Are Android Apps Being Protected Well Against Attacks?

Siqi Ma, Yang Liu, and Surya Nepal

Abstract

Authentication is the most pervasive means for developers to protect users' private data against attacks while using mobile applications. Incorrect implementations of authentication make 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 1200 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 bug detection. Our system successfully identifies 691 SSL/TLS authentication bugs with precision, recall, and F1 value as 52.75, 93.89, and 67.55 percent, respectively.

Introduction

Nowadays, authentication has become ubiquitous in our lives, not only applied to applications, but also involved in smart contracts in the blockchain system [1]. For mobile devices, we rely on applications to process business (e.g., email apps), socialize 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, which contains access control to prevent 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 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 requires 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. For each 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 by incorrect implementations [4]. Simple password authentication and OTP authentication 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 authentication implementation follows a certain pattern, we design an automated tool to learn that pattern and identify the corresponding authentication bug without any involvement of human effort. In advance, we define four types of authentication implementation:

1. Secure implementation: a correct authentication implementation
2. Insecure certificate check: an incorrect authentication that violates any requirements of certificate check;
3. Insecure hostname check: an incorrect authentication that accepts all hostnames
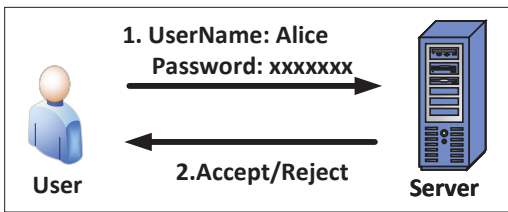
Siqi Ma is with the University of Queensland; Yang Liu is with Xidian University; Surya Nepal is with Data61, CSIRO.

FIGURE 1. Password Authentication Protocol.

4. Insecure OTP value: an OTP value that 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, LSTM) to build a detection model, which is further used to identify incorrect authentication implementations.

In order to evaluate the performance of our system, we randomly collect 1200 Android applications from the 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 Secure Sockets Layer (SSL)/Transport Layer Security (TLS) vulnerability detection tool, against our dataset. MalloDroid is an SSL/TLS certificate validation tool (and thus does 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 percent, respectively. It also recognizes 321 insecure hostname checks with precision, recall, and F1 of 89.7, 76.64, and 82.64 percent, 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 better detection precision but 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 effort. 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 article is organized as follows. We provide background information on authentication schemes used in Android apps and their correct implementation. We introduce how our system learns patterns of each category and how it detects authentication bugs. We report the accuracy of our system against our manually built ground-truth dataset and compare it with the accuracy of MalloDroid. We then conclude the article.

BACKGROUND

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

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)

Considering the resource limitation and user's usability, authentication schemes in mobile devices are required to be simple and convenient to be acceptable to 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. 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 function access. We introduce both password authentication and one-time password authentication in detail below.

PASSWORD AUTHENTICATION PROTOCOL

Password Authentication Protocol (PAP) [7] is a password-based authentication protocol. PAP is considered to be a simple and weak authentication, which only relies on a password set created by each user for identity verification. The user's identity is verified relying on a two-way handshake, which is illustrated in Fig .1. First, a user sends a combination of username and password to the server. The server sends "accept" (if credentials are okay) 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 entirely relies 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 (MITMAs) and brute force attacks. To protect PAP against attacks, SSL/TLS encryption is constructed, through which the

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 entirely relies on confidentiality and the strength of the password, PAP is the most vulnerable authentication scheme.
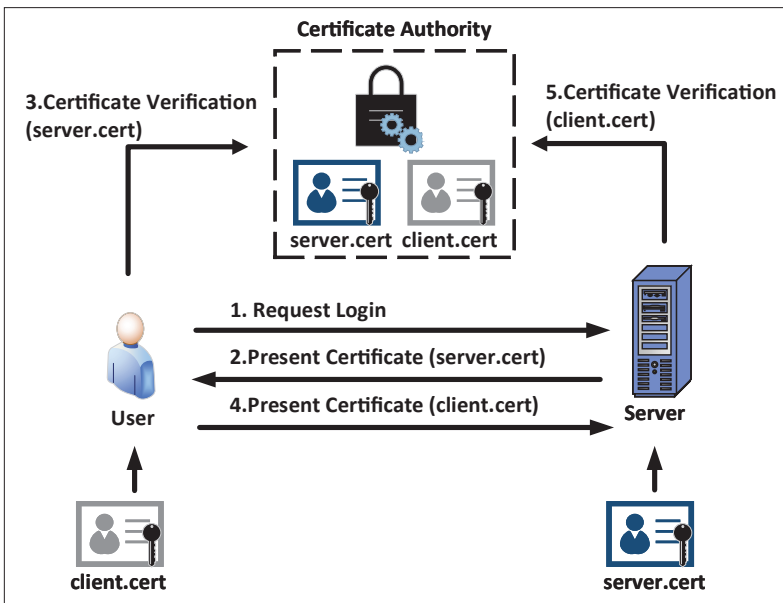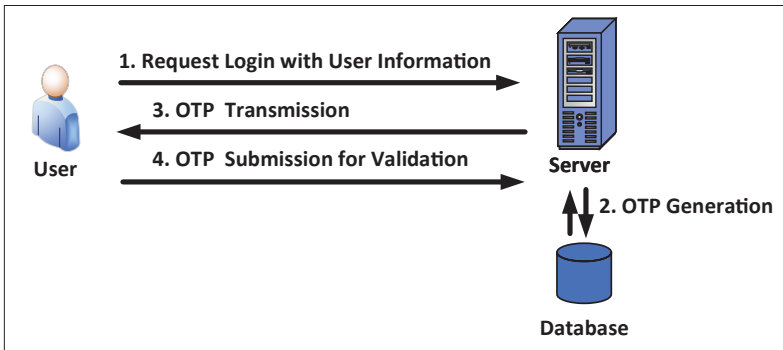
FIGURE 2. SSL/TLS encryption.

FIGURE 3. One-time password authentication.

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 Fig. 3, 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. 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 [1] when:

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

SYSTEM

In this section, we introduce our automated detection system to identify authentication vulnerabilities in mobile applications. Our system is illustrated in Fig. 4, and consists of three components: *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.

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.

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 a 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, which is composed of class(es), function(s), and variable(s).

Auth Locator: Having the SMALI file of each application, all original source code information is 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, that is, 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 define 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

combination of username and password is transmitted in ciphertext.

Steps to construct SSL/TLS encryption are shown in Fig. 2. 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 cracking even harder. Nonetheless, constructing a secure SSL/TLS encryption is challenging and error-prone. An MITMA 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, and self-signed certificates are all acceptable.
- The identity of each server is not verified.

ONE-TIME PASSWORD AUTHENTICATION PROTOCOL

To enhance the security of password authentication, two-factor authentication is applied. One-time password (OTP) authentication is usually combined with PAP in mobile apps. OTP authentication relies on a one-time password token,

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.

To derive control dependencies, our system monitors the execution information of function calls. Each piece of 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 extract 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. Given each execution sequence, we sequentially explore the following function calls starting from the current branch.

Authentication Function Identify. To achieve 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 must 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 1.

Functions 1 and 2 are related to obtain the password set for password authentication, which is 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 sequences. An app is labeled 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 is bytecode where registers (e.g., V1, V2) are used to describe variables and commands used for demonstrating operations. Thus, the original functions and variable names are removed, which means that the inputs of some functions (i.e., argument names in the execution information) might be the same.

Since one register might be used multiple times by different function calls, it is time-consuming to extract all data flow information 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
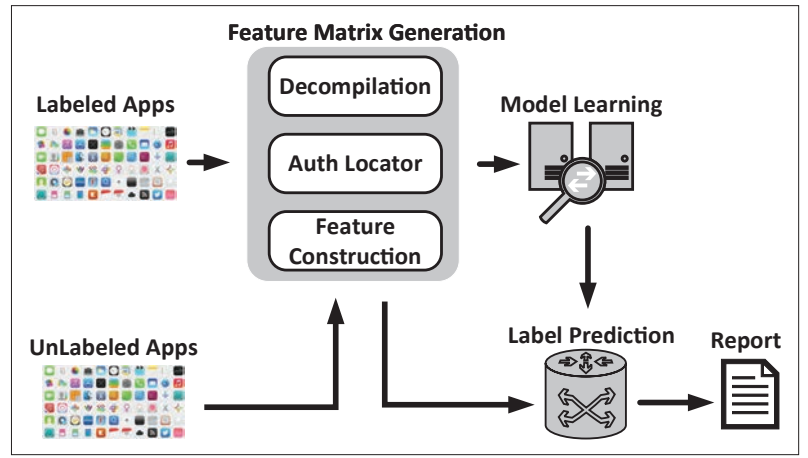

FIGURE 4. Workflow of our vulnerability detection system.

	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

TABLE 1. Reference functions.

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

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 app (i.e., secure authentication, insecure certificate check, insecure hostname check, insecure OTP value), each vector is labeled as 0, 1, 2, 3, respectively.

MODEL LEARNING

The goal of 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 LSTM [13], which is able to process an entire sequence of data.

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 Recall F1	52.75% 93.89% 67.55%					93.93% 27.31% 42.32%				

TABLE 2. Detection results.

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 the 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 construct 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, namely, secure authentication, insecure certificate check, insecure hostname check, and insecure OTP value. The category with the highest likelihood would be output as the predicted label for the app. This step is performed as a natural extension of model learning.

EVALUATION

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

EXPERIMENT SETUP

Dataset: Since there is no existing dataset that contains labeled applications with authentication bugs, we built the ground truth ourselves. We collected 1200 free applications from the official Google Play Store [14]. Applications are selected from six categories that are highly relevant to authentication: 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 post-doctoral research fellows and one Ph.D. student to manually analyze the collected applications and check whether the password authentication scheme is securely implemented in each application. All the researchers have more than seven years of Java programming experience. First, the researchers were required to analyze these applications independently. Then we went through their annotations together and discussed the applications that were labeled differently. A final agreement would be made for applications with different labels. Finally, 1205 implementations of password authentication protocols were identified in 742 Android applications, in which some applications implemented multiple protocols. Among them, 284 applications used PAP, while 736 applications implemented the protection of SSL/TLS incorrectly. While analyzing OTP authentication, we identified 323 applications with OTP authentication. We further conducted bug detection of OTP authentication on these applications.

Setup: We performed a 10-fold cross validation to run the experiment. First, the dataset was split into 10 folds. Each fold contained around 70 vulnerable applications. Because an application might have multiple authentication flaws, we tried to split them evenly into different groups, but the number of each type of authentication flaw in each group cannot be exactly the same. The experiment was run 10 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.

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.

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

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

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

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 2: our system detected 691 authentication bugs. And it achieves the effectiveness of precision, recall, and F1 at 52.75, 93.89, and 67.55 percent. In comparison, MalloDroid achieves 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 at 67.55 percent. Note that different from our system, MalloDroid only verified the implementation correctness of SSL/TLS. Therefore, we only compare 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 in which an OTP value is still repeated multiple times once it is being consumed. We also detected 39 applications that accept consumed OTP values.

Conclusion

In this article, we assess the correctness of implemented authentication protocols in Android applications. Targeting 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 methods, 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 1200 Android applications and create the ground truth by ourselves. Compared to a state-of-the-art tool, MalloDroid, our system performs much better, achieving precision, recall, and F1 with 52.75, 93.89, and 67.55 percent. Not only is the authentication between mobile applications vulnerable, but also authentication between mobile applications and IoT devices [15]. We will further analyze whether the authentication of IoT devices is secure.

References

[1] Y. Yu *et al.*, "Blockchain-Based Anonymous Authentication with Selective Revocation for Smart Industrial Applications," *IEEE Trans. Industrial Informatics*, vol. 16, no. 5, 2019, pp. 3290–3300.

[2] S. Ma *et al.*, "Finding Flaws from Password Authentication Code in Android Apps," *Euro. Symp. Research in Computer Security*, Springer, 2019, pp. 619–37.

[3] K. Ren, Z. Qin, and Z. Ba, "Toward Hardware-Rooted Smartphone Authentication," *IEEE Wireless Commun.*, vol. 26, no. 1, Feb. 2019, pp. 114–19.

[4] Y. Yu *et al.*, "Blockchain-Based Solutions to Security and Privacy Issues in the Internet of Things," *IEEE Wireless Commun.*, vol. 25, no. 6, 2018, pp. 12–18.

[5] J.-h. Lee, "Secure Authentication with Dynamic Tunneling in Distributed IP Mobility Management," *IEEE Wireless Commun.*, vol. 23, no. 5, 2016, pp. 38–43.

[6] S. Fahl *et al.*, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in) Security," *Proc. 2012 ACM Conf. Computer and Commun. Security*, 2012, pp. 50–61.

[7] L. Lamport, "Password Authentication with Insecure Communication," *Commun. ACM*, vol. 24, no. 11, 1981, pp. 770–72.

[8] D. M'Raihi *et al.*, "Hotp: An HMAC-Based One-Time Password Algorithm," The Internet Society, Network Working Group RFC 4226, 2005.

[9] D. M'Raihi *et al.*, "Totp: Time-based Onetime Password Algorithm," Internet Request for Comments, 2011.

[10] C. Mulliner *et al.*, "SMS-Based One-Time Passwords: Attacks and Defense," *Int'l. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2013, pp. 150–59.

[11] S. Ma *et al.*, "CDREP: Automatic Repair of Cryptographic Misuses in Android Applications," *Proc. 11th ACM on Asia Conf. Computer and Commun. Security*, 2016, pp. 711–22.

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

[13] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM Neural Networks for Language Modeling," *13th Annual Conf. Int'l. Speech Commun. Assn.*, 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 Commun.*, vol. 25, no. 6, 2018, pp. 60–67.

Biographies

SIQI MA (siqi.ma@csiro.au) is a lecturer at the University of Queensland. Her research work mainly focuses on software security including mobile apps and IoT firmware. She has a number of publications in top conferences and journals. She has been on the program committees of software and cybersecurity conferences such as Automated Software Engineering (ASE) and a reviewer of journals such as *IEEE Transactions on Information Forensics*.

YANG LIU (xdly9491@gmail.com) received his Bachelor's and Master's degrees from Xidian University. He is currently working toward a Ph.D study in computer science and technology at Xidian University. He works in the area of privacy preservation.

SURYA NEPAL (surya.nepal@csiro.au) is currently a senior principal research scientist with Data61, CSIRO. He currently leads the distributed systems security group. His main research focus is on the development and implementation of technologies in the area of distributed systems (including cloud, IoT, and edge computing) and social networks, with a specific focus on security, privacy, and trust. He has over 200 peer reviewed publications to his credit. He is currently a Theme Leader of the Cybersecurity Cooperative Research Centre (CRC), a national initiative in Australia. He holds a conjoint faculty position at the University of New South Wales and an honorary professor position at Macquarie University. He has co-edited three books, including *Security, Privacy, and Trust in Cloud Systems* (Springer), and coinvented three patents. He is a member of the Editorial Boards of *IEEE Transactions on Service Computing*, *ACM Transactions on Internet Technology*, and *Frontiers of Big Data — Security, Privacy, and Trust*.

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.