Finding Flaws from Password Authentication Code in Android Apps

Siqi Ma[1](✉), Elisa Bertino[2], Surya Nepal[1], Juanru Li[3], Diethelm Ostry[1], Robert H. Deng[4], and Sanjay Jha[5]

[1] CSIRO, Sydney, Australia
{siqi.ma,surya.nepal,diet.ostry}@csiro.au
[2] Purdue University, West Lafayette, USA
bertino@purdue.edu
[3] Shanghai Jiao Tong University, Shanghai, China
jarod@sjtu.edu.cn
[4] Singapore Management University, Singapore, Singapore
robertdeng@smu.edu.sg
[5] University of New South Wales, Sydney, Kensington, Australia
sanjay.jha@unsw.edu.au

Abstract. Password authentication is widely used to validate users' identities because it is convenient to use, easy for users to remember, and simple to implement. The password authentication protocol transmits passwords in plaintext, which makes the authentication vulnerable to eavesdropping and replay attacks, and several protocols have been proposed to protect against this. However, we find that secure password authentication protocols are often implemented incorrectly in Android applications (apps). To detect the implementation flaws in password authentication code, we propose `GLACIATE`, a fully automated tool combining machine learning and program analysis. Instead of creating detection templates/rules manually, `GLACIATE` automatically and accurately learns the common authentication flaws from a relatively small training dataset, and then identifies whether the authentication flaws exist in other apps. We collected 16,387 apps from Google Play for evaluation. `GLACIATE` successfully identified 4,105 of these with incorrect password authentication implementations. Examining these results, we observed that a significant proportion of them had multiple flaws in their authentication code. We further compared `GLACIATE` with the state-of-the-art techniques to assess its detection accuracy.

Keywords: Password authentication protocol · Mobile application security · Authentication protocol flaws · Vulnerability detection · Automated program analysis

1 Introduction

Although a variety of authentication protocols are proposed, most Android applications (apps for short) with online services are still using password to authenticate user's identity because it is simple and inexpensive to create, use and revoke [13]. To validate the identity in the password authentication protocol [18] (named as BPAP in this paper), a user sends a combination of username and password in plaintext to a server through a client app, and the server replies with an authentication-acknowledgement if the received password is valid.

While using BPAP over an insecure communication channel, the transmission and verification of password become vulnerable to many attacks, such as eavesdropping and replay attacks. In recent years, many cases of password leakage, even from those large corporations (e.g., Facebook and Yahoo), are reported. To regulate the use of password, some secure password authentication protocols (PAP) are proposed to help developers validate users' credential: (1) BPAP over Secure Socket Layer/Transport Layer Security (SSL/TLS) [4], which validates the identities of the client and the server by checking their certificates and hostname to set up a secure channel between them [12], and then the client sends the combination of username and password over the secure channel; and (2) nonce-based PAP [30], which utilizes the user's password as a secret key to compute a cryptographic function on a nonce value.

Unfortunately, we found that app developers tend to implement those secure password authentication protocols incorrectly even though the requirements for a secure password authentication are well-defined. A secure protocol with incorrect implementation makes the authentication process become vulnerable to attack. Suppose for example that in an app, a timestamp (Hour/Minute/Second) is generated for use in a password hash. An attacker could then launch replay attacks by using the hashed password at the same time every day.

To detect implementation flaws of PAP in Android apps, several approaches are proposed: MalloDroid [10] detects SSL implementation errors by checking network API calls and Internet permissions. SMV-Hunter [25] detects SSL vulnerabilities by launching MITM attacks, using generated inputs to simulate interactions between users and servers. Chen et al. [5] proposed an approach that targets the host head of HTTP implementations and launched a new attack "Host of Troubles" on those HTTP implementations, and analyzed their behaviour in handling the host headers. However, these approaches are highly implementation dependant (i.e., they rely on specific APIs and inputs that can only recognize certain protocols). To the best of our knowledge, there exist no approach that can analyze password authentication protocols in a more general scope (e.g., BPAP over SSL/TLS and nonce-based PAP). Moreover, most of the detected flaws are summarized in a manual and ad-hoc way, and thus the detection processes are neither automated nor general.

To address the limitations of previous approaches, i.e., implementation dependant and high manual-effort, we propose a novel approach to extend state-of-the-art insecure password authentication implementation detection. Our approach first uses a machine learning algorithm, agglomerative hierarchical clustering, to summarize detection rules in a fully automated way, and then utilizes a fine-grained program analysis to detect flaws in Android apps according to the generated rules. We implemented GLACIATE[1], an automated analysis tool

[1] GLACIATE: proGram anaLysis And maChIne leArning To dEtect.

to support end-to-end automatic detection of insecure password authentication implementations. Given only a small amount of training data, `GLACIATE` creates detection rules automatically. It generates enriched call graphs for the apps and groups similar enriched call graphs into different clusters, and mines the patterns of flaws in each cluster to obtain templates of insecure implementation. `GLACIATE` then uses a forward and backward program slicing to locate the code part of password authentication in an Android app, and compares it with the obtained templates to check whether the implementation is insecure.

To assess the effectiveness of `GLACIATE`, we compared `GLACIATE` with two state-of-the-art tools, `MalloDroid` [10] and `SMV-Hunter` [25]. We found that `GLACIATE` successfully identified 686 authentication flaws that are related to SSL/TLS, achieving precision, recall, and F1 metrics of 91.3%, 93.5%, and 92.4%, respectively. In the mean time, `MalloDroid` and `SMV-Hunter` only detected 201 and 572 flawed apps, respectively. Additionally, we downloaded 16,387 apps from Google Play and utilized `GLACIATE` for a large scale analysis. `GLACIATE` identified 5,667 apps that implemented password authentication protocols, and found that only 28% of them were implemented securely. Among the vulnerable apps detected, 65% suffered from authentication flaws related to SSL/TLS. While analyzing the transmitted passwords, 20% of them transmit passwords with insecure hash, or even in plaintext. Moreover, 15 apps violate all the requirements of establishing PAP.

Contributions: Overall, our contributions are as follows:

- We proposed a novel end-to-end approach to identify authentication flaws from the implementation code of secure password authentication protocols. By analyzing the authentication code of client apps, our approach locates all the authentication flaws accurately.
- We designed a fully automated detection tool, `GLACIATE`. With only limited training data, it uses both intra- and inter-procedural analyses to construct enriched call graphs which represent the call relationships and data dependencies in an app. `GLACIATE` then applies a clustering algorithm to construct rule templates automatically. `GLACIATE` subsequently uses program analysis to match an input app with those rule templates and so identify authentication flaws.
- We compared `GLACIATE` and state-of-the-art tools to assess its detection effectiveness. We also applied `GLACIATE` on a large dataset of Android apps to analyze the implementation code of secure password authentication protocols.

Organization: The rest of this paper is organized as follows. Section 2 provides background information on authentication protocols used in Android apps and their correct implementation. In Sect. 3, we give an overview of `GLACIATE` design and each component of `GLACIATE` in details. In Sect. 4, we evaluate the detection effectiveness of `GLACIATE` against our dataset and compare it with the accuracy of `MalloDroid` and `SMV-Hunter`. We discuss related work in Sect. 5 and Sect. 6 concludes the paper and outlines future work.

2 Common Violations of Password Authentication Protocols

In this section, we give an overview of the most commonly used secure password authentication protocols (SPAP) in Android. In Sect. 2.1, we describe security properties to establish secure password authentication protocols. and then we list four types of violations that are commonly existed in the password authentication implementation and describe how they can be exploited by attackers in Sect. 2.2.

2.1 Secure Password Authentication Protocol

The basic password authentication protocol (BPAP) is intended for users requiring authentication by a local computer or a remote server over a closed network, because BPAP is very simple, and only one message from the client to the server is required, without the need for any cryptographic operations. To establish a secure password authentication protocol (SPAP) over an opened network, the following authentication protocols are commonly used.

BPAP over SSL/TLS. A common mitigation of the BPAP vulnerabilities is using BPAP over SSL/TLS, where SSL/TLS is executed first to establish a secure communication channel between the client and the server and then the username and password are sent over the secure channel.

In SSL/TLS, the server is configured with a pair of public and private keys. The public key is certified by a Certification Authority (CA) which issues a public key certificate to the server. There are over 100 trusted CAs[2] to support Android apps. During the execution of the SSL/TLS protocol, it is crucial that the client correctly performs a number of verifications on the public key certificate received from the server. The verification steps are described as follows.

Step 1: Certificate Validation. The client verifies the server's certificate by performing three different checks [1,10]: (1) whether the certificate is signed by a trusted CA; (2) whether the certificate is self-signed; and (3) whether the certificate has expired.

Step 2: Hostname Verification. The client checks whether the hostname in the subjectAltname field of the certificate matches the host portion of the server's URL in order to make sure that the certificate indeed belongs to the server that the client is communicating with.

Nonce-Based Password Authentication Protocols. Another approach to counter password eavesdropping and replay attacks is the use of nonce-based password authentication protocols [14]. A nonce is a number used only once in the execution of a protocol. Depending on whether the nonce is a random number or a timestamp, nonce-based protocols can be classified into either challenge-response or timestamp-based password authentication protocols. In the former, the server sends a random number as a challenge to the client, and the client uses

[2] https://developer.android.com/training/articles/security-ssl.

the user's password as a secret key to compute a cryptographic function on the nonce (i.e., either by encryption of the nonce or a keyed hash of the nonce), and sends the result to the server. In the latter, the client uses the user's password as a secret key to compute a cryptographic function on a timestamp and sends the result to the server. Due to the use of a nonce, both protocols prevent replay attacks in the sense that any replayed protocol message can be detected as such by the server.

2.2 Authentication Flaws

A password authentication protocol is designed to meet specified security objectives, but its security can be undermined if the implementation is incorrect. We examine the authentication code in real-world apps and compare the implementations with the authentication primitives provided by the developer's guides[3]. Three types of authentication flaws listed below are discovered in Android apps.

Flaw 1: Insecure Password Transmission. Passwords are required to be encrypted and hashed by the client app before transmission. An app without encrypting passwords makes the authentication protocol become vulnerable to eavesdropping and replay attacks. Consider the situation of transmitting an encrypted password without being hashed, the password is easily to be leaked at the server-side.

Flaw 2: Insecure Server Connection. To establish a secure channel between apps and their servers, each app should follow two verification steps mentioned in Sect. 2.1 to validate a server. However, we observe that some apps incorrectly implement these two steps by simply accepting either all certificates or all hostnames.

Accepting all certificates represents that invalid certificates, including certificates signed by untrusted CAs, self-signed certificates, or expired certificates, are also acceptable. It makes an app become vulnerable to several attacks, such as MITM attacks, phishing attacks, and impersonation attacks. An attacker can use a forged certificate to connect with the app to steal users' usernames and passwords.

Only checking the certificate from a server is not enough. An app should also check if the hostname in the certificate matches that in the server's URL. A mismatch in hostname indicates that the server is using someone else's (probably valid) certificate in the SSL/TLS handshake. Any app with this flaw is potentially vulnerable to be connected to a malicious counterfeit server.

Flaw 3: Repeatable Timestamp. Timestamps must be used with great caution in any authentication protocol. For the timestamp-based password authentication protocol, a timestamp in the format of Minute/Second results in the protocol message being replayed every hour at the same minute and second without being detectable by the server. A prudent practice is to have the timestamp in the format of Year/Month/Day/Hour/Minute/Second. This ensures the

[3] Android Developers: https://developer.android.com/.

uniqueness of the timestamp and hence the protocol message in any foreseeable future.

Another potential authentication flaw is use of a repeatable challenge in the challenge-response password authentication protocol. However, without access to the source code of the authentication server, we are not aware of any efficient techniques to determine the randomness of the challenge generated by the server. Hence, we leave the analysis of this implementation flaw in Android apps as part of our future work.

3 GLACIATE

In this section, we describe how GLACIATE detects authentication flaws automatically (i.e. without manual predefined rules). Figure 1 illustrates the workflow of GLACIATE, which contains two phases, *Rules Creation* and *Flaws Detection*. We provide details of each phase below.

3.1 Rules Creation

The rules creation phase generates *rule templates* by processing labeled apps in three steps - flow sequence construction, learning cluster generation, and detection rules mining.

Flow Sequence Construction. GLACIATE extracts *enriched call graphs* by analyzing the Jimple code of each app and traverses each enriched call graph to construct *flow sequences*. Details to construct flow sequences are listed below.

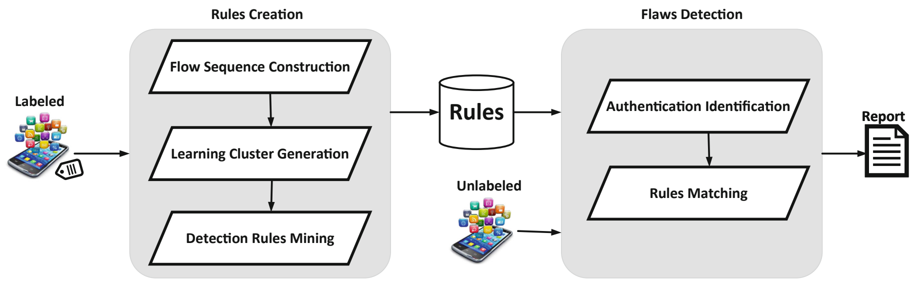

Fig. 1. Workflow of GLACIATE

Enriched Call Graphs Generation. GLACIATE applies Soot [27] to translate low-level Android bytecode into its intermediate representation (IR) (i.e., Jimple code in this paper) and generates enriched call graphs. Each node in an enriched call graph represents either a local function or an external method[4],

[4] A local method is a method designed by developers, and an external method is a system or library call.

which can be represented by a 4-tuple: (*ClassName*, *ReturnType*, *MethodName*, *ParameterTypes[]*). An edge connecting two nodes has three types: *Call Edge*, *Control Flow Edge* and *Data flow Edge* that represent a method invocation, flow of control and flow of data, respectively. The enriched call graph is generated in the following steps:

1. `GLACIATE` first performs an intra-procedural analysis [3] to extract method calls in each method and control flow relationships among those calls. At the end of this process, for every method, we have a graph that captures the control flow relationships among method calls made in the method.
2. Next, for each graph constructed in Step 1, `GLACIATE` examines the declared arguments and variables to extract data dependencies. According to each data dependency, a data flow edge is created between two nodes, and the data flow edge is labeled with the corresponding argument/variable through which the data dependency occurs.
3. Finally, we combine the graphs extracted in Steps 1 and 2, across all methods by adding edges corresponding to method invocations.

Each enriched call graph may have redundant methods. `GLACIATE` performs *local distortions* [21] to alter the graph topology (i.e., remove redundant methods), without changing the code's functionality. To remove redundant functions, `GLACIATE` first splits an enriched call graph into smaller pieces based on the local method <init>. It then removes the <init> method which has only one connected local method.

Flow Sequence Conversion. A flow sequence consists of a sequence of *vector*s, each of which has four elements (S_{from}, S_{to}, V_{in}, V_{out}), indicating that method S_{from} is the caller of method S_{to}, values V_{in} are input parameters of method S_{to}, and values V_{out} are returned parameters of method S_{to}. Note that V_{in} and V_{out} can be null to specify a method without any input parameters or return values.

Given an enriched call graph, `GLACIATE` extracts the corresponding flow sequences in three steps:

1. Following call edges, `GLACIATE` collects method invocations from the enriched call graph and constructs pairs in the form of (S_{from}, S_{to}).
2. Following data flow edges, `GLACIATE` inserts input values V_{in} and returned values V_{out} of each callee into the corresponding pair to construct a vector (S_{from}, S_{to}, V_{in}, V_{out}).
3. Following control flow edges, `GLACIATE` extracts the sequence of vectors. Note that we generate a flow sequence for each condition while processing the decision making statements (e.g., if-else, switch, break).

Learning Cluster Generation. `GLACIATE` computes the similarity between each enriched call graph and the other enriched call graphs, and groups the similar enriched call graphs to produce *learning clusters*. Details are shown as below.

Similarity Computation. Based on the app labels, GLACIATE first classifies enriched call graphs into five groups (Secure Group, Group 1, Group 2, Group 3, and Group 4), which correspond to secure authentication, authentication with flaw 1, authentication with flaw 2, authentication with flaw 3, and authentication with flaw 4, respectively.

Within each group, GLACIATE compares enriched call graphs mutually, and computes similarity scores by applying pairwise comparison [2]. Two enriched call graphs are deemed to be similar only if their flow sequences are similar. To check for this similarity, GLACIATE proceeds in the following steps:

1. GLACIATE constructs a pairwise comparison matrix to find the highest similarity score between two enriched call graphs. The flow sequences FS of an enriched call graph ECG_i are listed on top row of the pairwise comparison matrix, and the flow sequences FS' of another enriched call graph ECG_j are listed on the left hand column of the pairwise comparison matrix.
2. GLACIATE compares flow sequences of two enriched call graphs by extracting the longest common substring (LCS) [23] and fills the corresponding blank matrix cell with the LCS length L of two flow sequences.
3. From each column, GLACIATE extracts the cell with the highest value L^{max}. The column and row of each cell should be unique.
   If two cells from the same row are picked, GLACIATE then selects the next highest value L^{next} in each column and computes $L^{max} + L^{next}$ interlaced. GLACIATE chooses the pair (L^{max}, L^{next}) with the highest sum value.
4. Finally, GLACIATE computes the similarity score as $S_{Sim}(ECG_i, ECG_j) = \sum L^{max}$.

Group Clustering. Given the similarity scores, GLACIATE performs *agglomerative hierarchical clustering* [15], which works by measuring pairwise distances between data points and grouping the data points one by one on the basis of selecting nearest neighbours. GLACIATE uses the $ECGs$ as a set of data points and then applies the following steps to cluster them. We use the reciprocal of the similarity score as the distance between two $ECGs$.

1. Given the ECGs, GLACIATE first labels each ECG as a single cluster $C(ECG)$.
2. For two enriched call graphs ECG_1 and ECG_2, GLACIATE uses the reciprocal of the similarity score to denote the distance between them as $dist(1, 2)$.
3. Next, GLACIATE finds the closest pair of clusters $C(ECG)_m$ and $C(ECG)_n$ from those single clusters, according to $dist(m, n) = dist_{min}(i, j)$, where the minimum is over all pairs of clusters in the current clustering.
4. GLACIATE merges clusters $C(ECG)_m$ and $C(ECG)_n$ to form a new cluster, and repeats from Step 2 until all the data points are merged into a single cluster.
5. GLACIATE finally picks a distance threshold T_{dist} to cut the single cluster into several different clusters, each of which is a learning cluster, used to generate rule templates.

Table 1. Indicator instructions

Secure protocols	Instruction #	Indicator instructions
BPAP	1	java.net.PasswordAuthentication char[] getPassword
	2	java.net.Authenticator java.net.PasswordAuthentication requestPasswordAuthentication
SSL	3	javax.net.ssl.SSLSocketFactory java.net.Socket createSocket
	4	javax.net.ssl.SSLContext javax.net.ssl.SSLConext getInstance
	5	javax.net.ssl.SSLSession java.security.cert.Certificate[] getLocalCertificates
	6	javax.net.ssl.TrustManagerFactory javax.net.ssl.TrustManagerFactory getInstance
	7	java.Security.cert.X509Certificate void verify
	8	java.security.cert.X509Certificate: void checkValidity
	9	javax.net.ssl.HostnameVerifier boolean verify
Timestamp	10	java.lang.System long currentTimeMillis

3.2 Detection Rules Mining

`GLACIATE` learns a *rule template* from each learning cluster. A rule template consists of a set of *indicator instructions*, which specifies methods that are invoked by all enriched call graphs, and a *rule sequence*, which specifies a subsequence of vectors that is executed by all enriched call graphs.

To create a rule template from a learning cluster, `GLACIATE` executes an iterative pattern mining which captures higher-order features from flow sequences. A vector in a flow sequence, corresponding to a method invocation and a data flow, can be treated as a feature. We apply an algorithm to mine *closed unique iterative patterns* [22], which can capture all frequent iterative patterns without any loss of information. In each learning cluster, `GLACIATE` compares enriched call graphs and proceeds in the following steps:

1. `GLACIATE` observes the frequent vectors appeared in all enriched call graphs and creates a set of indicator instructions. We manually selected nine indicator instructions from the document provided by Android[5], which are listed in Table 1.

[5] Android Doc: https://developer.android.com/training/articles/security-ssl#java.

2. Starting from a frequent vector, `GLACIATE` creates a rule sequence. `GLACIATE` searches for the following vector that appears in every enriched call graph, and if found, includes it in the rule sequence. The rule sequence is created successfully only if its length is longer than a threshold min_{rule}. Step (2) is executed recursively until the rule sequence is closed (i.e., does not grow).
3. For each rule sequence, `GLACIATE` finally replaces all concrete identifier values (i.e., variables) with placeholders.

3.3 Flaws Detection

`GLACIATE` detects authentication flaws by selecting the most suitable template in two steps as follows. These two steps are iterated until no further vulnerable code segments are detected.

Authentication Identification. To detect whether there is any implemented password authentication protocols, `GLACIATE` checks for matches with the sets of indicator instructions. It compares flow sequences with each set of indicator instructions and computes how many indicator instructions in the set match. However, "noise" or unrelated vectors are present among the indicator instructions. The "noise" can correspond to unrelated method invocations, such as toString(), <init>, etc. In view of this, we decide that a flow sequence matches a set of indicator instructions if at least 80% of the indicator instructions are matched.

Rules Matching. There are likely to be multiple rule templates which match an enriched call graph. Instead of analyzing all the flow sequences of an enriched call graph, `GLACIATE` applies program slicing [28] to compare flow sequences with the corresponding matched rule templates one by one and in the following three steps.

1. `GLACIATE` first identifies where the indicator instructions are located.
2. Beginning with each indicator instruction, `GLACIATE` compares the vectors in the flow sequence FS with the vectors in the rule sequence RS by performing forward program slicing. If sequences in FS can be matched with sequences in RS, this enriched call graph will be labeled the same as RS, that is, secure, flaw 1, flaw 2, flaw 3, or flaw 4. Noting that FS may include some redundant vectors (i.e., redundant method invocations), FS and RS are matched if RS is a subsequence of FS.
3. `GLACIATE` proceeds to the next detection template which matched, and executes Step (2) until all matched rule templates have been analyzed.

4 Evaluation

In this section, we report the results of two experiments. The first experiment assesses the performance of `GLACIATE` and compares it with `MalloDroid` [10] and

SMV-Hunter [25], state-of-the-art tools for identifying flaws in the implementation of SSL/TLS validation in Android apps. MalloDroid is a semi-automated detection tool, which requires manually-defined templates. SMV-Hunter is an automatic detection tool that requires the manually generated inputs are accurate enough to trigger vulnerabilities accurately. Differently, GLACIATE is designed to detect violations in authentication code automatically, and as far as we are aware, there are no other tools that can learn rules and detect authentication flaws in this way. The second experiment demonstrates how GLACIATE automatically analyze a large collection of Android apps to gain further insights on the prevalence of authentication implementation flaws in these apps.

4.1 Assessment of GLACIATE

Dataset. We randomly collected 1,200 free apps from Google Play. 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*, and 200 apps from each category.

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.

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 whether implementations of password authentication protocols in apps followed the rules that we created. We first required team members to label apps independently. Then all members went through the labels together and discussed any 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 [11]. 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 protocols), and 1,087 authentication flaws were identified in 695 apps (Flaw 1: 284, Flaw 2: 736, Flaw 3: 67).

Experiment Design. We used 10-fold cross validation [17] to evaluate the effectiveness of GLACIATE. Furthermore, we compare GLACIATE with MalloDroid [10] and SMV-Hunter [25]. While detecting authentication flaws, we set $T_{dist} = 1.3$ to ensure that enriched call graphs in each cluster would be highly similar to each other, and $min_{rule} = 2$.

To assess the performance of GLACIATE, we generated an evaluation matrix of the precision, recall, and F1 metrics. Precision is for measuring how accurate our tool performs, recall reflects how many vulnerabilities are actually detected, and F1 is used to balance precision and recall.

Performance. For comparison, we applied the MalloDroid, SMV-Hunter and GLACIATE to the entire dataset. Since MalloDroid and SMV-Hunter only detect SSL/TLS-related flaws (i.e., flaw 2 in this paper), we limited GLACIATE to detect flaw 2 (736 flaws in total) in this test. From the results we computed the precision, recall and F1 over the entire dataset for each tool.

Table 2. Detection result: GLACIATE, MalloDroid, and SMV-Hunter

Flaw	GLACIATE		MalloDroid		SMV-Hunter	
	Detected	Correct	Detected	Correct	Detected	Correct
Flaw 2	751	686	214	201	627	572
Precision	91.3%		93.9%		91.2%	
Recall	93.5%		27.3%		77.7%	
F1	92.4%		42.3%		83.9%	

Table 2 shows the assessment results. GLACIATE correctly detects 686 out of 736 flaws, with precision, recall, and F1 values of 91.3%, 93.5%, and 92.4%, respectively. On the other hand, MalloDroid can only detect 201 flaws, achieving a recall of only 27.3%. SMV-Hunter successfully detects 572 SSL/TLS-related flaws with precision, recall and F1 values of 91.2%, 77.7%, and 83.9%. Though MalloDroid has fewer false positives, as evident from the marginally higher precision (i.e., 93.9% against 91.3%), GLACIATE detects about 2.4 times more flaw 2 than MalloDroid. Compared with SMV-Hunter, GLACIATE detects 20.2% more flaws and has a 1.2% better precision. This means that GLACIATE 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, GLACIATE also finds three new types of vulnerable TrustManagers, namely BlindTrustManager, InsecureTrustManager and AllTrustingTrustManager. Apps with these vulnerable TrustManagers suffer from flaw 2.

GLACIATE: **Further Analysis of Performance.** In comparing the detection performance of GLACIATE and MalloDroid, we find that MalloDroid fails to correctly analyze apps that implement authentications across different classes, which means MalloDroid is unable to analyze method invocation relationships and cannot extract inter-component communications in apps. Furthermore, comparing the results for GLACIATE and SMV-Hunter, SMV-Hunter relies on user inputs to trigger the recognition of authentication flaws. However, it is a challenging to generate accurate inputs to trigger the procedures.

GLACIATE did fail to analyze some apps. Since GLACIATE is 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 can be reconsidered when Soot is next upgraded[6].

4.2 GLACIATE: Large Scale Analysis of Password Authentication

For this analysis, we downloaded 16,387 free apps at random from Google Play and used our ground truth to build our detection model for further analysis. We first checked whether our collected apps implemented any password authentication protocols. In total, 13,747 apps were successfully analyzed, and 5,667 (41%) of them implemented BPAP. Further analyses were performed on those 5,667 apps. Apps failed to be analyzed by GLACIATE are unable to be decompiled by Soot.

Table 3. Secure password authentication protocols in Android apps

# of apps	Secure password authentication protocols
3,353	Only BPAP over SSL/TLS
804	Only timestamp-based password authentication
385	Both BPAP over SSL/TLS and timestamp-based password authentication

Based on the detection report generated by GLACIATE (see Table 3), we find that 4,542 apps establish secure password authentication protocol by using at least one protection protocol. Among the apps with at least one protection protocol, we observe that 3,738 implemented BPAP over SSL/TLS, which indicates that SSL/TLS is the most common protection mechanism in practice. We also identify 385 apps with both protections, i.e., BPAP over SSL/TLS and timestamp-based password authentication protocols. By further analyzing those apps with multiple password authentication protocols, we find that some apps implement multiple login schemes (e.g., Facebook login, Wechat login, Tencent login), and their developers import external authentication libraries directly to implement those login schemes. The library providers offer a variety of password authentication protocols[7].

The password authentication protocol is suppose to be securely implemented. To our surprise, A large portion of apps have flaws discussed in Sect. 2 in their authentication code (shown in Table 4). Only 1,562 apps in our dataset implemented secure password authentication protocols. GLACIATE reports that passwords in 1,125 apps are not been well-protected. For these apps with Flaw 1, we

[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.

[7] BPAP with SSL/TLS is nevertheless most used.

Table 4. Authentication flaws in Android apps

# of apps	Authentication flaws
1,125	Insecure Password Transmission (Flaw 1)
2,684	Insecure Server Connection (Flaw 2)
250	Repeatable Timestamp (Flaw 3)
1,562	No flaw

observe that some of them use MD5 hash functions with a constant salt, which is easy for attackers to find collision. However, most passwords are transmitted in plaintext over an insecure HTTP channel. As SSL/TLS is the most common mechanism used to protect BPAP, SSL/TLS-related flaw is also the most common one, i.e., flaw 2 (i.e., Insecure Server Connection). We also investigate whether apps have multiple flaws. In what follows we discuss further insights gained from this analysis.

Flaw 2: Insecure Server Connection. This is the most common implementation flaw presented in 2,684 apps; that is, nearly 47% of the apps with password authentication meet this authentication flaw. This result indicates that developers are security conscious and understand that secure communication (e.g., SSL/TLS) should be used for transmitting passwords. However, they seem to be unaware of the importance of validating certificates and hostnames of the server, and the consequences of accepting invalid certificates and mismatched servers, or they decide not to validate certificates and hostnames with the aim of improving the app's run-time performance.

Certificate Validation. In total, `GLACIATE` identifies 2,417 apps suffers the flaws of accepting invalid certificates. A certificate validation includes two aspects: signature validation and a certificate expiration check. The authentication code is insecure unless both checks are executed. Based on the trusted CAs provided by Android[8], we classify invalid certificates into certificates signed by invalid CAs, self-signed certificates, and expired certificates. Table 5 lists the number of apps with these types of certificate flaws. Those certificate validations are incomplete in that 1,298 apps only verify whether certificates are signed by valid CA but neglect to check whether they are self-signed or expired, and 185 apps only verify two of the necessary checks of certificate validity. Almost 35% of the apps with flaw 2 do not have any certificate validation at all.

Hostname Verification. 2,059 of apps with flaw 2 accept all hostnames. Comparing this result with the result of certificate checking, a smaller number of apps suffer from this, since more aspects are required to be checked when validating certificates, i.e., expiry date and signature.

[8] The list of trusted CAs can be found in https://www.digicert.com/blog/official-list-trusted-root-certificates-android/.

Table 5. Apps with incomplete certificate validation

# of apps	Certificate validations performed
1,298	Only implement one check, whether the certificates are signed by an invalid CA
54	Only implement two checks, whether the certificates are self-signed or signed by an invalid CA
131	Only implement two checks, whether the certificates are expired or signed by an invalid CA
934	None of the above (e.g., they do not implement any certificate verification)

Flaw 3: Repeatable Timestamp. Most apps with timestamp-based password authentication are securely implemented, but nevertheless 250 out of 804 apps used a repeatable timestamp.

Multiple Flaws. We also collected information about apps which were found to have multiple violations. For the apps that used both protection mechanisms, `GLACIATE` identified 37 apps suffering from two types of authentication flaws. Authentication code in 29 apps accept all certificates and generate repeatable timestamps. 8 of them implement the authentication protocol as accepting all host names and generating repeatable timestamps. Additionally, `GLACIATE` detected 15 apps that violates all the authentication requirements, that is, accepting all certificates and all hostnames, and use repeatable timestamps. These results suggest that the capability of analyzing multiple password authentication protocols in the same app is essential for a complete identification of vulnerabilities.

5 Related Work

In the following, we first discuss detection techniques that are rule-based and attack-based. We then discuss fully-automated approaches that use machine learning algorithms.

5.1 Rule-Based Techniques

Most existing techniques detect vulnerabilities by using pre-defined rules/templates [9,10,24,29]. `CRYPTOLINT` [9] detects cryptographic misuses in Android apps. According to the manually predefined cryptographic rules, `CRYPTOLINT` computes a super control flow graph for each app and uses program slicing to identify the violations. `MalloDroid` [10] is a detection tool for checking whether the SSL/TLS code in Android apps are potentially vulnerable to MITM attacks. By checking the network API calls and Internet permissions, `MalloDroid` determine 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 performing code analysis, HVLearn [24] is a black-box learning approach that infers certificate templates from the certificates with certain common names by using an automata 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.

Besides these static analysis techniques, some dynamic approaches have been proposed without analyzing the code [6,26]. Spinner [26] is a tool that uses a dynamic black-box detection approach to check certificate pinning vulnerabilities which may hide improper hostname verification and enable 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.

To address the limitations of dynamic analysis, some approaches use a hybrid analysis (i.e., static and dynamic analysis) [16,25]. SMV-Hunter [25] simulates user interactions and launches MITM attacks to detect SSL vulnerabilities. However, its detection performance relies on how well user inputs were created, and some vulnerabilities cannot be identified since they are not triggered by the MITM attacks.

Compared to these techniques, GLACIATE is a fully automated tool that does not require any manual effort. Instead of summarizing detection rules manually, we use machine learning to learn those rules automatically.

5.2 Attack-Based Techniques

Instead of using any rules/templates, some approaches launch attacks to locate vulnerabilities [5,7,8,31]. AUTHScope [31] 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. Instead of launching one attack, six different attack scenarios are launched by AndroSSL [7], 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.

5.3 Machine Learning Techniques

Manual effort involving is tedious, inefficient, and expensive. To address this drawback, machine learning is proposed to construct a fully automated detection approach.

`VulDeePecker` [20] and `SySeVr` [19] 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 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` can detect many types of vulnerabilities without any manual effort, one important requirement for the training dataset is that each code segment may include only one vulnerability.

The above detection approaches that use machine learning algorithms 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., CNN, decision tree, naive Bayes, SVM, and logistic regression) to build detection models. However the detection results were found to be poor.

6 Conclusion

In this paper, we proposed a novel end-to-end approach for the automatic detection of flaws in the implementation of authentication in mobile apps. The detection tool, `GLACIATE`, analyzes whether the secure password authentication protocols are correctly implemented in apps. `GLACIATE` first uses clustering and pattern mining techniques to learn rules automatically from a small training dataset, followed by a program analysis technique which uses these rules to detect flaws. `GLACIATE` automates the whole process so that it only needs few manual efforts to build a small labeled dataset and achieves a better detection accuracy. We assessed the detection accuracy of `GLACIATE` on a dataset of 16,387 real world Android apps. `GLACIATE` identifies 5,667 apps with secure password authentication protocols, but only 28% of them implemented the protocols correctly. We intend to make `GLACIATE` available as an open source tool that can contribute to the development of secure Android apps.

Acknowledgments. This work was partially supported by the Key Program of National Natural Science Foundation of China (Grant No. U1636217), the General Program of National Natural Science Foundation of China (Grant No. 61872237), the National Key Research and Development Program of China (Grant No. 2016YFB0801200).

References

1. Alghamdi, K., Alqazzaz, A., Liu, A., Ming, H.: 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, pp. 95–102. ACM (2018)

2. Barzilai, J.: Deriving weights from pairwise comparison matrices. J. Oper. Res. Soc. **48**(12), 1226–1232 (1997)
3. Burke, M., Cytron, R.: Interprocedural dependence analysis and parallelization, vol. 21. ACM (1986)
4. Canvel, B., Hiltgen, A., Vaudenay, S., Vuagnoux, M.: Password interception in a SSL/TLS channel. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 583–599. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_34
5. Chen, J., Jiang, J., Duan, H., Weaver, N., Wan, T., Paxon, V.: Host of troubles: multiple host ambiguities in http implementations. In: Proceedings of the 2016 ACM Conference on Computer and Communications Security (CCS), pp. 1516–1527. ACM (2016)
6. Chen, J., et al.: IoTFuzzer: discovering memory corruptions in IoT through app-based fuzzing. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS). Citeseer (2018)
7. Gagnon, F., Ferland, M.-A., Fortier, M.-A., Desloges, S., Ouellet, J., Boileau, C.: AndroSSL: a platform to test Android applications connection security. In: Garcia-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) FPS 2015. LNCS, vol. 9482, pp. 294–302. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30303-1_20
8. D'Orazio, C.J., Choo, K.K.R.: A technique to circumvent SSL/TLS validations on iOS devices. J. Future Gener. Comput. Syst. **74**, 366–374 (2017)
9. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in Android applications. In: Proceedings of the 2013 ACM Conference on Computer and Communications Security (CCS), pp. 73–84. ACM (2013)
10. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: 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 (CCS), pp. 50–61. ACM (2012)
11. Fleiss, J.L., Levin, B., Paik, M.C.: Statistical Methods for Rates and Proportions. Wiley, New York (2013)
12. Hubbard, J., Weimer, K., Chen, Y.: A study of SSL proxy attacks on Android and iOS mobile applications. In: Proceedings of IEEE 11th Consumer Communications and Networking Conference (CCNC), pp. 86–91. IEEE (2014)
13. Liu, J., Ma, J., Zhou, W., Xiang, Y., Huang, X.: Dissemination of authenticated tree-structured data with privacy protection and fine-grained control in outsourced databases. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11099, pp. 167–186. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98989-1_9
14. Juels, A., Triandopoulos, N., Van Dijk, M.E., Rivest, R.: Methods and apparatus for silent alarm channels using one-time passcode authentication tokens. US Patent 9,515,989 (2016)
15. Karypis, G., Han, E.H., Kumar, V.: Chameleon: hierarchical clustering using dynamic modeling. J. Comput. **32**(8), 68–75 (1999)
16. Koch, W., Chaabane, A., Egele, M., Robertson, W., Kirda, E.: Semi-automated discovery of server-based information oversharing vulnerabilities in Android applications. In: Proceedings of the 26th ACM International Symposium on Software Testing and Analysis (ISSTA), pp. 147–157. ACM (2017)
17. Kohavi, R., et al.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: IJCAI, Montreal, Canada, vol. 14, pp. 1137–1145 (1995)
18. Lamport, L.: Password authentication with insecure communication. J. Commun. ACM **24**(11), 770–772 (1981)

19. Li, Z., et al.: SySeVr: a framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756 (2018)
20. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)
21. Linkola, S., et al.: A feature-based call graph distance measure for program similarity analysis (2016)
22. Lo, D., Cheng, H., Han, J., Khoo, S.C., Sun, C.: Classification of software behaviors for failure detection: a discriminative pattern mining approach. In: Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 557–566. ACM (2009)
23. Ma, S., Thung, F., Lo, D., Sun, C., Deng, R.H.: VuRLE: automatic vulnerability detection and repair by learning from examples. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10493, pp. 229–246. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_13
24. Sivakorn, S., Argyros, G., Pei, K., Keromytis, A.D., Jana, S.: HVLearn: automated black-box analysis of hostname verification in SSL/TLS implementations. In: Proceedings of 2017 IEEE Symposium on Security and Privacy (SP), pp. 521–538. IEEE (2017)
25. Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., Khan, L.: SMV-hunter: large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS). Citeseer (2014)
26. Stone, C.M., Chothia, T., Garcia, F.D.: Spinner: semi-automatic detection of pinning without hostname verification. In: Proceedings of the 33rd ACM Annual Computer Security Applications Conference (ACSAC), pp. 176–188. ACM (2017)
27. Vallée-Rai, R. Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a Java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp. 214–224. IBM Corp. (2010)
28. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering (ICSE), pp. 439–449. IEEE Press (1981)
29. Xiong, B., Xiang, G., Du, T., He, J.S., Ji, S.: Static taint analysis method for intent injection vulnerability in Android applications. In: Wen, S., Wu, W., Castiglione, A. (eds.) CSS 2017. LNCS, vol. 10581, pp. 16–31. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69471-9_2
30. Yang, C.C., Yang, H.W., Wang, R.C.: Cryptanalysis of security enhancement for the timestamp-based password authentication scheme using smart cards. IEEE Trans. Consum. Electron. **50**(2), 578–579 (2004)
31. Zuo, C., Zhao, Q., Lin, Z.: AUTHScope: towards automatic discovery of vulnerable authorizations in online services. In: Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS), pp. 799–813. ACM (2017)