Are Android apps being Protected Well Against Attacks?

Abstract—**Authentication is the most pervasive mean for developers to protect users' private data against attacks while using mobile applications. Incorrect implementations of authentication cause users' accounts vulnerable to several attacks such as eavesdropping attacks, reply attacks, and man-in-the-middle attacks, and thus break the first line of defense in securing mobile services. To solve this problem, we design a system that learns patterns from authentication bugs, and identifies incorrect authentication implementations from mobile applications. By conducting a static analysis, our system extracts control and data dependencies for further pattern learning and utilizes a machine learning algorithm to build a classification model. To distinguish whether an application contains any authentication bugs, we take the unknown application as an input and recognize the vulnerable patterns. To evaluate the accuracy of our system, we collected 1,200 Android applications from the official Google Play store, representing a variety of categories. We compare our system with MalloDroid, a state-of-the-art tool for SSL/TLS authentication bugs detection. Our system successfully identifies 691 SSL/TLS authentication bugs with precision, recall, and F1 value as 52.75%, 93.89%, and 67.55%, respectively.**

Keywords: Android Security, Password Authentication, OTP Authentication.

I. Introduction

Nowadays, authentication becomes ubiquitous in our lives, not only applied to applications, but also involved in smart contracts on the blockchain system [1]. For the mobile devices, We rely on the applications to process business (e.g., email apps), social with others (e.g., whatsapp), and support our daily lives (e.g., MRT transport) [2]. For each user, the system sets up a private account for them, which contains access control to prevent from privacy leakage. When a user wants to retrieve any of her private data, an identity authentication is required. However, authentication in mobile devices can easily become vulnerable under certain authentication scenarios [3]. Therefore, we target on a security analysis of user authentication protocols implemented in mobile apps.

Generally, developers choose to implement single factor authentication for Android applications. Most of them use password authentication, which relies on a combination of username and password. A user sends a combination of username and password in plaintext to a server through a client application, and then the server replies with an authentication-acknowledgement if the received password is valid. However, this scheme is vulnerable to replay attacks if users' passwords are not being protected (i.e., transmitted in plaintext over an insecure network). For the other applications, they apply one-time password (OTP) authentication, which require servers to generate pseudo-random numbers for identity validation. In this scheme, a user first provides a valid mobile phone number or an Email account to the server. The server then generates a pseudo-random number and sends it to the user. The user is regarded as valid if and only if the correct pseudo-random number is submitted. During each time of authentication, the pseudo-random number should be random (i.e., unpredictable).

Nonetheless, implementing password authentication and OTP authentication correctly are not easy. Potential privacy leakage is caused because of the incorrect implementations [4]. For simple password authentication and OTP authentication, they are vulnerable to eavesdropping and reply attacks because users' passwords and the generated OTP values are transmitted in plaintext. To protect these authentication against attacks, a countermeasure, to build SSL/TLS secure connection between clients and servers, is proposed. This countermeasure is secure only if the server's certificate and hostname are checked; otherwise, the secure connection is still vulnerable to eavesdropping and reply attacks. Moreover, OTP values generated by servers **MUST** be unpredictable and unrepeatable.

Instead of analyzing cryptographic algorithms and the security of authentication protocols [5] , we aim to identify incorrect authentication code by analyzing Android apps. However, identifying incorrect implementation manually is tedious and inaccurate. Since each correct way of an authentication implementation follows a certain pattern, we design an automated tool to learn the certain pattern and identify the corresponding authentication bug without any involvement of human efforts. In advance, we define four types of authentication implementation: 1) secure implementation - it represents a correct authentication implementation; 2) insecure certificate check - it indicates an incorrect authentication that violates any requirements of certificate check; 3) insecure hostname check - it describes an incorrect authentication that accepts all hostnames; 4) insecure OTP value - it indicates an OTP value is either repeatable or predictable.

Given applications with the above labels, we first extract control and data dependencies of each app by analyzing its code. We then traverse the dependencies into a vector and combine all vectors to construct a training matrix. Each vector represents an application, including its dependencies and its label. To learn the pattern of each category, we leverage a machine learning algorithm (i.e., Long short-term memory) to build a detection model, which is further being used to identify incorrect authentication implementations.

In order to evaluate the performance of our system, we randomly collect 1,200 Android applications from official Google Play Store. These applications are chosen from six categories: Communication, Finance, Dating, Health & Fitness, Shopping, and Social networking. Since there is no existing ground truth with authentication bugs labeled, we manually analyzed these collected Android applications and labeled them as secure authentication, insecure certificate check, insecure hostname check, and insecure OTP value. Due to the lack of a tool comparable to our system, we benchmarked MalloDroid [6], a state-of-the-art SSL/TLS vulnerability detection tool, against our dataset. MalloDroid is a SSL/TLS certificate validation tool (and thus do not detect the other bugs that our system covers). We benchmarked our system over the dataset of SSL/TLS related authentication bugs and compared it with MalloDroid over this dataset. Our system successfully identifies 370 insecure certificate checks with precision, recall, and F1 of 92.66%, 78.70%, and 85.11%, respectively. It also recognizes 321 insecure hostname checks with precision, recall and F1 of 89.7%, 76.64%, and 82.64%, respectively. MalloDroid only detects 211 SSL/TLS related authentication bugs, containing 151 insecure certificate checks and 60 insecure hostname checks. Because MalloDroid labels all the potential vulnerable apps as vulnerable, it has a better detection precision, but a low recall (i.e., false positive).

Contributions: Overall, our contributions are as follows:

- A novel automated approach that is able to learn patterns of each category of authentication bug without requiring any manual efforts. It then checks whether password authentication and OTP authentication in an Android app are correctly implemented.
- A comparison of our system with a state-of-the-art tool, MalloDroid. Our system performs much better than MalloDroid with only a few data.

Organization: The rest of this paper is organized as follows. Section II provides background information on authentication schemes used in Android apps and their correct implementation. In Section III, we introduce how our system learns patterns of each category and how it detects authentication bugs. In Section IV, we report the accuracy of our system against our manually built ground-truth dataset and compare it with the accuracy of MalloDroid. Section V concludes the paper.

II. Backgrounds

In this section, we present the definitions about password authentication and one-time password authentication.

A. Authentication Schemes

A wide variety of authentication schemes are proposed to verify users' identities on mobile apps. These authentication schemes are constructed based on the principles of

- Something the user knows (e.g., passwords, patterns).
- Something the user has (e.g., hardware tokens).
- A biometric property of the user (e.g., fingerprints).

Consider the resource limitation and user's usability, authentication schemes in mobile devices are required to be simple and convenient that are acceptable by most users. The authentication scheme is designed depending on the sensitivity of applications.

By inspecting real-world applications, password authentication (i.e., using a combination of username and password) is mostly applied. For some apps accessing sensitive resources (e.g., personal contacts, SMS messages) or executing sensitive functions (e.g., money transferring), a multi-factor authentication scheme is involved. The multi-factor authentication represents an authentication using a combination of the above mentioned principles. Commonly, two-factor authentication (a combination of password authentication and one-time password authentication) is the most popular scheme implemented in mobile applications with sensitive functions accessing. We introduce both password authentication and one-time password authentication in details below.

B. Password Authentication Protocol

Password authentication protocol (PAP) [7] is a password-based authentication protocol. PAP is considered as a simple and weak authentication, which only relies on a password set created each user for identity verification. The user's identity is verified relying on a two-way handshake, which is illustrated in Figure 5. First, a user first sends a combination of username and password to the server. The server sends "accept" (if credentials are OK) or "reject" (otherwise).

PAP is simple to use and deploy because only a combination of username and password is required without any extra configurations. Hence, PAP is suitable for all types of network infrastructure. As the security of PAP is entirely relying on confidentiality and the strength of the password, PAP is the most vulnerable authentication scheme.

When passwords are transmitted in plaintext over the network, the simple PAP is vulnerable to man-in-the-middle attacks (MITMA) and brute force attacks. To protect PAP against attacks, SSL (Secure Sockets Layer)/TLS (Transport Layer Security) encryption is constructed, through which the combination of username and password is transmitted in ciphertext.

Steps to construct SSL/TLS encryption are shown in Figure 6. Prior to connecting a server, a user first sends a login request to the server. The server presents its certificate (i.e., server.cert) to the client. After validating the certificate from the server, the user sends his/her certificate (i.e., client.cert) to the server for verifying the client's identity. If both the server and the client are valid, the connection between them is started and encrypted. Because of the encrypted server-client connection, the user's password is transmitted in ciphertext, which makes password crack even harder. Nonetheless, constructing a secure SSL/TLS encryption is challenging and error-prone. A man-in-the-middle attack can be launched successfully, if SSL/TLS is implemented as follows:

- All certificates from servers (i.e., server.cert) are acceptable on the user-side. In other words, fake certificates, expired certificates, self-signed certificates are all acceptable.
- The identity of each server is not verified.

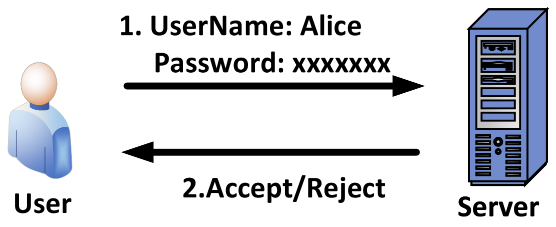

Fig. 1. Password Authentication Protocol

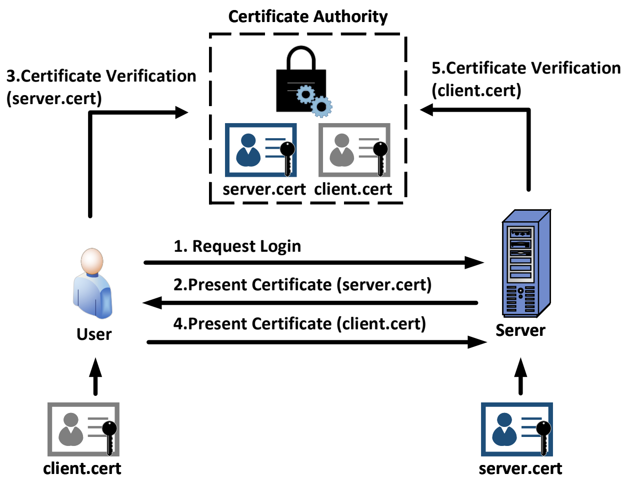
Certificate Authority
3.Certificate Verification (server.cert)
5.Certificate Verification (client.cert)
server.cert
client.cert
1. Request Login
2.Present Certificate (server.cert)
4.Present Certificate (client.cert)
User
Server
client.cert
server.cert

Fig. 2. SSL/TLS Encryption

C. *One-time Password Authentication Protocol*

To enhance the security of password authentication, two-factor authentication are applied. One-time password authentication (OTP authentication) is usually combined with PAP in mobile apps. OTP authentication relies on a one-time password token, which is a security hardware device or a software program that is capable of producing a single-use password. Two types of OTP authentication are proposed: HMAC-Based One-Time Password (HOTP) [8] and Time-Based One Time Password (TOTP) [9]. Referring to the OTP authentication presented in Figure 7, first, a user initializes a login request and sends it to the server. Then, the server creates an HMAC hash from a secret key and a counter An output value is further generated. Next, the server sends the generated output to the user. To fulfill the identity verification, the user provides the output value within a period of time.

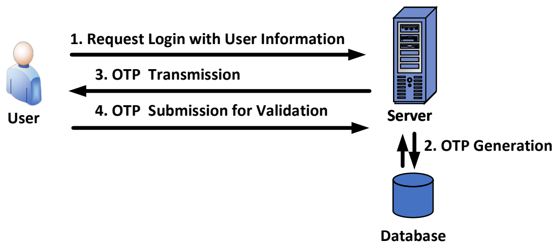
1. Request Login with User Information
3. OTP Transmission
4. OTP Submission for Validation
User
Server
2. OTP Generation
Database

Fig. 3. One-time Password Authentication

Theoretically, an OTP is more secure than a static password because it is not reusable and predictable. However, several pitfalls may be generated, which makes OTP authentication even weaker. An implemented OTP authentication is vulnerable to replay attacks and brute force attacks [10] when:

- It generates a sequence of predictable OTPs.
- It allows an OTP to be reusable.

III. System

In this section, we introduce our automated detection system to identify authentication vulnerabilities in mobile applications. Our system is illustrated in Figure 8, which consists of three components, including *feature matrix generation*, *model learning*, and *label prediction*. By taking mobile applications as inputs, our system identifies correlated functions and constructs a feature matrix based on those function calls. We further use a machine learning algorithm to generate a prediction model for further label prediction. Finally, a report including all identified vulnerabilities is provided.

A. *Feature Matrix Generation*

The component of feature matrix generation takes mobile applications as inputs. It takes the steps of *decompilation*, *auth locator*, and *feature construction* introduced as follows:

1) Decompilation.: Our system is built on top of *APKTool*[1], which is a reverse engineering tool. It relies on *baksmali*, a disassembler to convert Dalvik executable (.dex) files used by Dalvik Virtual Machine into SMALI files. SMALI files are similar to the application's original source for rebuilding the application, and each SMALI file represents a ".class" file, that is composed of class(es), function(s), variable(s).

2) Auth Locator.: Having SMALI files of each applications, all original source code information are able to be extracted including variable declarations, function calls, and dependency relationships. We further extract control and data dependencies in each application [11].

Control Dependency. It describes sequences of function executions. The sequentially executed functions are considered as control dependencies, i.e., the successor instruction is directly dependent on its previously executed function and indirectly dependent on all the predecessor instructions. Prior to obtaining control dependencies, we defined two types of statements for further analysis.

- **Conditional Transfer:** It describes the conditional statements including IF-ELSE, IF-THEN-ELSE, and SWITCH-CASE. If the given condition is satisfied, the statements in the current branch are executed. Otherwise, the statements declared in the other branch(es) are executed.
- **Stable Sequence:** It contains the remaining statements that are not included in the previous type. The execution is simply followed.

[1]APKTool: https://ibotpeaches.github.io/Apktool/install/

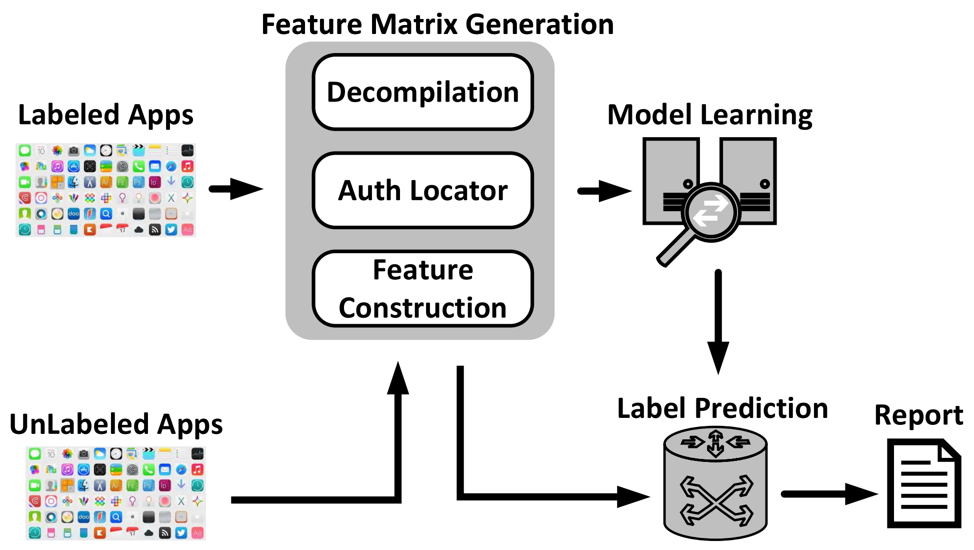

Fig. 4. Workflow of our vulnerability detection system

To derive control dependencies, our system monitors the execution information of function calls. Each execution information is represented as a 4-tuple: (*Function Name*, *Return type*, *argument type*, *argument name*). Depending on the type of each statement, our system selects the following execution path. We sequentially extracts the following function calls if the stable sequence is identified. When a conditional transfer is caught, our system generates N branches of execution by duplicating the current execution sequence. N is chosen depending on how many branches are created. For IF-ELSE and IF-THEN-ELSE, $N = 2$ and $N = 3$, respectively. If there is a SWITCH-CASE, N is chosen based on the number of CASE statements. Give each execution sequence, we sequentially explore the following function calls starting from the current branch.

Authentication Function Identify. To achieve an efficient dependency analysis, we retrieve the function calls that are relevant to authentication implementations. By analyzing the official Android documents provided for developers[2], we observe that certain functions are required to be called to implement password authentication in Android apps. Hence, we manually obtain these required functions from the official Android documents and use these functions as reference functions to identify password authentication. Five reference functions are listed in Table I.

Function 1 and 2 are related to obtain the password set for password authentication, which are required while implementing password authentication. To set up the SSL/TLS protocol, either function 3 or 4 is called for certificate verification and function 5 is implemented to recognize a valid hostname. To identify the dependency that is relevant to password authentication, we match the reference functions with all the execution information (i.e., executed functions) in control flow

[2]https://developer.android.com/training/id-auth

TABLE I
REFERENCE FUNCTIONS

	Function Names
1	Ljava/net/PasswordAuthentication;->getPassword
2	Ljava/net/Authenticator;->requestPasswordAuthentication
3	Ljava/Security/cert/X509Certificate;->verify
4	Ljava/security/cert/X509Certificate;->checkValidity
5	Ljavax/net/ssl/HttpsURLConnection;->setHostnameVerifier

sequences. An app is labeled as authentication correlated if any reference function is matched. Starting from the matched function, we extract authentication-related dependencies by extracting all the other correlated functions and variables based on the control and data dependencies.

Data Dependency. Execution information extracted from control dependencies are insufficient because they only introduce the execution sequences of functions without the involvement of variables. Unlike Java source code, SMALI code are bytecode that registers (e.g., V1, V2) are used to describe variables and commands are for demonstrating operations. Thus, the original functions and variable names are removed, which causes that the inputs of some functions (i.e., argument names in the execution information) might be the same.

Since one register might be used for multiple times by different function calls, it is time-consuming to extract all data flow information that are related to a register for each function call. We rely on the located reference functions and start from the involved arguments and construct the data dependencies, which illustrate the flow of variables. The flow of each involved variable (i.e., register) is stored in the format of (*register*, *type*, *value*). Starting from each function call, we first recognize the registers utilized in the function and then apply backward program slicing [12]. We determine where a variable is declared and what value it contains. Each track of backward program slicing terminates when the following conditions are met:

- when a constant value or string is assigned to the register;
- when the register is recognized as the return value of another function call;
- when the initial function is identified.

3) Feature Construction: As the authentication-related dependencies are demonstrated as a dependency graph, we traverse the dependency graph into a vector. We list the starting points at the row and the end point at the column. Suppose that A depends on B, we demonstrate such a dependency as $\{A, B\} = 1$; otherwise, $\{A, B\} = 0$. We finally combine the vectors of all input apps to construct a feature matrix. According to the label of each apps (i.e., secure authentication, insecure certificate check, insecure hostname check, insecure OTP value), each vector is labeled as 0, 1, 2, 3, respectively.

B. Model Learning

The goal of the model learning is to learn a discriminative model that identifies apps containing authentication bugs. Our system takes as input the generated feature matrix and then relies on the model learning phase to learn some characteristics of each category from the given feature values of the scripts belonging to the three categories. As features for training are represented in vectors (i.e., the format of strings), we choose to apply Long short-term memory (LSTM) [13], which is able to process an entire sequence of data.

C. Model Prediction

The label prediction process takes as input the discriminative model learned by the model learning phase. To identify bugs in the unlabeled apps, we construct control and data dependencies of each app using the feature matrix generation component. Similarly, the app is decompiled by *Apktool*, and then we identify the authentication related function calls and construction the dependency graph. The dependency graph is finally traversed into a feature vector, which is taken as input of the label prediction phase. The discriminative model would assign the likelihoods of the vector to belong to each of the four categories, nameyly, secure authentication, insecure certificate check, insecure hostname check, and insecure OTP value. The category with the highest likelihood would be outputted as the predicted label for the app. This step is performed as a natural extension of the model learning part.

IV. Evaluation

In this section, we evaluate the effectiveness of our system by comparing it with a state-of-the-art tool, MalloDroid [6].

A. Experiment Setup

1) Dataset: Since there is no existing dataset that contains labeled applications with authentication bugs, we built the ground-truth by ourselves. We collected 1,200 free applications from official Google play Store [14]. Applications are selected from six categories that are highly relevant to authentication, that is, Communication, Dating, Finance, Health & Fitness, Shopping, and Social Networking. From each category, we chose the top 200 applications. To build the ground-truth, we asked a team of researchers with two postdoctoral research fellows and one Ph.D student to manually analyze the collected applications and checked whether the password authentication scheme is securely implemented in each application. All the researchers have more than seven years of Java programming experience. First, researchers were required to analyze these applications independently. Then, we went through their annotations together and discussed the applications that are labeled differently. A final agreement would be made for applications with different labels. Finally, 1,205 implementations of password authentication protocols were identified in 742 Android applications, in which some applications implemented multiple protocols. Among them, 284 applications used password authentication with protection, while 736 applications implemented the protection of SSL/TLS incorrectly. While analyzing OTP authentication, we identified 323 applications with OTP authentications. We further conducted bug detection of OTP authentication on these applications.

2) Setup: We performed a 10-fold cross validation to run the experiment. First, the dataset was split into ten folds. Each fold contained around 70 vulnerable applications. Because an application might have multiple authentication flaws, we tried to split them averagely into different groups but the number of each type of authentication flaw in each group cannot be exactly the same. The experiment were run for ten times. During each time, one unique group was selected as a test group and the other nine groups were used as the training group. We then computed experiment results on average.

3) Evaluation Metrics: To assess the effectiveness of our system, we used precision, recall, and F1 as the evaluation metrics, shown in Eq. 1, Eq. 2, and Eq. 3, respectively.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

B. Performance

To assess the performance of our system, we count the number of authentication bugs that are correctly detected by our system. The results are illustrated in Table II: our system detected 691 authentication bugs. And it achieves the effectiveness of precious, recall, and F1 at 52.75%, 93.89%, and 67.55%. In comparison, MalloDroid achieves a higher precision than that of our system because it labels all the potential bugs as authentication bugs, which also causes low recall (i.e., high false positive). While considering both true positives and false positives, our system performs better than MalloDroid with F1 as 67.55%. Note that different from our system, MalloDroid only verified the implementation

TABLE II
DETECTION RESULTS

Authentication Bugs	Our system					MalloDroid				
	Detected	Correct	Precision	Recall	F1	Detected	Correct	Precision	Recall	F1
Certificate	742	370	92.66%	78.70%	85.11%	152	151	99.34%	39.22%	56.24%
Hostname	568	321	89.67%	76.64%	82.64%	62	60	80.64%	14.24%	24.21%
Precision	52.75%					93.93%				
Recall	93.89%					27.31%				
F1	67.55%					42.32%				

correctness of SSL/TLS. Therefore, we only compares our system with MalloDroid on identifying the authentication bug of SSL/TLS. Obviously, our system performs much better than MalloDroid.

We further counted the number of applications that use predictable OTP values for authentication and accept consumed OTP values. In total, our system successfully identified 26 applications that generate repeated OTP values. It describes the situation that an OTP value is still repeated for multiple times once it is being consumed. We also detected 39 applications that accept consumed OTP values.

V. Conclusion

In this paper, we assess the correctness of implemented authentication protocols in Android applications. Targeting on both password authentication protocol and one-time password authentication protocol, we study secure ways to protect password authentication and OTP authentication against attacks. Based on these secure ways, we develop a system to identify whether those authentication implementation follow the requirements. Our system extracts control and data dependencies of an application and relies on a machine learning algorithm to learn the characteristics of each type of authentication bugs. In total, we collect 1,200 Android applications and create the ground truth by ourselves. Comparing with a state-of-the-art tool, MalloDroid, our system performs much better than MalloDroid, which achieves precision, recall, and F1 with 52.75%, 93.89%, and 67.55%. Not only the authentication between mobile applications is vulnerable, but also authentication between mobile applications and IoT devices [15]. We will further analyze whether the authentication of IoT devices are secure.

References

[1] Y. Yu, Y. Zhao, Y. Li, X. Du, L. Wang, and M. Guizani, "Blockchain-based anonymous authentication with selective revocation for smart industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 5, pp. 3290–3300, 2019.

[2] 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.

[3] K. Ren, Z. Qin, and Z. Ba, "Toward hardware-rooted smartphone authentication," *IEEE Wireless Communications*, vol. 26, no. 1, pp. 114–119, 2019.

[4] Y. Yu, Y. Li, J. Tian, and J. Liu, "Blockchain-based solutions to security and privacy issues in the internet of things," *IEEE Wireless Communications*, vol. 25, no. 6, pp. 12–18, 2018.

[5] J.-h. Lee, "Secure authentication with dynamic tunneling in distributed ip mobility management," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 38–43, 2016.

[6] 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.

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

[8] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen, "Hotp: An hmac-based one-time password algorithm," *The Internet Society, Network Working Group. RFC4226*, 2005.

[9] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "Totp: Time-based one-time password algorithm," *Internet Request for Comments*, 2011.

[10] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert, "Sms-based one-time passwords: attacks and defense," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 150–159.

[11] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 711–722.

[12] D. W. Binkley and K. B. Gallagher, "Program slicing," in *Advances in Computers*. Elsevier, 1996, vol. 43, pp. 1–50.

[13] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[14] G. Play, https://play.google.com/store.

[15] S. Wang, J. Wang, and Z. Yu, "Privacy-preserving authentication in wireless iot: Applications, approaches, and challenges," *IEEE Wireless Communications*, vol. 25, no. 6, pp. 60–67, 2018.

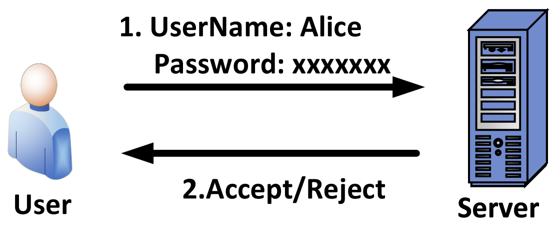

Fig. 5. Password Authentication Protocol

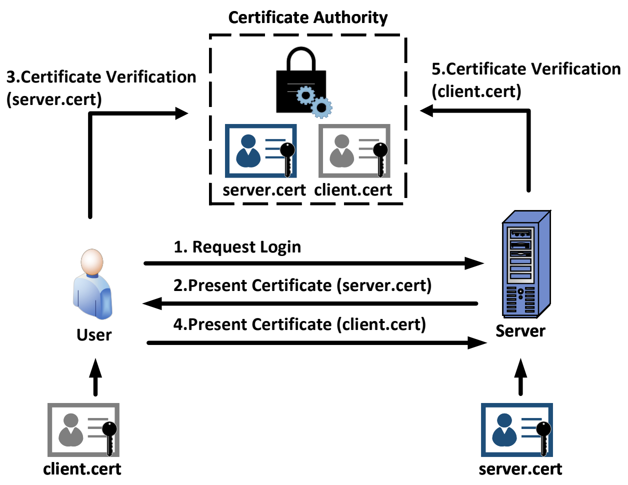
Certificate Authority
3.Certificate Verification (server.cert)
5.Certificate Verification (client.cert)
server.cert
client.cert
1. Request Login
2.Present Certificate (server.cert)
4.Present Certificate (client.cert)
User
Server
client.cert
server.cert

Fig. 6. SSL/TLS Encryption

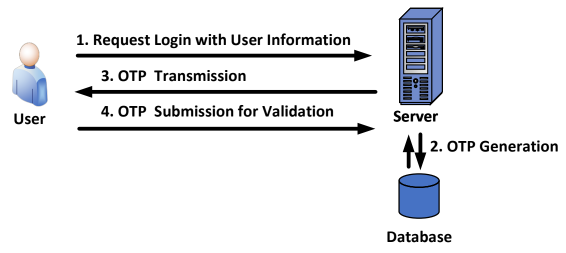
1. Request Login with User Information
3. OTP Transmission
4. OTP Submission for Validation
User
Server
2. OTP Generation
Database

Fig. 7. One-time Password Authentication

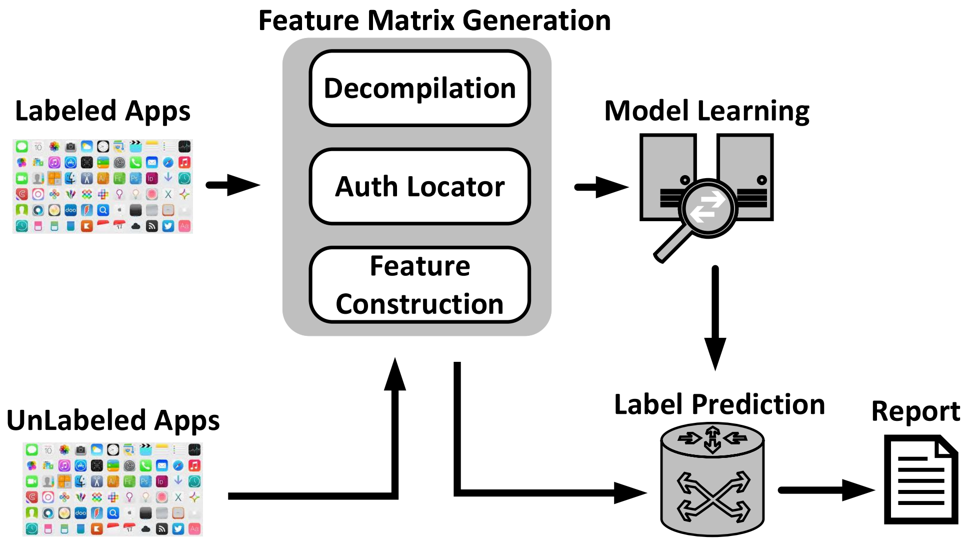

Fig. 8. Workflow of our vulnerability detection system