TLS and timestamps. Our approach to identify the location of relevant code snippet addresses Challenge 2 and Challenge 3 (see Section 3.2) since AUTHEXPLOIT only recognizes statements that might be related to password authentication protocols. AUTHEXPLOIT then traverses the program backward or forward starting from a matched node (i.e., the function that matches with any indicator instructions). It initially creates an empty data flow set in the form of $< O_n, O_p >$ to record the execution, where O_n is a variable or a function. O_p is a variable, a value, a function call, or a field. $< O_n, O_p >$ indicates that a variable O_n is assigned by O_p or a function O_n takes O_p as input. AUTHEXPLOIT takes the following two steps to add the dependent objects to the data flow set. First, it extracts variables that are taken as input of an indicator instruction or assigned by indicator instructions. Then, it moves forward or backward to analyze the dependencies and extracts operations on the other dependent variables. This algorithm does not terminate until a variable is assigned to a real value or the super call graph terminates.

4.2.3 Misuse Detection

Based on data and control dependencies defined for each app, AUTHEXPLOIT checks whether the code has any of the three flaws discussed in Section 2.2. Specifically, it checks whether the implementation of the password authentication protocol complies with the corresponding requirements.

***Flaw 1*: Plain password transmission.** According to the indicator instruction 1 or 2, AUTHEXPLOIT determines where the password is. If it does not detect any additional protection scheme that is dependent on the password, it labels the app as insecure in that users' passwords are transmitted in plaintext.

***Flaw 2*: Insufficient SSL/TLS validation.** AUTHEXPLOIT analyzes certificate signatures and hostnames separately. First, it identifies the node with indicator instruction 6. It returns the certificate that is used during communication.

Then, it analyzes the nodes that are related to the certificate. If a validation function is found and it never throws an exception or reports an invalid certification (i.e., return true by default), AUTHEXPLOIT regards the implementation as affected by Flaw 2. In addition, `TrustManager` is another way to manage trust-related information. AUTHEXPLOIT thus analyzes the implemented `TrustManager` and checks if it reports invalid certifications.

Depending on the connection with customized hostname verification, AUTHEXPLOIT relies on either indicator instructions 11 or 12. It further examines the verification function and its return value. AUTHEXPLOIT regards an implementation as vulnerable if the input of `setHostnameVerifier` is set as `ALLOW_ALL_HOSTNAME_VERIFIER` or the customized verfication function returns `TRUE` by default.

***Flaw 3*: Incomplete timestamp format.** AUTHEXPLOIT extracts the dependence relationship between the timestamp-related indicator instructions and the BPAP. If they are correlated, it further examines the format of the timestamp (i.e., the value assigned to the timestamp variable). AUTHEXPLOIT finds a flaw if the format is incomplete, that is, the required format "Year/Month/Day/Hour/Minute/Second" is not followed. For indicator instructions 13 and 14, they generate a valid timestamp format only if their return types are defined as "long"; otherwise, the timestamp is incomplete. In addition, AUTHEXPLOIT checks the format of the date format when the indicator instructions 3 and 4 are detected. It only classifies the implementation secure if the format of "Days" is customized as "YYYY-MM-DD".

5 EVALUATION

In this section, we conduct two experiments to analyze the implementation of password authentication in Android apps. First, we assess the effectiveness of orchestrated and automated approaches for exploring authentication flaws in Android apps. Then, we analyze the correlations between the occurrence of authentication flaws and various app characteristics.

5.1 Assessment of Orchestration Vs Automation

5.1.1 Dataset

We randomly collected 1,200 free apps from the Official Google Play site[5]. In order to ensure that our dataset has a wide coverage and does not have a bias towards any particular type of app, we included apps from six categories: *Communication*, *Dating*, *Finance*, *Health & Fitness*, *Shopping*, and *Social Networking*. We downloaded 600 highly popular apps (100 in each category) which had at least 1,000 ratings and four stars or more. The other 600 apps are less popular (100 in each category) with 200 or fewer ratings. The size of the apps used ranges from 1MB to 70MB.

Due to the lack of an open source labeled dataset of apps with identified authentication flaws, we created our own. As most implementations of password authentication protocols follow the same structure, we believed that the structures are generalizable enough for our purpose.

5. Google Play: https://play.google.com/store?hl=en

For creating this ground-truth dataset, we asked a team of annotators (1 PhD student and 2 postdoctoral research fellows), all with more than 7 years of programming experience in Java, to check implementations of password authentication protocols in apps. We first required team members to label apps independently. Then all members went through the labels together and discussed apps that were labeled differently. The team had to come to an agreement before an app could be included in the dataset. To evaluate whether the agreement was good enough, we computed the Fleiss's Kappa score [29]. The kappa score of the agreement is 0.901, which means there was almost perfect agreement. Ultimately this procedure found a total of 1,205 implementations of password authentication protocols in 742 Android apps (since some apps implement multiple schemes), and 1,087 authentication flaws were identified in 695 apps (Flaw 1: 284, Flaw 2: 736, Flaw 3: 67).

5.1.2 Experiment Design

To evaluate the performance of AUTHEXPLOIT, we generated an evaluation matrix of Precision, Recall, and F1 metrics. Precision measures how precise/accurate our tool is; recall reflects how many vulnerabilities are actually detected; and F1 is used to balance precision and recall. They are defined as follows:

$$\begin{aligned} Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TP}{TP+FN} \\ F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \end{aligned}$$

where TP is the number of authentication flaws that are detected correctly, FP is the number of authentication flaws that are detected incorrectly (false positives), and FN is the number of authentication flaws that are not detected.

5.1.3 Performance Comparison

To evaluate the detection effectiveness of orchestrated and automated tools, not only did we compare our orchestration approach, AUTHEXPLOIT, with our proposed automation tool, GLACIATE [16], but also compares with two state-of-the art orchestration tools, MalloDroid [12] and SMV-Hunter [17]. Similar to AUTHEXPLOIT, both MalloDroid and SMV-Hunter require manual efforts; MalloDroid relies on predefined rules to identify vulnerable implementations, and SMV-Hunter triggers vulnerabilities by using manually generated inputs. Different from AUTHEXPLOIT and GLACIATE, MalloDroid and SMV-Hunter only focus on examining the correctness of the SSL/TLS implementation.

For comparison we applied AUTHEXPLOIT, GLACIATE, MalloDroid, and SMV-Hunter to the entire dataset. As GLACIATE relies on agglomerative hierarchical clustering to learn flaw patterns automatically, we used 10-fold cross validation [30] to evaluate the performance. Following the original settings of GLACIATE, we set $T_{dist} = 1.3$ and $min_{rule} = 2$.

Since MalloDroid and SMV-Hunter only detect SSL/TLS-related vulnerabilities (i.e., Flaw 2 in this paper), we limited AUTHEXPLOIT and GLACIATE to detect Flaw 2 in this

TABLE 2
Detection Result: AUTHEXPLOIT, MalloDroid, SMV-Hunter, and GLACIATE

Flaw	AUTHEXPLOIT		MalloDroid		SMV-Hunter		GLACIATE	
	Detected	Correct	Detected	Correct	Detected	Correct	Detected	Correct
Flaw 2	668	627	241	201	627	572	751	686
Precision	93.9%		91.3%		93.9%		91.2%	
Recall	85.4%		27.3%		77.7%		93.5%	
F1	89.5%		42.3%		83.9%		92.4%	

experiment. From the results, we computed the Precision, Recall and F1 over the entire dataset for each tool.

Table 2 shows the assessment results. Specifically, the automated tool, GLACIATE, performed better than the other tools, achieving a F1 of 92.4%. Different from GLACIATE, the other three detection tools are orchestration tools that rely on manual efforts to determine the vulnerable patterns. Therefore, AUTHEXPLOIT, MalloDroid, and SMV-Hunter achieved a higher precision than GLACIATE. More specifically, AUTHEXPLOIT has the highest precision that correctly detected 627 out of 736 SSL/TLS authentication flaws, with the Precision value of 93.9%. Comparing with GLACIATE, AUTHEXPLOIT has a 2.96% better precision. Through our manual inspection, we found that GLACIATE filtered out some control flows that contain customized functions. For these situations, developers generally declared authentication activities into different Java classes. When the declared activities are too complicated, GLACIATE fails to identify the correlations among these Java classes.

In addition, MalloDroid only detected 201 flaws out of 736 SSL/TLS-related authentication flaws, achieving a recall of only 27.3%. AUTHEXPLOIT detected about 2 times more SSL/TLS-related authentication flaws than MalloDroid. SMV-Hunter successfully detected 572 SSL/TLS-related authentication flaws with Precision, Recall and F1 values of 91.2%, 77.7%, and 83.9%. Compared with SMV-Hunter, AUTHEXPLOIT identified 9.6% more flaws and has a 2.9% better Precision. This means that AUTHEXPLOIT generates proportionally fewer false positives than SMV-Hunter.

TrustManagers are responsible for managing the trust material that is used for deciding whether the received public key certificates should be accepted. Besides the vulnerable TrustManagers detected by MalloDroid, AUTHEXPLOIT also found three new types of vulnerable TrustManagers, namely BlindTrustManager, InsecureTrustManager and AllTrustingTrustManager. Apps with these vulnerable TrustManagers suffer from Flaw 2.

5.1.4 Further Analysis of Performance

In comparing the detection performance, we found that MalloDroid and SMV-Hunter fails to analyze apps with authentications implemented in different classes.

AUTHEXPLOIT and GLACIATE did fail to analyze some apps successfully. Since they were built on top of Soot, each app has to be decompiled using Soot. In total, Soot was unable to decompile 184 apps, failing in "Soot.PackManager". This method runs the ThreadPoolExecutor multiple times, and the executor Runnable is unable to handle those threads separately. These fail-to-decompile apps will be reconsidered when Soot is next upgraded[6].

5.2 Characteristics of Apps with Authentication Flaws

To gain further insights, we posed the following questions to determine whether the occurrence of authentication flaws is correlated with selected app characteristics:

- **RQ1:** Are paid apps more secure than free apps?
- **RQ2:** Are highly popular apps more secure than less popular apps? How long does it take to repair an authentication flaw?
- **RQ3:** Which is the most secure category, e.g., e-commerce apps?
- **RQ4:** Which party is responsible for these authentication implementation flaws - third party packages or developers?

5.2.1 RQ1: Paid vs Free

From a user's perspective, one might intuitively expect that a paid app would be designed by a professional developer, and thus be more secure than a free app. However, previous research has shown that paid apps have more vulnerabilities since they include more code and functionalities [31]. Hence, we hypothesized that being paid or not does correlate with the correctness of password authentication implementation. To evaluate the relationship between payment and authentication flaws, we randomly downloaded the top-150 paid apps and the top-150 free apps from Google Play. We ran AUTHEXPLOIT to detect authentication flaws in these apps, and for each app.

We used the Mann Whiteney U test [32], intended for testing differences in the means of two independent samples. It determines the significance of the differences in the occurrence of authentication flaws in paid and free apps. The test is applied to validate the following null (H_0) and alternative (H_1) hypotheses:

- H_0: The mean number of authentication flaws found in free apps $\leq$ the mean number of authentication flaws found in paid apps.
- H_1: The mean number of authentication flaws found in free apps $>$ the mean number of authentication flaws found in paid apps.

We ran the Mann Whitney U Test by performing a one tail test and setting the significance level at 5%. The

6. The exception, "ERROR heros.solver.CountingThreadPoolExecutor - Worker thread execution failed: Dex file overflow", was posted in March, 2018. Soot might solve this problem in its next version.

calculated result shows that $p - value = 5.4e^{-8} < 5\%$. Thus, H_0 is rejected, which means that authentication flaws found in paid apps are significantly fewer than flaws found in free apps.

5.2.2 RQ2: Highly Popular vs Less Popular

The popularity of apps might also influence the implementation of password authentication protocols. We hypothesized that highly popular Android apps are not more secure than less popular apps. A prior study by Scholte et al. [33] has concluded that the highly popular web apps are not less vulnerable than the less popular ones. The following analysis was conducted on the vulnerable apps detected by AUTHEXPLOIT to evaluate the correlation between app popularity and authentication flaws.

The null (H_0) and alternative (H_1) hypotheses for the Mann Whiteney U Test are:

- H_0: There is no difference in the mean number of authentication flaws found in the highly and less popular apps.
- H_1: There is difference in the mean number of authentication flaws found in the highly and less popular apps.

For this dataset, we ran the Mann Whitney U Test as a one tail test with alpha = 0.5. The calculation gave $p - value = 0.579 > 0.5$, which means H_0 cannot be rejected. Thus, there is no statistically significant differences between the number of authentication flaws found in the highly popular and less popular apps.

Besides, we also investigated how promptly developers fix authentication flaws. We randomly chose 200 apps and downloaded their previous updated versions at intervals of three months. Figure 2 reports the results of such analysis. AUTHEXPLOIT initially found that the number of apps with Flaws 1, 2, 3 are 13, 24, 4, respectively. Then an analysis of the subsequent versions showed that most authentication flaws are fixed after six months. However, 13 apps retained their flaws for more than one year.

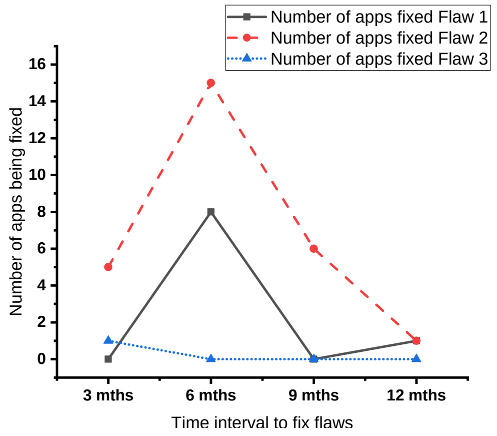

Fig. 2. Time Interval to Fix Authentication Flaws.

5.2.3 RQ3: App Categories

Since users might have different security expectations for different application categories, we were interested in assessing whether e-commerce apps are more secure than apps in other categories. Vijayaraghavan and Kaner [34] highlighted that e-commerce apps often use secure protocols, such as SSL, to protect online transactions. Therefore, we were expecting that the password-based authentication schemes of e-commerce apps (e.g., shopping and finance apps) would be more secure than apps in other categories. We used the flaw detection results for the different categories to analyze the association between app categories and authentication flaws.

We used the Chi-Square Independent test [35] to determine whether there is a significant relationship between two categorical variables: type of authentication flaw and app categories (i.e., Communication, Dating, Finance, Health & Fitness, Shopping, and Social Networking). The test is used to validate the following null (H_0) and alternative (H_1) hypotheses:

- H_0: There is no association between authentication flaws and app categories.
- H_1: There is an association between authentication flaws and app categories.

We set the significance level to 5%. H_0 is rejected if the p-value $\leq 5\%$. The Chi-Square Independent test gave $p(\chi^2 > 16.040) = 0.379$. Thus, H_0 cannot be rejected, i.e., there is no statistically significant association between app categories and authentication flaws.

We further built a multiple linear regression model to estimate how these categories are affected by the occurrence of authentication flaws. Regarding the coefficients as noteworthy, a category is statistically significant if $p < 5\%$. Based on the results of this multiple linear regression model, we observed that no category is significantly associated with the authentication flaws when $p - value > 5\%$ is used as test.

5.2.4 Causes of Authentication Flaws

There are a number of libraries available for developers to implement secure password authentication protocols [36], such as native SSL/TLS libraries supported by Android, Firebase Authentication, GnuTLS, openSSL[7]. We further examined the ground-truth dataset to investigate whether the found authentication flaws were related to the use of external libraries for implementation or due to incorrect custom implementations by developers.

By analyzing the flawed customized implementations, we observed that they are written as not to check the hostname or never throw exceptions regardless of the certificate validity. A more interesting finding is that although an external library provides an option to build a secure SSL implementation, developers still choose to ignore invalid certificates and hostnames even though an option of verification is provided. One potential reason why developers do so is that verifying certificates and hostnames may slow the applications at runtime.

5.3 Limitation

By manually inspecting the detection results, we conclude the following limitation of AUTHEXPLOIT.

7. Firebase: https://firebase.google.com/docs/auth/; GnuTLS: https://www.gnutls.org/; openSSL: https://www.openssl.org/

[Unsuccessful Decompilation.] Since we build AUTHExploit on top of Soot, each application has to be decompiled using Soot. In total, Soot fails to decompile 184 applications, which is caused by "Soot.PackManager". It runs the ThreadPoolExecutor multiple times, and the executor Runnable is unable to handle those threads separately. These unsuccessful applications can be analyzed again once Soot is upgraded [8].

[Complex Dependencies.] For some cases, the same SootField may be used by multiple methods. When applying program dependency analysis, two independent SootMethods may be considered as dependent because of the mutual SootField. This impacts the analysis since two independent SootMethods will be analyzed as one. Suppose that two independent SootMethods $method_1$ and $method_2$ are connected through a mutual SootField. In $method_1$, a PAP with the timestamp scheme uses a timestamp generated by calling the method *currentTimeMillis()*. However, $method_2$ only extracts the date from method *currentTimeMillis()* for the other purpose. AUTHExploit considers such date as the timestamp used for PAP and determines that the PAP with timestamp implementation is insecure.

[Customized Authentication.] Since we summarize the indicator instructions through the official documents and websites, we only collect the APIs that are official included in the Java programming languages. Therefore, for some authentication procedures that are designed by developers, AUTHEXPLOIT is unable to exploit them.

6 RELATED WORK

We broadly classify existing flaw detection techniques to two broad categories: orchestration and automation. Orchestration refers to the approaches that put human in the loop, whereas automation refers to the approaches that do not need human intervention. Orchestration approaches can be further classified into two categories, rule-based techniques and attack-based techniques. Machine learning approaches are all put together under the machine learning techniques below.

6.1 Rule-based Techniques

Most existing techniques detect vulnerabilities by using pre-defined rules/templates. MalloDroid [12] is a tool for analyzing SSL/TLS code in Android apps to check whether the code is potentially vulnerable to MITM attacks. MalloDroid checks the Internet permissions and analyzes the network API calls to identify certificates and hostname verification code. According to the API calls and permissions, MalloDroid determines whether the code has vulnerabilities, including accepting all certificates, accepting all hostnames, trusting many CAs, and using mixed-mode/no SSL. However, because it only analyzes the network API calls, MalloDroid is unable to identify all the potential flaws due to its inability to extract the inter-component communications. Instead of constructing a call graph/control flow graph, HVLearn [37] is a black-box learning approach that infers certificate templates from the certificates with certain common names by using an automated learning algorithm. It further detects those invalid certificates that cannot be matched with certificate templates. However, this approach can only be applied to the certificates with specific common names. Spinner [38] is also a tool targeting on black-box detection. It checks certificate pinning vulnerabilities which may hide improper hostname verification and enables MITM attacks. Without requiring access to the code, Spinner generates traffic that includes a certificate signed by the same CA, but with a different hostname. It then checks whether the connection fails. A vulnerability is detected if the connection is established and encrypted data is transmitted. However, some unnecessary input will be generated while applying a fully automated approach.

Other types of vulnerabilities are detected using similar approaches. IntentSoot [39] detects intent injection vulnerabilities by using static analysis. It builds call and control flow graphs for an app, and then tracks taint propagation within a component, between components, and during the reflection call. An intent injection vulnerability is detected if the taint propagation violates the defined rules. Similarly, cryptographic misuses are predefined in CRYPTOLINT [40]. By manually analyzing Android apps, the correct ways of implementing cryptographic algorithm are provided. CRYPTOLINT first computes a super control flow graph for the app, and then uses program slicing to identify whether the cryptographic implementation violates any of the correct implementations. IoTFUZZER [41] aims to detect flaws in IoT devices. Without precisely locating software flaws, IoTFUZZER performs a dynamic analysis to identify the content in IoT apps, and then forms and delivers messages to the target devices. It monitors the devices to capture a triggered crash. Manual effort is needed in defining the fuzzing policy which IoTFUZZER follows to generate messages.

Besides analyzing source code or binary code of software, some researchers created inputs to trigger the corresponding vulnerabilities. DTaint [42] performs static analysis to detect taint-style vulnerabilities - vulnerabilities in the firmware where an input results in an unsafe path to a sensitive sink. By analyzing the unsafe paths (from sources to sensitive sinks), DTaint analyzes data dependency from callees to callers. DTaint explores vulnerabilities through static code analysis, SMV-Hunter [17] conducts dynamic analysis by simulating user interactions and launching MITM attacks to detect SSL vulnerabilities. However, the detection performance relies on how well user inputs are created, and some vulnerabilities cannot be identified since they are not triggered by the MTIM attacks. Like SMV-Hunter, Hush [43] also uses hybrid analysis to detect server-based oversharing vulnerabilities. First, Hush applies static analysis to invoke network APIs that can deserialize network data and manipulate user interfaces. Then, it uses dynamic analysis to test the candidate vulnerabilities. To confirm a vulnerability, Hush analyzes whether some specific fields have flows from sources to sinks. Hush is a semi-automated detection tool, requiring one to create an account for each app before connecting to the server. Although inputs can be generated automatically, it is still a challenge to construct precise inputs to trigger vulnerabilities.

8. This exception is proposed in March, 2018. Soot might solve this problem in its next version.

6.2 Attack-based Techniques

Instead of using rules/templates, some orchestration approaches locate vulnerabilities by launching attacks. AUTHScope [44] targets the vulnerabilities at the server side. Since it is difficult to extract the source code running on the remote servers, AUTHScope sends various network requests to the server and applies differential traffic analysis to identify when the server does not provide proper token verification. D'Orazio et al. [45] applied their detection tool to Android and iOS. The tool creates a self-signed certificate to start a connection with the server to identify whether a self-signed certificate can be accepted. However, those two approaches only analyze one type of vulnerability, do not consider apps with multiple protection schemes and can only identify apps without hostname verification. Instead of launching one attack, six different attack scenarios must be launched by AndroSSL [13], which provides an environment for developers to test their apps against connection security flaws. The environment has an actual server that accepts authentication requests and static and dynamic URLs without verifying the hostnames and certificates. Chen et al. [46] focused on the Host head of HTTP implementations. They launched a new attack "Host of Troubles" on those HTTP implementations, and analyzed behaviours of implementations in their handling of Host headers. Such approach only detected a specific type of vulnerability, namely SSL-related vulnerabilities.

6.3 Machine Learning Techniques

Orchestration approaches might be inefficient because of the involvement of manual efforts. To address this drawback, machine learning can be used to construct a fully automated detection approach. Harer et al. [47] created two models, a build-based model and a source-based model, to detect vulnerabilities. Build-based features are collected from the function level by extracting operations (i.e., the definition and use of variables), and source-based features are collected from source code by extracting individual tokens (e.g., types, variable names and comments). However, the result is imprecise since source code is not simply a string of words, but syntax- and semantic-based representations.

VulDeePecker [48] and SySeVr [49] detect vulnerabilities by using deep learning, which can replace human expert effort while learning. By extracting library/API function calls, VulDeePecker generates training vectors to represent the invocations of these function calls. It then trains a bidirectional long short-term memory (BLSTM) neural network model with the training vectors. To improve the detection accuracy, SySeVr collects more features, including function calls, array usage, pointer usage, and arithmetic expressions for training. Although VulDeePecker and SySeVr detected many types of vulnerabilities without any manual effort, a strong requirement for the training dataset is that each code segment includes only one vulnerability.

A large dataset is always required for machine learning approaches. DRLgencert [50] tests certificate validation in SSL/TLS implementations by using deep reinforcement learning. It first chooses one certificate from the dataset to perform differential testing. If a discrepancy is triggered, this certificate is valid. Otherwise, DRLgencert collects some features from the certificate based on a pre-defined feature extraction scheme, and then sends it to reinforcement learning model to be modified. WoodPecker [51] detects shadowed domains by using machine learning models (Support Vector Machine, Random Forest, Logistic Regression, Naive Bayes, and Neural Network models). It takes as input a large set of training data (confirmed shadowed domains) and then collects 17 features to train its classification models.

The above detection approaches which use machine learning/data mining have the desirable property of working automatically and we investigated their application to our problem. We extracted control flow graphs and used different machine learning algorithms (i.e., convolutional neural network (CNN), decision tree, naive Bayes, support vectore machine (SVM), and logistic regression) to build detection models. However the detection results were found to be poor. For example, SVM – the approach with the highest accuracy, had Precision, Recall, and F1 of 52.7%, 92.9%, and 67.6%, respectively. Our observations show that the main difficulties in using these approaches for our problem is the question of how to filter the useful features and how to build a model accurately from limited data.

7 Conclusion

In this paper, we studied the detection effectiveness of two detection schemes, orchestration and automation, while exploring authentication flaws in Android apps. According to the results generated by GLACIATE, we were interested in whether the performance can be improved by involving the manual effort (i.e., orchestration approach). Targeting on the authentication flaws in Android apps, i.e., *Flaw 1* — plain password transmission, *Flaw 2* — insufficient SSL/TLS validation and *Flaw 3* — incomplete timestamp format, we manually created several detection templates and then built an orchestration tool, AUTHEXPLOIT, to detect these flaws.

We assessed the effectiveness of AUTHEXPLOIT by applying it to 1,200 real world Android apps and then compared it with GLACIATE, and other two orchestration tools, MalloDroid, and SMV-Hunter. The results indicated that AUTHEXPLOIT if more accurate than GLACIATE with a higher Precision. However, GLACIATE performed the best when we consider the entire performance. While only comparing the orchestration tools, AUTHEXPLOIT performed better than both MalloDroid and SMV-Hunter. To gain a further insight, we focused on paid apps, popular apps and different app categories. Through statistic analysis, we analyzed whether these factors affect the implementation correctness of the password authentication protocol significantly. The results proved that there are no correlations between these features and the implementation correctness.

We intend to make AUTHEXPLOIT available as an open source tool that help towards the development of secure Android applications. However, app developers may need training to recognise the potential security threats that are most significant. As we only focus in this work on the timestamp-based password authentication protocol, we plan to extend AUTHEXPLOIT with additional security rules to capture implementation defects in nonce-based password authentication. Furthermore, as several implemented authentication flaws can be fixed automatically, it is also important to develop approaches to correct the defects or provide fix suggestions when they are detected.

REFERENCES

[1] S. Yovine and G. Winniczuk, "Checkdroid: a tool for automated detection of bad practices in android applications using taint analysis," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 175–176.

[2] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, no. 11, pp. 770–772, 1981.

[3] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password interception in a ssl/tls channel," in *Annual International Cryptology Conference*. Springer, 2003, pp. 583–599.

[4] L. O'Gorman, "Comparing passwords, tokens, and biometrics for user authentication," *Proceedings of the IEEE*, vol. 91, no. 12, pp. 2021–2040, 2003.

[5] J. Hubbard, K. Weimer, and Y. Chen, "A study of ssl proxy attacks on android and ios mobile applications," in *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*. IEEE, 2014, pp. 86–91.

[6] C.-M. Chen, L. Xu, W. Fang, and T.-Y. Wu, "A three-party password authenticated key exchange protocol resistant to stolen smart card attacks," in *Advances in Intelligent Information Hiding and Multimedia Signal Processing*. Springer, 2017, pp. 331–336.

[7] H.-M. Sun, Y.-H. Chen, and Y.-H. Lin, "opass: A user authentication protocol resistant to password stealing and password reuse attacks," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 2, pp. 651–663, 2012.

[8] T. Tsuji and A. Shimizu, "An impersonation attack on one-time password authentication protocol ospa," *IEICE Transactions on Communications*, vol. 86, no. 7, pp. 2182–2185, 2003.

[9] R. Amin, N. Kumar, G. Biswas, R. Iqbal, and V. Chang, "A light weight authentication protocol for iot-enabled devices in distributed cloud computing environment," *Future Generation Computer Systems*, vol. 78, pp. 1005–1019, 2018.

[10] M. Mannan and P. C. Van Oorschot, "Using a personal device to strengthen password authentication from an untrusted computer," in *International Conference on Financial Cryptography and Data Security*. Springer, 2007, pp. 88–103.

[11] C.-W. Lin, C.-S. Tsai, and M.-S. Hwang, "A new strong-password authentication scheme using one-way hash functions," *Journal of Computer and Systems Sciences International*, vol. 45, no. 4, pp. 623–626, 2006.

[12] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 50–61.

[13] S. Desloges, J. Ouellet, and C. Boileau, "Androssl: A platform to test android applications connection security," in *Foundations and Practice of Security: 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, vol. 9482. Springer, 2016, p. 294.

[14] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 859–874.

[15] D. P. Jablon, "Strong password-only authenticated key exchange," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 5, pp. 5–26, 1996.

[16] S. Ma, E. Bertino, S. Nepal, J. Li, D. Ostry, R. H. Deng, and S. Jha, "Finding flaws from password authentication code in android apps," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 619–637.

[17] 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 *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSSÂ¡Â14*. Citeseer, 2014.

[18] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[19] M. Conti, N. Dragoni, and V. Lesyk, "A survey of man in the middle attacks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2027–2051, 2016.

[20] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

[21] K. Alghamdi, A. Alqazzaz, A. Liu, and H. Ming, "Iotverif: An automated tool to verify ssl/tls certificate validation in android mqtt client applications," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 2018, pp. 95–102.

[22] A. Juels, N. Triandopoulos, M. E. Van Dijk, and R. Rivest, "Methods and apparatus for silent alarm channels using one-time passcode authentication tokens," Dec. 6 2016, uS Patent 9,515,989.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[24] A. Einarsson and J. D. Nielsen, "A survivor's guide to java program analysis with soot," *BRICS, Department of Computer Science, University of Aarhus, Denmark*, p. 17, 2008.

[25] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1. ACM, 2005, pp. 13–19.

[26] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[27] B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, 1994, pp. 66–79.

[28] H. Khanfar, B. Lisper, and A. N. Masud, "Static backward program slicing for safety-critical systems," in *Ada-Europe International Conference on Reliable Software Technologies*. Springer, 2015, pp. 50–65.

[29] J. L. Fleiss, B. Levin, and M. C. Paik, *Statistical methods for rates and proportions*. John Wiley & Sons, 2013.

[30] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, no. 2. Montreal, Canada, 1995, pp. 1137–1145.

[31] T. Watanabe, M. Akiyama, F. Kanei, E. Shioji, Y. Takata, B. Sun, Y. Ishi, T. Shibahara, T. Yagi, and T. Mori, "Understanding the origins of mobile app vulnerabilities: A large-scale measurement study of free and paid apps," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 14–24.

[32] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[33] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? an empirical study on input validation vulnerabilities in web applications," *Computers & Security*, vol. 31, no. 3, pp. 344–356, 2012.

[34] G. Vijayaraghavan and C. Kaner, "Bugs in your shopping cart: A taxonomy," *Retrieved July*, vol. 30, p. 2003, 2002.

[35] D. S. Moore, "Chi-square tests." PURDUE UNIV LAFAYETTE IND DEPT OF STATISTICS, Tech. Rep., 1976.

[36] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying tls usage in android apps," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2017, pp. 350–362.

[37] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 521–538.

[38] C. M. Stone, T. Chothia, and F. D. Garcia, "Spinner: Semi-automatic detection of pinning without hostname verification," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 176–188.

[39] B. Xiong, X. Guangli, T. Du, J. S. He, and S. Ji, "Static taint analysis method for intent injection vulnerability in android application," in *International Symposium on Cyberspce Safety and Security*. Springer, 2017, pp. 16–31.

[40] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 73–84.

[41] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *In Proceedings of the 21st Annual Network and Distributed System Security Symposium*. Citeseer, 2018.

[42] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, “Dtaint: detecting the taint-style vulnerability in embedded device firmware,” in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 430–441.

[43] W. Koch, A. Chaabane, M. Egele, W. Robertson, and E. Kirda, “Semi-automated discovery of server-based information oversharing vulnerabilities in android applications,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 147–157.

[44] C. Zuo, Q. Zhao, and Z. Lin, “Authscope: Towards automatic discovery of vulnerable authorizations in online services,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 799–813.

[45] C. J. D’Orazio and K.-K. R. Choo, “A technique to circumvent ssl/tls validations on ios devices,” *Future Generation Computer Systems*, vol. 74, pp. 366–374, 2017.

[46] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxon, “Host of troubles: Multiple host ambiguities in http implementations,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1516–1527.

[47] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood *et al.*, “Automated software vulnerability detection with machine learning,” *arXiv preprint arXiv:1803.04497*, 2018.

[48] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” *arXiv preprint arXiv:1801.01681*, 2018.

[49] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” *arXiv preprint arXiv:1807.06756*, 2018.

[50] C. Chen, W. Diao, Y. Zeng, S. Guo, and C. Hu, “Drlgencert:deep learning-based automated testing of certificate verification in ssl/tls implementations,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 48–58.

[51] D. Liu, S. Li, K. Du, H. Wang, B. Liu, and H. Duan, “Don’t let one rotten apple spoil the whole barrel: Towards automated detection of shadowed domains,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 537–552.