June 2019

Authentication and SQL-Injection Prevention Techniques in Web Applications

Cagri Cetin
University of South Florida, cagricetin@mail.usf.edu

Authentication and SQL-Injection Prevention Techniques in Web Applications

by

Cagri Cetin

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Dmitry Goldgof, Ph.D.
Yao Liu, Ph.D.
Sean Barbeau, Ph.D.
Kaiqi Xiong, Ph.D.

Date of Approval:
April 25, 2019

Keywords: Authentication methods, Formal verification, SQL-Injection attacks, GitHub analysis, Prepared statements

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation addresses the top two "most critical web-application security risks" by combining two high-level contributions [1].

The first high-level contribution introduces and evaluates collaborative authentication, or coauthentication, a single-factor technique in which multiple registered devices work together to authenticate a user [2, 3, 4]. Coauthentication provides security benefits similar to those of multi-factor techniques, such as mitigating theft of any one authentication secret, without some of the inconveniences of multi-factor techniques, such as having to enter passwords or biometrics. Coauthentication provides additional security benefits, including: preventing phishing, replay, and man-in-the-middle attacks; basing authentications on high-entropy secrets that can be generated and updated automatically; and availability protections against, for example, device misplacement and denial-of-service attacks. Coauthentication is amenable to many applications, including m-out-of-n, continuous, group, shared-device, and anonymous authentications. The principal security properties of coauthentication have been formally verified in ProVerif, and implementations have performed efficiently compared to password-based authentication.

The second high-level contribution defines a class of SQL-injection attacks that are based on injecting identifiers, such as table and column names, into SQL statements [5]. An automated analysis of GitHub shows that 15.7% of 120,412 posted Java source files contain code vulnerable to SQL-Identifier Injection Attacks (SQL-IDIAs). We have manually verified that some of the 18,939 Java files identified during the automated analysis are indeed vulnerable to SQL-IDIAs, including deployed Electronic Medical Record software for which SQL-IDIAs enable discovery of confidential patient information. Although prepared statements are the

standard defense against SQL injection attacks, existing prepared-statement APIs do not protect against SQL-IDIAs. This dissertation therefore proposes and evaluates an extended prepared-statement API to protect against SQL-IDIAs.

CHAPTER 1

INTRODUCTION

Web applications have become increasingly popular and ubiquitous with the popularity of mobile and Internet of Things devices. Web applications are also common targets of attacks, through injection, broken authentication, sensitive data exposure, cross-site scripting, and related vectors [1]. The top two "most critical" web-application vulnerabilities are

1. injection attacks and

2. broken authentication [1].

This dissertation addresses these top two vulnerabilities by (1) introducing and evaluating a novel authentication technique called collaborative authentication, and (2) defining and evaluating the prevalence of a class of SQL-Injection attacks that are based on injecting identifiers (e.g., table and column names). This chapter [1] introduces the collaborative authentication and a new class of SQL-Injection attacks.

1.1 Collaborative Authentication

With the growth of the Internet of Things, ubiquitous computing, and wearable, edible, and implantable devices, the overwhelming majority of adults may soon have multiple personal smart devices accessible at all times, all of which can be registered and used to authenticate. For example, to log in to a website, open a door, or start an engine, *two* of

[1] Parts of this chapter is published in ACM Symposium on Applied Computing [4] and IEEE Conference on Communications and Network Security [5]. Permissions to use these materials are provided in Appendix A.

a user's registered devices, perhaps a smartphone and smartwatch, might participate in the authentication. A gate or garage door might authenticate a request to open by requiring participation from both a registered car and a registered smartphone; then stealing only the car, or only the phone, would be insufficient for opening the door.

Even today many people only authenticate to certain services when multiple of their devices are present. For example, a user U may log in to banking services only from a certain PC while in the presence of U's smartphone. In this case the banking service could register these two user devices to U and require their participation in every authentication of U. Because the PC and smartphone are separate and heterogeneous, successfully stealing or otherwise attacking one device does not imply a successful attack on the other device. It is therefore of value to protect against attacks on only one of the two user devices.

We call this single-factor technique, in which multiple devices collaborate to authenticate a user, *coauthentication* [2, 3, 4, 6, 7]. The user devices collaborate through cryptographic protocols, such that an authenticator receives message(s) proving that all required user devices approve the authentication. Attackers who steal only one of the user devices cannot authenticate, because the unstolen device will not approve the authentication.

Benefits of coauthentication include protecting against the compromise of authentication secrets (cryptographic keys); preventing phishing, replay, and man-in-the-middle attacks; basing authentication on high-entropy secrets that can be generated and updated automatically; avoiding the inconveniences of factors like passwords and biometrics; implementing advanced authentication functionalities, including m-out-of-n, continuous, group, shared-device, and anonymous authentication; and, when implementing m-out-of-n authentication, providing availability protections against device misplacement and denial-of-service attacks.

1.2 SQL-Identifier Injection Attacks

Injection attacks such as SQL-Injection Attacks (SQLIAs) continue to be considered the most critical web-application vulnerabilities [1].

The following Java code provides a classic example of a SQLIA vulnerability.

```
String sql = "SELECT address FROM Customer WHERE password = '" + userInput + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

By entering an input such as `'OR true --`, an attacker can make the executed query be

```
SELECT address FROM Customer WHERE password = '' OR true --'
```

where `--` introduces a comment in SQL code and `OR true` bypasses the password check. Under this attack, the executed query returns all addresses in the Customer table.

A particularly problematic subclass of SQLIAs involves the injection of identifiers into SQL statements. We call such attacks SQL-Identifier Injection Attacks, or SQL-IDIAs [5]. As far as we are aware, this dissertation is the first to specifically define and address SQL-IDIAs.

Identifiers may appear in SQL statements as, for example, names of tables, columns, indexes, databases, views, functions, procedures, or triggers. The following Java code demonstrates a SQL-IDIA vulnerability.

```
String sql = "SELECT * FROM Contact ORDER BY " + userInput;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

Here the untrusted user input injects an identifier, referring to a column name in the Contact table, according to which the results will be ordered. By entering an input such as

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic WHERE firstName='John' AND lastName
   ='Doe') > 0 THEN Contact.lastName ELSE Contact.firstName END)
```

an attacker can observe the query results for the Contact table to infer whether John Doe appears in the Demographic table. As described in Chapter 6, we have successfully mounted such attacks against a deployed Electronic Medical Record web application to leak confidential information about patients.

SQL-IDIAs are of special interest because the existing standard SQLIA-preventing mechanisms (i.e., prepared statements) do not protect against SQL-IDIAs.

1.3 Contributions and Roadmap

The first contribution of this dissertation is a novel authentication technique called coauthentication; this dissertation introduces and evaluates coauthentication techniques, including several specific coauthentication system designs, protocols, and implementations. The second contribution of this dissertation is a new class of attacks called SQL-Identifier Injection Attacks (SQL-IDIAs); this dissertation defines SQL-IDIAs, analyzes the usage of SQL statements in source files from GitHub, and introduces an extended prepared-statement API for preventing SQL-IDIAs.

As far as we are aware, coauthentication is the first single-factor, multi-device technique for authenticating users without passwords or biometrics. Additionally, to the best of our knowledge, this is the first work that (1) defines SQL-IDIAs, (2) analyzes the usage of SQL statements in source files from GitHub, or (3) introduces an extended prepared-statement API for preventing SQL-IDIAs.

This dissertation is organized as following.

1. Chapter 2 presents the related work about authentication methods and SQL-Injection attacks.

2. Chapter 3 presents coauthentication system designs, attack models, policies, applications, and coauthentication protocols.

3. Chapter 4 presents the formal verification of the principal security properties of the coauthentication using ProVerif [8, 9]. Additionally, this chapter evaluates the implementability and performance of the coauthentication protocols.

4. Chapter 5 defines SQL-IDIAs. The definition is based on concatenating, into SQL statements, identifiers that have propagated from untrusted inputs.

5. Chapter 6 analyzes 120,412 Java source files from GitHub to demonstrate the prevalence of SQL-IDIAs.

6. Chapter 7 introduces a new extended prepared-statement API to prevent SQL-IDIAs.

7. Chapter 8 concludes and presents the future work.

Much of the text presented in this dissertation are the combination and extension of two published works [4, 5]. Permissions to use the materials of these papers are in Appendix A.

CHAPTER 2

RELATED WORK

Web applications are vulnerable to many different attacks [1, 10], including:

- Injection attacks [11, 12, 13, 14, 15]
- Broken authentication [16, 17, 18, 19, 20, 21]
- Cross-site scripting (XSS) [22, 23, 24, 25]
- Attacks that exploit security misconfiguration [26, 27, 28, 29, 30]
- Denial-of-Service (DoS) attacks [31, 32, 33, 34, 35]
- Attacks that exploit wireless communication [36, 37, 38, 39, 40, 41]

According to OWASP [1], injection attacks and broken authentication are the top two "most critical" security risks [1]. In this chapter [1], we focus on these top two vulnerabilities.

2.1 Authentication Methods

Authentication is one of the most common security activities end-users perform. Authentication is also a common target of attacks, through phishing, guessing, man-in-the-middle, token-theft, and related vectors. Due to the commonality of using and attacking authentication systems, even modest improvements to their security or usability may produce significant benefits.

[1] Parts of this chapter is published in ACM Symposium on Applied Computing [4] and IEEE Conference on Communications and Network Security [5]. Permissions to use these materials are provided in Appendix A.

2.1.1 Authentication Factors

As is well understood, user authentication is based on factors, the three standard factors are

- what you know (human-entered secrets like passwords),
- what you have (physical tokens like keys, electronic remote controls, or smartcards), and
- what you are (biometrics like fingerprints).

Every authentication system, regardless of the factors used, is based on secrets, which could take the form of passwords, patterns of metallic teeth on keys, radio frequencies at which devices transmit data, codes stored on devices and transmitted, fingerprints, etc. Authentication systems aim to protect against attackers who have not obtained the required secrets.

Each authentication factor has advantages and disadvantages [42]. Following subsections present the main disadvantages of the common authentication techniques.

2.1.1.1 What-you-know-factor

Password authentication is the most commonly used what-you-know-factor authentication technique [43]. However, the traditional password authentication is vulnerable to many well-studied attacks. For example, password authentication is vulnerable to guessing [44, 18], phishing [45, 19, 46], DoS [47, 48, 49], and dictionary [19, 50, 45] attacks.

Password authentication also suffers from many usability drawbacks. Research has shown that users rarely update their passwords [51]. Additionally, complex password policies decrease the usability. In particular, users often forget their passwords [52]. Due to the

complexity of passwords, users often write down secret password information on a piece of paper.

Using password authentication on mobile devices creates additional vulnerabilities and usability drawbacks. Research has shown that (1) it takes longer to type passwords on phones, and (2) users make more mistake and get frustrated while typing passwords on mobile devices [52]. Due to these problems, users tend to choose weaker passwords. Additionally, typing passwords on mobile devices may leak the password information via shoulder surfing [53].

2.1.1.2 What-you-are-factor

Common what-you-are-factor authentication techniques are fingerprint scans, voice recognition, facial recognition, and iris or eye recognition. All of these techniques are based on biometric information of humans. Biometric-based authentication methods are vulnerable to secret duplication and replay attacks [42]. Additionally, the common disadvantage of the biometric-based authentication method is the DoS problems due to the false matches and limited attempts.

The main drawback of the biometric-based authentication techniques is that the users cannot update their secret information. For example, if a user's fingerprint is compromised, the user cannot update his/her fingerprint information.

2.1.1.3 What-you-have-factor

Id cards, credit cards, security tokens are common examples of what-you-have-factor authentication techniques. Each of these techniques is vulnerable to theft-based attacks. For example, tokens are susceptible to theft, but doing so in the obvious way requires physical access. Users will often notice physical theft of a token more readily than a remote theft or guessing of a password or biometrics. However, tokens have traditionally relied on special-

purpose hardware and consequently been more expensive to implement and deploy than other factors. In addition, usability benefits of tokens have traditionally been offset by the costs of having to carry and handle the tokens [42, 54].

2.1.1.4 Multi-factor Authentication

Multi-factor authentication attempts to improve security by requiring successful attacks to compromise every factor being used [55]. One two-factor mechanism combines a username and password with a second password (a one-time password, OTP) texted to the user's phone [56]. Alternatively, instead of receiving an OTP from the authenticator, the phone may share a cryptographic key with the authenticator and generate its own OTP, called a time-based OTP or TOTP, as a cryptographic hash, using the shared key, of the current time [57]. A benefit of such mechanisms is that the physical-token factor is a device already possessed and carried by the user, thus avoiding expensive, dedicated hardware.

However, multi-factor techniques add the inconveniences of each factor required. For example, because OTP and TOTP techniques require users to enter two passwords and carry a registered device, they suffer from the nontrivial usability drawbacks of password-based authentication mechanisms (e.g., [58, 59, 51, 60, 52]) and the inconvenience of having to access a mobile device to authenticate.

2.1.2 Threshold Schemes and Multi-Signatures

An (m, n) threshold scheme enables a secret to be divided among n entities, such that each entity has one piece of the secret and m of the n pieces are required to determine the secret [61]. An (m, n) threshold scheme has cryptographic benefits analogous to the user-authentication benefits of an m-out-of-n-device coauthentication policy; both protect against fewer-than-m entities acting maliciously and at-most-n-minus-m entities being unavailable to participate.

Multi-signature schemes similarly enable different users or devices to generate a joint digital signature [62].

Threshold and multi-signature schemes do not provide coauthentication systems, and vice versa, as they differ in techniques and goals. Threshold (multi-signature) schemes contribute techniques for combining secret-pieces (signatures) into a joint secret (signature), while coauthentication systems require no joint secret or signature. Coauthentication secrets (i.e., keys) may be used only independently, to indicate one device's participation in user-level authentications, without ever being combined. The goals of threshold and multi-signature schemes focus on combining pieces of cryptographic secrets or signatures into joint secrets or signatures, while coauthentication's goals focus on user authentication.

2.1.3 OTPs and Other Techniques Using Multiple Devices

One group of techniques related to coauthentication uses OTPs, as discussed. The standard use of OTPs is as follows. A user enters a username and password on a requestor device, the authenticator SMS-texts an OTP to the user's phone (which may also be the requestor device), and the user sees the OTP and enters it on the requestor device as a second password required for authentication. This use of OTPs differs from coauthentication in several ways, perhaps the most significant being that the OTPs are used in two-factor systems, while coauthentication is a single-factor system. Hence, attackers can break the OTP portion of authentications by compromising one device, the victim's phone, or by reading the SMS messages sent to the phone [56, 63, 64].

Another related group of techniques use multiple devices to acquire multiple passwords or biometric data [65]. The authenticator combines these data to determine whether to authenticate a user. For example, if a user has a sensor-device implanted in each finger, then each device may send data related to that finger's motion to an authenticator, which can make authentication decisions based on whether a user has moved or gestured in the proper

way for that user. Although using multiple devices, this line of work relies on users to enter passwords or biometrics, which are assumed to be unguessable and unforgeable by attackers.

Coauthentication, like other zero- or low-interaction authentication systems [66], shields users from attacks based on guessing or forging authentication secrets, such as password phishing or biometric surveillance. Coauthentication users never have to access or even understand the secrets required for authentication, and coauthentication secrets can be generated automatically, with high entropy, and without concern for whether humans have the resources (cognitive ability, time, etc.) to generate, store, update, or enter the secrets.

Bonneau et al. evaluated authentication techniques, including OTPs, according to three axes: usability, deployability, and security [67]. A total of 25 criteria are considered along these axes, such as whether the techniques require users to memorize secrets or carry devices. As motivated in Section 1, we consider disadvantages related to requiring users to carry devices to be decreasing. In any case, we believe that coauthentication satisfies the majority of Bonneau et al.'s criteria, though it is difficult to make precise claims in this respect, due to subjectivity in the criteria [67, Section V-B]. The most significant criteria coauthentication does not satisfy relate to deployability; deploying coauthentication, like deploying any new authentication technique, would require updating authentication clients and servers, and in some implementations, relying on co-location verification.

2.2 SQL-Injection Attacks

SQL-injection-attack vulnerable programs form SQL statements by concatenating untrusted inputs. Every concatenation of untrusted input into a SQL statement can produce a SQLIA [15, 14]. Many mechanisms exist for mitigating SQL-Injection attacks. In this section, we present these existing SQLIA mitigation techniques.

2.2.1 Existing Methods

Due to the popularity of SQLIAs, several dynamic and static analysis methods have been proposed. Dynamic methods (e.g., [15, 14]) and tools (e.g., [68, 69, 70]) aim to mitigate injection attacks at runtime. However, none of them are widely adopted at present due to high performance overheads.

Static analysis tools (e.g., [71, 72, 73]) are also not widely adopted due to high false positives [74]. These false positives result, for example, from imprecision in the information-flow analyses used to determine how untrusted inputs get concatenated into SQL-statement outputs.

Input sanitation is a more common technique for mitigating SQLIAs. Input sanitation may entail whitelisting valid inputs, blacklisting invalid inputs, or escaping special characters [75]. All of these techniques have well-documented drawbacks, including:

- Whitelists and blacklists may introduce nontrivial complexities into application code. For example, creating a new database table may require dynamically changing a whitelist or blacklist of valid table names usable within SQL statements. In addition, incorrect or delayed dynamic updates to whitelists or blacklists may introduce false negatives or positives [76, 77].

- When escaping special characters, some unexpectedly encoded characters may not be properly recognized and escaped, creating false negatives [78]. Examples include SQL smuggling [79], character homoglyph injection [80], and string literal injection without quotes [81].

- Escaping special characters may also introduce SQLIA vulnerabilities. For example, escaping single quotes in input strings (e.g., converting `\';[code]--` to `\'';[code]--`) may cause an application to output the SQLIA-exhibiting statement `DELETE FROM table WHERE name='\';[code]--'` [78].

```
String sql = "SELECT address FROM Customer WHERE password = ?";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setString(1, userInput);
ResultSet rs = stmt.executeQuery();
```

Figure 2.1: A Java program using prepared statements.

- Applications may be vulnerable to second-order injection attacks when a sanitized input is stored in a database and the stored input is reused without sanitation [81].

Due to these drawbacks, prepared statements are the standard defense against SQLIAs [77].

There have been efforts to build automatic prepared-statement-generation tools [82, 83, 84]. These tools analyze source code and convert concatenated SQL-statements to prepared statements. Although some of the SQLIAs are prevented, none of these tools can prevent SQL-IDIAs due to the fact that prepared statements cannot fill placeholders with identifiers. Utilizing the proposed extended prepared-statement API can enable these tools to prevent SQL-IDIAs.

2.2.2 Prepared Statements

Prepared statements, also known as parameterized queries, are the de facto mechanism to prevent SQLIAs [75]. Figure 2.1 presents a program that employs prepared statements to perform database operations. At a high-level, preventing SQLIAs with prepared statements involves three main steps. First, an application creates a SQL-statement that has placeholders (i.e., the question mark symbol in the SQL statement) for literals and sends this statement to a database system. Then, when the prepare-statement function is executed, the database parses the statement and creates a statement structure having placeholders. Next, the application fills these placeholders (e.g., by calling the `setString` function) with values. This mechanism enforces that applications fill placeholders with literals, thus preventing SQLIAs.

A major limitation of prepared statements is that only application-level values are allowed to replace placeholders [77]. For example, for applications written in Java, placeholders in prepared statements may only be replaced by Java values such as string literals, integer literals, and Java objects.

This limitation to replace prepared-statement placeholders with only application-level values prevents safe construction of SQL statements having dynamically resolved identifiers [85, 86]. That is, standard libraries do not allow SQL identifiers such as table and column names to replace placeholders in prepared statements. SQL syntax allows identifiers to appear in many statements and clauses, including `create`, `alter`, and `drop` statements and `order by` and `group by` clauses.

Because standard libraries do not allow SQL identifiers to replace placeholders in prepared statements, developers must concatenate dynamically resolved identifiers into SQL statements, though such concatenations create SQL-IDIA vulnerabilities, like the one shown in Figure 5.1. In fact, prior work showed that every concatenation of untrusted input into a SQL statement can produce a SQLIA [15, 14].

CHAPTER 3

COAUTHENTICATION SYSTEMS

This chapter [1] introduces the coauthentication systems. The devices involved in coauthentication are the *authenticator* (e.g., a server deciding whether to authenticate a user), the *requestor* (on which the current authentication attempt is initiated), and one or more *collaborators*. The requestor and collaborator(s) are *registered* with the authenticator, meaning that the devices have access to a secret that the authenticator can use to verify the devices' participation in an authentication. This secret accessible to the requestor and collaborator(s) may, for example, be a secret key shared with the authenticator, or a private key K such that the authenticator can verify signatures created with K.

In some coauthentication protocols, the authenticator, upon receiving an authentication *request*, issues one or more *challenges* and awaits one or more valid *responses* to the challenges. Other protocols avoid authentication challenges. In all cases, the authenticator verifies that multiple registered devices, more specifically the secret keys accessible to those devices, participate in the authentication.

3.1 Attack Models and Assumptions

Coauthentication, like multi-factor techniques, protects against theft of any one authentication secret. The secrets in coauthentication are cryptographic keys. Theft of coauthenti-

[1] Parts of this chapter is published in ACM Symposium on Applied Computing [4]. Permission to use the material is provided in Appendix A.

cation secrets may occur in any way, including by remotely compromising devices to obtain their stored keys or physically stealing devices.

Attackers are assumed to be active and can eavesdrop on, insert, delete, and modify communications. Attackers may mount replay and man-in-the-middle attacks.

Attackers are however assumed to be incapable of cryptanalysis; attackers can only infer plaintexts from ciphertexts when also having the required secret key. Without such an assumption, attackers could extract credentials like session keys simply by monitoring and cryptanalyzing legitimate authentications.

Some coauthentication protocols protect against attackers who know all the secrets stored on a device that the victim user possesses. We call such attacks *key-duplication attacks*. For example, an attacker may duplicate a device's secret keys by remotely compromising the device. Alternatively, the attacker may physically steal a device, duplicate all keys accessible to the device, and return the device to the victim user, who may be unaware of the theft and duplication.

To protect against key-duplication attacks, the coauthentication protocols assume that a private communication channel, inaccessible to attackers, exists between the requestor and collaborator devices. Such an assumption is necessary because the duplicated keys must be updated through some channel inaccessible to the attacker; otherwise, the attacker—who has all of the victim device D's keys—could decrypt and obtain any updated keys sent to D, and modify any updated keys sent from D. Private channels may be implemented with short-range communications, such as NFC, zigbee, wireless USB, infrared, or near-field magnetic induction, under the assumption that attackers cannot access such communications because they are on direct, device-to-device channels.

Other coauthentication protocols do not require a private channel between requestor and collaborator devices. Although these protocols do not protect against key-duplication attacks, they do protect against attackers who obtain keys by stealing devices (without

duplicating the keys in, and returning, the devices). In other words, the attack model for these all-public-channel protocols assumes that if an attacker has obtained a device D's authentication secret, then D's legitimate user no longer possesses D.

All of this dissertation's coauthentication protocols assume that devices in the user's possession run as intended during the coauthentication process. Without such an assumption, malware on the user's requestor device could simply leak decrypted session keys or any other unencrypted private data, and malware on the user's collaborator device could simply approve an attacker's authentication requests. Protecting against malware that is actively running on a device in the user's possession, while the user is authenticating, is beyond the scope of coauthentication.

All of this paper's coauthentication protocols also assume that authenticators run as intended during the coauthentication process. Without such an assumption, malware on the authenticator could simply leak secrets or allow all authentication requests. Protecting against malware on authenticators is beyond the scope of coauthentication.

3.2 Collaboration Policies

Each collaborator may enforce its own policy defining the circumstances under which it participates in a coauthentication.

For example, a collaborator may only participate in an authentication after a user has clicked a button or provided some other input to confirm participation. Under this policy, if an attacker steals or compromises the requestor and initiates a coauthentication, the legitimate user will not confirm the attacker-initiated authentication on the collaborator, so the authentication attempt will fail.

Alternatively, a collaborator may automatically participate in an authentication but warn the user, or log, that it has done so, for example by displaying a text alert with an audible warning sound (e.g., a text message). The alert could provide a simple interface for the user

to notify the authenticator if the collaboration was unauthorized (i.e., an attacker-initiated authentication).

The first of these example policies, which we call the *disallow-by-default* collaboration policy, only collaborates when a user confirms the authentication. The second policy, which we call the *allow-by-default-with-warning* collaboration policy, relies on users to observe a warning and handle unauthorized collaborations after the fact.

These two policies illustrate a security-usability tradeoff: the disallow-by-default policy prevents attackers from initiating authentications (increased security) but requires user confirmation on each authentication (decreased usability). The allow-by-default-with-warning policy allows attackers to successfully be logged in during the window of time between when the collaborator warns the user and when the user observes the warning and notifies the authenticator of the unauthorized authentication (decreased security), but users don't have to confirm the authentications on the collaborator (increased usability).

For many applications the usability benefits of the allow-by-default-with-warning policy may outweigh the security costs; many modern authentication systems email or text users after suspicious logins and request after-the-fact notification of unauthorized access. For example, users of financial servers may prioritize usability because (1) authorized logins are far more common than unauthorized logins, and (2) the financial institutions, due to legal requirements or to entice customers, may guarantee to reimburse customers for any losses incurred from unauthorized logins.

Additional collaboration policies are possible. For example, a collaborator could decide whether to participate in a coauthentication based on the requestor's proximity, that is, whether the requesting device is co-located with the collaborator. In applications where the attack vector of concern is device theft, a collaborator may presume that a co-located requestor has not been stolen. Such a collaborator may tacitly allow collaborations with co-located requestors but show warnings for, require explicit confirmations for, or disallow

Table 3.1: Example collaboration policies.

Description of the collaboration policy
Disallow by default (require user confirmation before collaborating)
Allow by default, with a warning or log of the collaboration
If co-located then tacitly allow, else allow by default with a warning
If co-located then tacitly allow, else disallow by default
If co-located then tacitly allow, else disallow entirely
If co-located then allow by default with a warning, else disallow by default
If co-located then allow by default with a warning, else disallow entirely
If co-located then disallow by default, else disallow entirely

entirely, collaborations with non-co-located requestors. Many methods exist for detecting or enforcing co-location; a simple implementation might just require communication through a short-range channel such as NFC, Bluetooth, or body area network [87, 88].

Table 3.1 lists several of these example collaboration policies. Many others are possible, such as taking into account the source of authentication requests or how much time has passed since earlier collaborations.

3.3 The Full Coauthentication Protocol

Figure 3.1 illustrates the full coauthentication protocol for two user devices. Authentication secrets in this protocol are shared symmetric-cryptography keys, and there is only one collaborator. Secret key K_{AR} (K_{AC}) is shared between authenticator and requestor (collaborator). Each N_i is a nonce, and $\{M\}_K$ is the encryption of M using key K. The third message is sent through a private channel.

Following the flow of data in Figure 3.1, the full protocol operates as follows. Assume that during device registration, the authenticator A and requestor R share a secret key K_{AR}, and the authenticator A and collaborator C share a secret key K_{AC}.

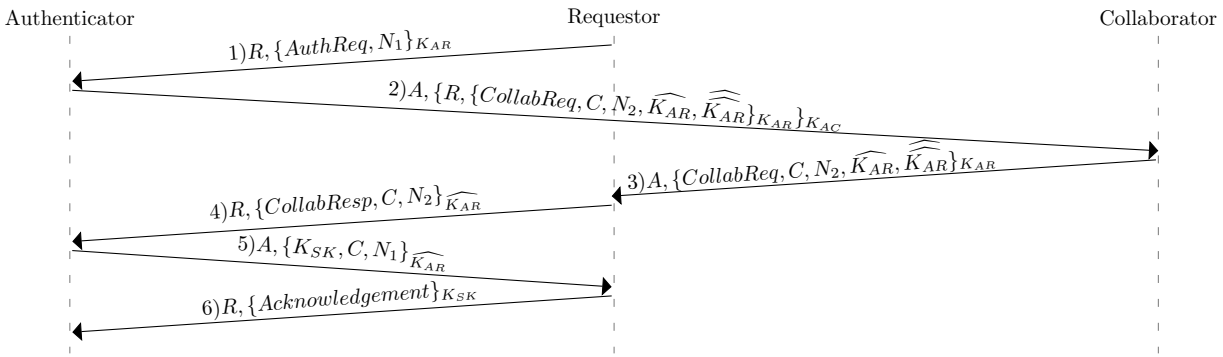


Figure 3.1: The full coauthentication protocol.

1. Requestor R initiates the coauthentication by sending the authenticator A its ID and an encrypted authentication-request message containing a challenge nonce N_1 (which serves to authenticate A to R).

2. Authenticator A receives and decrypts the request message, finds that the requestor R is registered to a user having collaborating device C, creates a challenge nonce N_2 (which serves to authenticate R to A), generates *two new keys* ($\widehat{K_{AR}}$ and $\widehat{\widehat{K_{AR}}}$) to share with R (to rotate keys, to ensure forward secrecy and prevent key-duplication attacks), and double encrypts these data in a collaboration-request message to C, the first (inner) encryption using K_{AR} and the second (outer) encryption using K_{AC}. By double encrypting nonce N_2, the authenticator ensures participation of both user devices' secret keys (K_{AR} and K_{AC}) in the coauthentication.

3. Collaborator C receives and decrypts the previous message, verifies the identity of the requestor, and forwards the decrypted message (which is still ciphertext encrypted with K_{AR}) to requestor R through a private channel.

4. Requestor R receives and decrypts this message using K_{AR}, verifies the identity of the collaborator, and obtains N_2, $\widehat{K_{AR}}$, and $\widehat{\widehat{K_{AR}}}$. The requestor then generates and sends the authenticator a collaboration-response message containing N_2 encrypted with its

first updated key, $\widehat{K_{AR}}$. The requestor saves the second updated key, $\widehat{\widehat{K_{AR}}}$, for a future coauthentication request.

5. Authenticator A receives the collaboration-response message, decrypts, and verifies the collaborator's identity and that the received nonce matches the N_2 it sent earlier. Because A has now verified participation of both keys K_{AR} and K_{AC}, it sends an authentication-complete message, for example containing a session key K_{SK}, to the requestor R.

6. Requestor R sends an acknowledgment to the authenticator.

Timestamps may be added to these messages, for example to implement timeouts or fine-grained logging. Message Authentication Codes (MACs) may also be added to these messages to provide data integrity.

Notice that full coauthentication stores three keys long term: K_{AR} may be stored long term before the current round of authentication, $\widehat{\widehat{K_{AR}}}$ may be stored long term after the current round of authentication, and K_{AC} may be stored long term before and after the current round of authentication.

3.3.1 Properties of the Full Protocol

The full coauthentication protocol uses nonces to authenticate the requestor and authenticator to each other—session keys are only shared between mutually authenticated devices. Requestor R only shares session keys with authenticated As, and authenticator A only shares session keys with authenticated Rs.

The full protocol also employs key rotation to ensure forward secrecy. An attacker who acquires the keys stored long term on at most one user device cannot obtain past session keys. Each session key K_{SK} is encrypted with an updated $\widehat{K_{AR}}$.

The full protocol mitigates man-in-the-middle attacks by making the authentication secrets shared between the authenticator and user devices be cryptographic keys, used to encrypt communications. In contrast, man-in-the-middle attacks may be possible on password or biometrics systems because the authenticator may only share, with users or user devices, secrets that are insufficient for cryptographic use. For example, a man-in-the-middle attack on an OTP system may proceed as follows: the victim enters a username and password on a fake website; the fake website forwards this information to the real website, which then issues an OTP; the victim receives and enters the OTP into the fake website; the attacker completes the authentication on the real website and masquerades as the user. In this case the shared username/password (or hash thereof) is insufficient for providing the cryptographic properties needed to mitigate man-in-the-middle attacks.

Now suppose an attacker acquires the long-term secrets stored on at most one user device. Acquiring K_{AC} only enables an attacker, even one with access to the private channel, to permit or deny authentications initiated by the victim. Attackers are already assumed to be active and consequently capable of denying service by dropping network messages. Acquiring K_{AC} therefore provides an attacker with no new capabilities (and Section 3.4 describes extensions of coauthentication that mitigate denial-of-service attacks on user devices).

On the other hand, acquiring only the K_{AR} to be used in the next coauthentication request enables an attacker to request authentication, but assuming an appropriate collaboration policy, the collaborator will notify the victim user of the authentication attempt. From the victim's perspective, this attacker-initiated authentication attempt will be unexpected, so the victim will deny collaboration and therefore the authentication.

Acquiring only the K_{AR} to be used in the next coauthentication request also enables an attacker mounting a key-duplication attack to wait for and decrypt a legitimate authentication request coming from the requestor device, still in the victim's possession. However, such an attacker only obtains nonce N_1 in the process and cannot decrypt any of the remaining

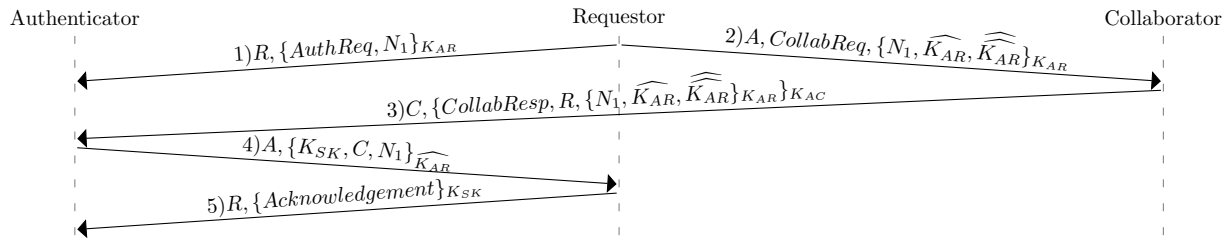


Figure 3.2: A coauthentication protocol omitting authenticator challenges.

messages in the protocol, because they are either encrypted with different keys or sent on a private channel. Obtaining K_{AR} and N_1 provides an attacker with no new capabilities.

The full coauthentication protocol therefore protects against attackers who have acquired the long-term secrets stored on at most one user device. ProVerif has been used to formalize and verify these arguments, as described in Chapter 4.

3.3.2 Variation: Omitting the Challenge-Response

It is possible to avoid the challenge-response portion of the full coauthentication protocol, implemented with nonce N_2, by having the requestor send two requests, one to the authenticator (to request authentication) and another to the collaborator (to request collaboration).

Figure 3.2 shows such a challengeless protocol. The second message is sent through a private channel. The requestor sends two requests, one to the authenticator and another to the collaborator, containing the same nonce N_1. The requestor also includes the updated versions of K_{AR} in its collaboration-request message, which the collaborator forwards to the authenticator. These updated keys are double encrypted during transit from the collaborator to the authenticator, protecting the keys against attackers having obtained at most one of K_{AR} and K_{AC}. After verifying that both the requestor and its registered collaborator have participated in an authentication by sending the same N_1, the authenticator sends a new session key to the requestor, encrypted with the proper updated version of K_{AR}. As in the

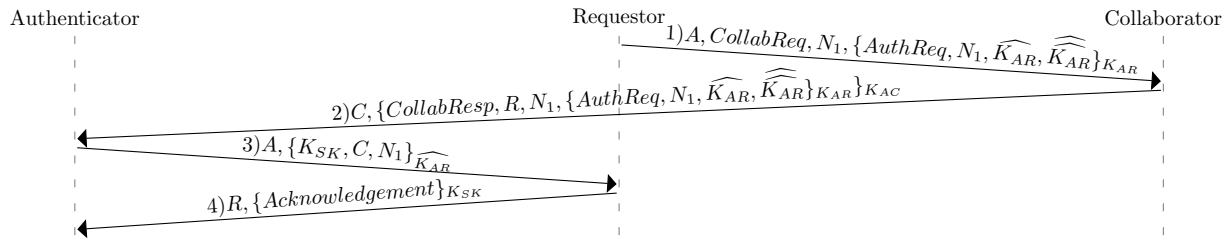


Figure 3.3: A challengeless coauthentication incorporating message forwarding.

full protocol, this challengeless version results in the authenticator and requestor sharing an updated $\widehat{\widehat{K_{AR}}}$, usable in a subsequent run of the protocol as the new version of K_{AR}.

Having formally verified both the full and challengeless coauthentication protocols, to our knowledge they provide the same security guarantees. The known tradeoffs between these protocols relate to performance. The challengeless protocol is expected to be more efficient overall, due to the omission of challenge creation and the parallelization or batching of some of the communications (e.g., the first and second messages in Figure 3.2). However, the computations performed by individual devices may be more efficient in the full version. For example, from the requestor's perspective, the challengeless protocol essentially replaces the computations needed to decrypt the third message and generate the fourth message of Figure 3.1 with the computations needed to generate the second message of Figure 3.2, including generating updated versions of K_{AR}. For some user devices, such as IoT devices with limited resources, some of these computations may be more expensive than others, making one protocol more efficient than another for those devices.

3.3.3 Variation: Incorporating Message Forwarding

Figure 3.3 shows a variation of the challengeless protocol that incorporates message forwarding. The first message is sent through a private channel. The protocol shown in Figure 3.3 is the same as the one shown in Figure 3.2 but with the collaborator forwarding the authentication-request message to the authenticator on behalf of the requestor. The protocol

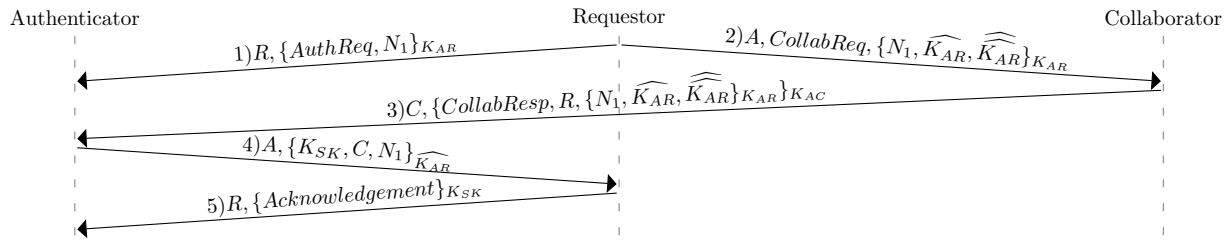


Figure 3.4: An all-public-channel variation of the full coauthentication (Figure 3.1).

of Figure 3.3 is of particular interest because it achieves the minimal number of messages needed to coauthenticate. Ignoring the final acknowledgment, successful coauthentication requires at least three messages because both the requestor and collaborator must demonstrate participation to the authenticator, and the authenticator must respond with a new session key or other post-authentication capability.

3.3.4 Variation: No Private Channels

In cases where a private channel does not exist between the requestor and collaborator, coauthentication protocols cannot prevent key-duplication attacks. The ability of an attacker, who has acquired all the secrets stored on a user-possessed requestor R, to eavesdrop on and modify all communications to and from R, makes it impossible to update R's secrets without the attacker also obtaining any updates sent to R and modifying any updates sent from R.

In practice it may be acceptable to dismiss key-duplication attacks by relying on alternative mechanisms to mitigate them. For example, a device's long-term, rarely updated key K_{AR} may be stored in a trusted platform module (TPM) [89]. With K_{AR} in a TPM, we might assume that attackers, who possibly have physical access to the requestor R, may be able to *use* K_{AR} to initiate authentications on R, but cannot *extract* K_{AR} from R. That is, mechanisms like TPMs may mitigate key-duplication attacks by allowing authentication secrets to be used but not extracted, and therefore not duplicated.

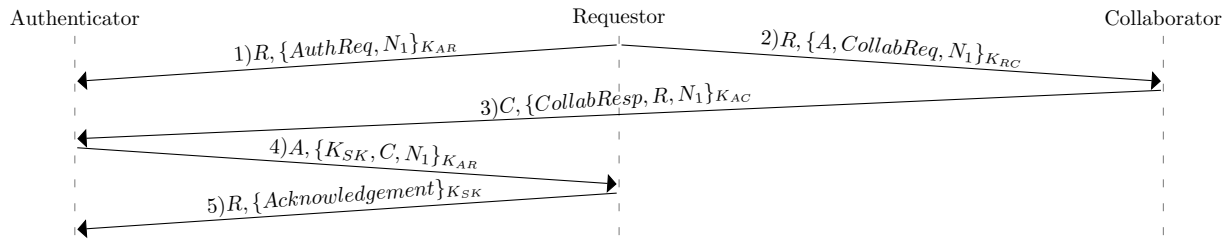


Figure 3.5: All-public variation of the challengeless coauthentication (Figure 3.2).

It may also be acceptable to dismiss key-duplication attacks in cases where the threat is considered remote or private channels simply cannot be implemented or would be costly to implement.

In any of these cases, the coauthentication protocols can be varied to no longer require a private channel between the requestor and collaborator, yet still protect against non-key-duplication attacks. The attack model for these all-public-channel protocols assumes that if an attacker has obtained a device D's authentication secret, then D's legitimate user no longer possesses D. This attack model still covers attacks based on stealing devices and attempting to authenticate on the stolen devices.

All-public-channel variations exist for all the protocols shown in Figures 3.1–3.3.

Figures 3.4–3.6 show all-public-channel variations of Figures 3.1–3.3. These all-public-channel protocols match the private-channel protocols, except they abandon all messages and data whose purpose was to update keys (such as the third message in Figure 3.1) and encrypt all messages sent between the requestor and collaborator with a shared key K_{RC}.

The all-public-channel protocols are simpler, and expected to run more efficiently, than the private-channel protocols but do not protect against key-duplication attacks and do not satisfy forward secrecy.

In practice a hybrid approach may be preferred: coauthentication keys may be updated only periodically, using private channels at opportune times, while public-channel protocols are used in the common case.

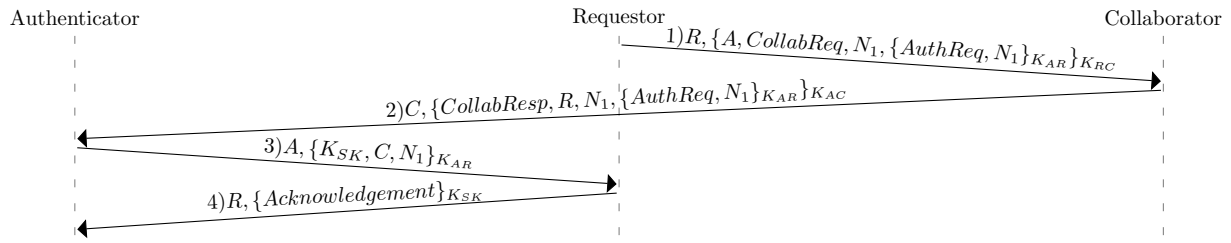


Figure 3.6: An all-public-channel message forwarding coauthentication.

To make an analogy with password-based authentication systems, ideally—from a security perspective—users would update their passwords on every authentication, to limit attackers who have acquired passwords. Doing so would be like using the private-channel protocols for coauthentication. In practice, however, tradeoffs are made, and passwords are typically updated only rarely [59].

3.3.5 Variation: Asymmetric Cryptography

Asymmetric (public-key) operations may replace the symmetric-cryptographic operations in coauthentication protocols.

For example, Figure 3.7 shows a public-key version of the all-public-channel protocol shown in Figure 3.4. The encryption of M using the requestor's public key is notated $\{M\}_R$, and $\{M\}_{R^{-1}}$ refers to R's digital signature of M (and similarly for authenticator A and collaborator C). Converting from *all-public*-channel protocols based on symmetric cryptography (such as are shown in Figures 3.4–3.6) to ones based on asymmetric cryptography requires only standard techniques (encryptions in the symmetric version based on shared keys are replaced by encryptions in the asymmetric version based on the recipient's public key, digital signatures are added to messages, etc).

Converting from *private*-channel protocols based on symmetric cryptography (such as are shown in Figures 3.1–3.3), to ones based on asymmetric cryptography requires additional techniques. In these cases, where defense against key-duplication attacks and forward secrecy

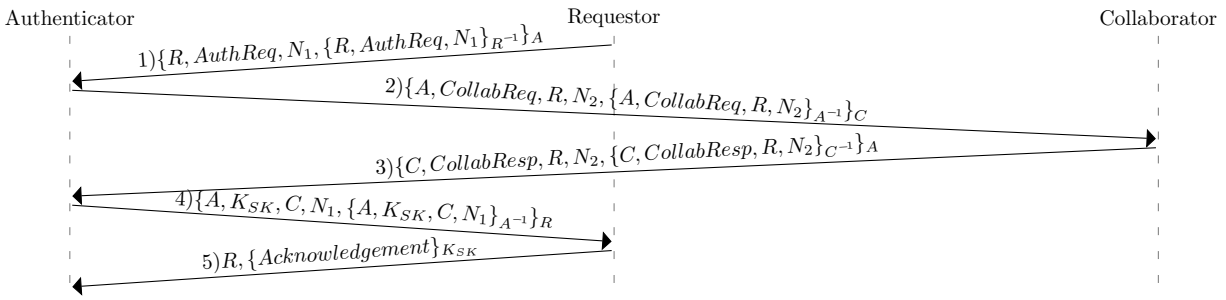


Figure 3.7: A public-key variation of the protocol in Figure 3.4.

need to be provided, the requestor's (public, private) key pair must be updated on every authentication.

Updates to the requestor's (public, private) keys may occur in various ways. A basic design would have the authenticator provide two new key pairs to the requestor in the same ways that the symmetric designs (e.g., Figure 3.1) have the authenticator provide two new shared keys to the requestor. However, having the authenticator generate private keys for the requestor, even private keys only used for the authenticator's services, may not be considered appropriate for asymmetric systems. Less practically, the requestor could update its keys in the public-key infrastructure before or during every authentication.

3.3.6 Variation: Three Device Coauthentication

A variety of coauthentication protocols exist for cases in which a user has registered more than two devices with an authenticator. When coauthenticating, such a user may have multiple possible collaborator devices. The authenticator in such a case may determine some subset of the registered user devices to which to send challenges, possibly based on guidance from the requestor. The authenticator may then send one or more challenges to this subset of devices, such that the challenges cryptographically require participation from some or all of the user devices.

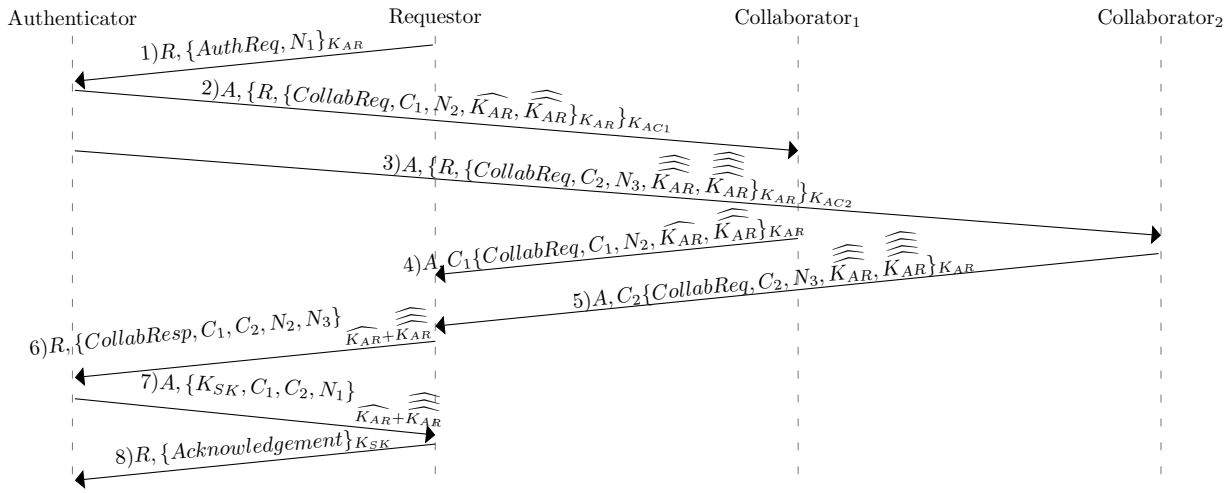


Figure 3.8: A three device variation of the protocol in Figure 3.1.

Figure 3.8 shows such a coauthentication protocol that uses two collaborators. The authenticator sends two collaboration requests, one to the first registered collaborator C_1 and another to the second registered collaborator C_2. Notice that the authenticator includes updated versions of K_{AR} in its collaboration-request messages. Once the requestor obtains two collaboration messages from the collaborators, it sends a collaboration-response message that is encrypted with the combination of updated keys obtained from the two collaboration messages. This three-device version of coauthentication results in the authenticator and requestor sharing an updated $\widehat{K_{AR}}$ and $\widehat{\widehat{K_{AR}}}$ combination, usable in a subsequent run of the protocol as the new version of K_{AR}.

This three-device variation of coauthentication protects against the compromise of two user devices. Compromising two of the user's devices is not sufficient for authenticating as that user, because an attacker still needs an extra device for coauthentication.

Formal verification of protocols for 3 and 4 device coauthentication is presented in Section 4.1, and further generalizations to multi-device coauthentication are discussed in Section 3.4.5.

3.4 Applications of Coauthentication

Besides the more-obvious applications of authentication, such as logging in to operating systems and web services, authentication occurs in less-obvious, but common, ways, including pushing a button on a remote control to open a door or gate, using a physical key to unlock a door or start an engine, or swiping, scanning, or inserting a payment card, passport, or driver's license.

A thief who successfully breaks into a car containing a garage-door or gate controller can push a button to open the victim's garage or gate. Such attacks have occurred [90, 91]. With coauthentication, a garage-door or gate controller may require both the requestor (a car or a remote control in the car) and a collaborator (a smartphone) to participate in the authentication required for opening. Then an attacker stealing only the victim's car, or only the victim's smartphone, cannot open the door or gate. This benefit is nontrivial due to the *heterogeneity* of the required user devices (car and phone): people rarely leave phones in cars unattended, so a successful attack on, or theft of, one of the two user devices does not normally provide all the secrets required for coauthentication.

Door locks are a similar application. Here the requestor may be a radio transmitter, for example on smart apparel, that sends requests to all locks (authenticators) located within 1m, and the collaborator may be a smartphone. The collaboration policy might require that the phone tacitly allows collaborations for co-located requestors and disallows, with warnings, all other collaborations. Such a system mitigates the recent relay attacks on car doors [92].

Payment cards might also be coauthenticated, to require participation of a registered smartphone as a collaborator to the requesting payment card (which itself might be combined into an existing device, such as a smartwatch). This coauthenticated payment system would achieve security protections similar to systems in which an OTP, sent to the user's

phone, is required for payment-card use; however, the coauthentication system could run automatically, without requiring the user to enter any passwords, for example with the smartphone enforcing the second-to-last collaboration policy listed in Table 3.1.

When run automatically, without requiring user interaction, coauthentication is a zero-interaction authentication system [66]. By making authentications transparent and unobtrusive, zero-interaction systems enable more devices to benefit from authentication, without fatiguing users with authentication activities. These benefits include enforcing access controls and adjusting to the preferences of each registered user. For example, smart home assistants, smart appliances, and computer components (microphones, keyboards, cameras, memory modules, ALUs) may coauthenticate users to mitigate unauthorized use; televisions may coauthenticate users to enforce parental controls; and chairs, lights, HVAC systems, etc., may coauthenticate users to adjust to their personal preferences.

For this same reason of coauthentication being able to run transparently, or with limited user interaction such as clicking a confirmation button, coauthentication is well suited to continuous authentication [66, 93]. Because continuous authentication requires reauthenticating users periodically, only zero- and low-interaction techniques are appropriate for this application.

Coauthentication may also provide accessibility benefits over existing authentication methods. For example, typing a password can be challenging for visually impaired users on mobile devices [94]. Using the existing accessibility features of mobile devices such as voice recognition and large text fonts may leak sensitive password information to attackers via shoulder surfing [43, 53] or auditory surveillance [95]. Visually impaired users may benefit form coauthentication; users may click a large button to coauthenticate, and then the coauthentication system could run automatically, without requiring the users to enter any passwords.

Extensions and generalizations of coauthentication are possible. Unlike the variations presented in Sections 3.3.2–3.3.4, this extensions presented in the following subsections are not tied to the details of the full protocol.

3.4.1 Additional Challenges and Responses

The full coauthentication protocol uses a challenge-response process in which the authenticator first encrypts a nonce to challenge the requestor R and then encrypts the result to challenge the collaborator C. Receiving a valid response requires participation from, and collaboration between, both R and C.

Many other challenge-response processes are possible. For example, the authenticator may reverse the order of encryptions to require the requestor to participate before the collaborator; the authenticator may only encrypt the challenge with one shared key (e.g., K_{AR}) and wait for a valid response encrypted with the other shared key (e.g., K_{AC}); the authenticator may concurrently send the requestor and collaborator the same or different challenges and require valid responses from both; or the authenticator may issue challenges for which generating valid responses requires interacting with one or more third-party servers. Authenticators may issue challenges requiring other sorts of responses as well, such as requiring message authentication codes (MACs) or passwords or other secrets in responses.

3.4.2 Locally Broadcasting Challenges

In some applications it may be useful to make some of the coauthentication protocol's communications multicast or broadcast. For example, to require additional user interaction during coauthentication, the requesting device may receive a challenge directly from the authenticator and then display (visually/locally broadcast) the challenge as a QR code [96]. The user could then scan the challenge QR code on the collaborating device, which could then send a valid response to the authenticator.

3.4.3 Collaboration to Obtain Session Keys

Similar to the double encryption for transmitting challenges in the full coauthentication protocol, authenticators may double encrypt authentication-complete (e.g., session-key) messages. Encrypting session-key messages with both K_{AC} and K_{AR} (or $\widehat{K_{AR}}$) forces both the requestor and collaborator to participate, to obtain the session key. Requiring this additional collaboration mitigates attacks in which the requestor is noticeably compromised after the collaborator responds to the authentication challenge, giving the collaborator one more chance to confirm the authentication before the requestor obtains the session key.

3.4.4 Authenticating Other Devices

The coauthentication protocols may also be modified to give the collaborator access to the session key. For example, coauthentication may proceed as normal before the requestor shares the session key with the collaborator, possibly after a mutual authentication between requestor and collaborator. Another example involves a device, pre-authenticated with the requestor, using the requestor as a proxy to authenticate to the authenticator, building a chain of authentication. Such designs enable n-way authentication, or mutual authentication between multiple pairs of devices.

3.4.5 Multiple Collaborators, m-out-of-n Policies, and Availability Benefits

There are advantages to systems in which users register more than two devices with an authenticator. Suppose a user has registered n devices and the authenticator requires any m of the n devices to coauthenticate, where $2 \leq m \leq n$. In the coauthentication protocols described so far, m=n=2, but now suppose m=2 and n=3. In this case, compromising only one of the user's devices (i.e., obtaining only one device's authentication secrets) is still insufficient for authenticating as that user, because m=2. At the same time, because m<n,

the user can be authenticated even after forgetting or losing a device, or having a device become inoperable, for example due to a denial-of-service attack.

This m-out-of-n-device policy, enforced at the authenticator, tolerates the absence of $n-m$ devices. Hence, user-side denial-of-service attacks require denying service to $n-m+1$ devices. When these devices communicate through heterogeneous channels, denial-of-service attacks based on jamming or otherwise interfering with specific communication channels become more difficult to mount.

To prevent attackers from using $n-m$ compromised devices to coauthenticate, m may be further constrained to be greater than $n-m$, that is, $m > n/2$. For example, a system that requires only 2 out of 4 devices to coauthenticate (i.e., $m{=}2{=}n/2$) tolerates the absence of 2 devices, but if those 2 devices are absent due to theft, then the thief can use them to coauthenticate. To prevent such attacks, the m-out-of-n-device policy may be constrained to $2 \leq m \leq n < 2m$

The m-out-of-n-device policy can be generalized further, to policies in which devices are, for example, (1) weighted in various ways to get above a threshold (e.g., 2 "votes" are required to authenticate the current user, but each smart shoe only gets half a vote), (2) required (e.g., 2 devices are required but one must be the user's smartphone), or (3) excluded (e.g., high-risk users may not use easily-transferrable smartcards for coauthentication).

3.4.6 Group Coauthentication

Users may also be coauthenticated simultaneously, as a group. Such authentication subsumes the famous two-person concept for authenticating users who will have access to nuclear and other weapons [97, 98], or to bank vaults. For example, a two-person policy may require two users to simultaneously turn four keys, one in each hand, to gain access to a weapon-deployment system. The goal is to require both users to participate in the authentication.

Because coauthentication requires participation of multiple devices in an authentication, it may require participation of multiple users in an authentication, where each user has at least one registered device. The same coauthentication protocols can be followed to authenticate multiple users' devices simultaneously. More sophisticated group coauthentications could, for example, require participation of m-out-of-n devices from each of j-out-of-k users.

Group coauthentication may also be used to implement parental control applications. For example, a learners-permit holding child can only run a car when his smartphone and his parent's registered smartwatch participate in the coauthentication. On the other hand, the parent can run the car by using his own devices. As another example, a smart TV can only display certain TV channels when the parent's smartwatch is participating in the coauthentication.

3.4.7 Device Sharing and Anonymous Coauthentication

Users may also share devices. For example, a garage-door authenticator may receive a request from a shared family car and send challenges to all the smartphones of drivers in the family, or only those smartphones in near-proximity. The smartphones might enforce the collaboration policy of tacitly participating if co-located with the requestor and not participating otherwise.

Alternatively, assume that every collaborator (smartphone) shares *the same* secret key with the garage-door authenticator. Then the authenticator may, upon receiving a request from the family car, respond directly to the car with a challenge requiring participation from any collaborator—and leave it to the car to obtain a collaborator's participation. An interesting aspect of this alternative is the anonymity it provides: the authenticator only communicates with the shared requestor device and does not know which user has been authenticated, nor which device has collaborated. Authentications are still protected against attackers acquiring one of the secret keys.

It is also possible to achieve anonymous coauthentication for systems in which requestor devices are not shared, by having all potential requestors share the same secret key with the authenticator. Because coauthenticators verify usage of keys, anonymity is achieved by having devices share keys.

Of course, these designs only protect anonymity during the authentication process. Authenticators frequently have other opportunities to de-anonymize users, though techniques like onion routing [99] may mitigate some de-anonymizations.

3.4.8 Post-Quantum Coauthentication

Many existing authentication methods rely on public-key cryptography to transfer messages over the public network. However, public-key methods such as RSA and ECC can be compromised using Shor's algorithm when a large-scale quantum powered computer becomes available [100]. Furthermore, a powerful enough quantum computer can crack hashed passwords effectively [101]. Therefore, password authentication and other authentication methods relying on public key cryptography may became ineffective against quantum computers.

As far as we are aware, existing symmetric key algorithms (e.g., AES-256) will remain resistant against a powerful quantum computer [102]. Therefore, symmetric-key coauthentication protocols illustrated in Figures 3.1– 3.6 can be resistant to a powerful post-quantum adversary.

CHAPTER 4
FORMAL AND EMPIRICAL EVALUATION OF COAUTHENTICATION PROTOCOLS

This chapter [1] presents an empirical and formal analysis of coauthentication protocols.

4.1 Formal Evaluation

The principal security properties of the example coauthentication protocols shown in Figures 3.1–3.7, and several multi-device coauthentication protocols such as the one shown in Figure 3.8, have been formally verified with ProVerif [8, 9]. ProVerif uses a resolution-based strategy to verify that protocols satisfy desired security properties. A benefit of using ProVerif is that it can model arbitrarily many sessions of a protocol running concurrently.

4.1.1 Protocol Modeling

The protocol encodings faithfully follow the communications shown in Figures 3.1–3.8.

To model key updates in the private-channel protocols (Figures 3.1–3.3) we used key tables [103, p.37]. These key tables are only accessible to the legitimate actors (i.e., Requestor, Authenticator, and Collaborator) of the protocol. The protocols dynamically generate new keys (i.e., $\widehat{K_{AR}}$ and $\widehat{\widehat{K_{AR}}}$) during an authentication session, and at the end of the session, the new long-term key ($\widehat{\widehat{K_{AR}}}$) gets inserted into the key table.

[1] Parts of this chapter is published in ACM Symposium on Applied Computing [4]. Permission to use the material is provided in Appendix A.

Another way to model key updates is to use a global state in the protocol encoding. However, ProVerif cannot model global states in protocols. StatVerif is an extension to ProVerif that enables modeling of global states in protocol models [104]. We also modeled key updates in the private-channel protocols (Figures 3.1–3.3) using StatVerif. In this model, instead of using a key table, we used a global variable to store the secret key, and at the end of an authentication session, we updated the global variable with the new key. In both coauthentication models—using key tables and global states—we verified the same properties and obtained the same results.

To generalize coauthentication protocols, we modeled the three-device protocol shown in Figure 3.8 as an m-out-of-n protocol, where m is the number of participating devices (in this case $m = 3$), and n is the total number of devices. To model such a protocol, we created a new process in the encoding that registers arbitrarily many user devices. In the model, 3-out-of-n devices need to participate in a coauthentication session. To ensure that coauthentication mitigates compromise of $m - 1$ devices, we gave 2 user devices (i.e., the secrets stored on 2 user devices) to attackers. To make sure that all coauthentication versions mitigate compromise of $m - 1$ devices, we also modeled and verified the 4-out-of-n coauthentication version in the same way that we modeled the three-device version shown in Figure 3.8.

Each protocol session of 2-device coauthentication protocols (Figures 3.1–3.7) runs 3 processes (authenticator A, requestor R, and collaborator C), and the main ProVerif process considers arbitrarily many sessions of a protocol running concurrently. The m-out-of-n protocols have (1) additional processes for each collaborators, and (2) an additional key-generation process for registering arbitrarily many devices. For example, the 3-out-of-n protocol runs 5 processes (authenticator A, requestor R, collaborator$_1$ C_1, collaborator$_2$ C_2, and a key-generation process).

Our ProVerif encodings of the coauthentication protocols (including the stateful versions), and the properties verified, are available online [105].

4.1.2 Assumptions

The protocols were modeled and verified under the assumptions stated in Section 3.1. The private-channel protocols (Figures 3.1–3.3 and m-out-of-n protocols) have strong attack models allowing key-duplication attacks.

The all-public-channel protocols (Figures 3.4–3.7) have a weaker attack model that assumes authentication secrets (K_{AR}, K_{AC}, and K_{RC}) are only accessible to attackers through device theft. In terms of the ProVerif encodings, this weaker attack model means that, in cases where attackers are assumed to know K_{AR}, the collaborator does not respond to collaboration requests. The justification is that if an attacker has acquired K_{AR}, then by assumption the legitimate user does not possess the requestor, so collaboration requests must be for unauthorized, attacker-initiated authentications. It is assumed that, with appropriate collaboration policies, users do not approve collaborations for unauthorized authentications.

In all the protocols, attackers are active and may freely eavesdrop on, insert, delete, and modify communications. Attackers are not constrained to operate according to any of the protocols.

4.1.3 Verification Setup

Each protocol was verified in 3 runs; as shown in Table 4.1.

1. The first run began with attackers knowing no secret keys.

2. The second run began with attackers knowing all the long-term keys accessible to the collaborator. For the protocols shown in Figures 3.1–3.4, attackers were given K_{AC}; for the protocols shown in Figures 3.5–3.6, attackers were given K_{AC} and K_{RC}; for

Table 4.1: Verification setup of each protocol in three different runs.

Protocol	Attackers' Knowledge		
	Run 1	Run 2	Run 3
Figure 3.1	No secrets	K_{AC}	K_{AR} and $\widehat{\widehat{K_{AR}}}$
Figure 3.2	No secrets	K_{AC}	K_{AR} and $\widehat{\widehat{K_{AR}}}$
Figure 3.3	No secrets	K_{AC}	K_{AR} and $\widehat{\widehat{K_{AR}}}$
Figure 3.4	No secrets	K_{AC}	K_{AR}
Figure 3.5	No secrets	K_{AC} and K_{RC}	K_{AR} and K_{RC}
Figure 3.6	No secrets	K_{AC} and K_{RC}	K_{AR} and K_{RC}
Figure 3.7	No secrets	C^{-1}	R^{-1}
3-out-of-n	No secrets	All K_{AC}s	K_{AR}, $\widehat{\widehat{K_{AR}}}$, $\widehat{\widehat{\widehat{K_{AR}}}}$, and K_{AC}
4-out-of-n	No secrets	All K_{AC}s	K_{AR},$\widehat{\widehat{K_{AR}}}$, $\widehat{\widehat{\widehat{K_{AR}}}}$, $\widehat{\widehat{\widehat{\widehat{K_{AR}}}}}$,and 2 K_{AC}s

the protocol shown in Figure 3.7, attackers were given C^{-1}; and for the m-out-of-n protocols, attackers were given all the collaborator keys that are generated by the key-generation process.

3. The third run began with attackers knowing all the long-term keys accessible to the requestor in the protocols shown in Figures 3.1– 3.7. For the protocols shown in Figures 3.1–3.3, attackers were given K_{AR} and $\widehat{\widehat{K_{AR}}}$; for the protocols shown in Figures 3.5–3.6, attackers were given K_{AR} and K_{RC}; and for the protocol shown in Figure 3.7, attackers were given R^{-1}. In the third run of the m-out-of-n protocols, attackers were given all the long-term keys stored on $m-1$ devices: for the 3-out-of-n protocol, at-

tackers were given the requestor's long-term keys and one other collaborator's key; and for the 4-out-of-n protocol, attackers were given the requestor's long-term keys and two other collaborators' keys.

In all 3 runs of each of the protocols, we attempted to verify the following security properties.

4.1.3.1 P1: Secrecy of the Session Key

The session key K_{SK} is only known to the authenticator and requestor. This property subsumes forward secrecy of session keys (K_{SK}) in the third run of the private-channel protocols (Figures 3.1–3.3) because knowing the requestor's future authentication secret ($\widehat{\widehat{K_{AR}}}$, which becomes K_{AR} in the next round of authentication) does not leak session keys.

4.1.3.2 P2: Authentication of R to A

With one exception, we specified authentication of R to A as requiring that if the authenticator receives an acknowledgment of a session key (and therefore believes it shares the session key with the requestor) then the requestor was indeed its interlocutor and the collaborator indeed collaborated. This is an event-based property [106] having the form

$$endA \implies (beginA \wedge collabA),$$

where $endA$ refers to the event of A receiving the acknowledgment, $beginA$ to R sending the authentication request, and $collabA$ to C sending its participation message (in the third message of Figures 3.1, 3.2, 3.4, 3.5, and 3.7 and the second message of Figures 3.3 and 3.6). For the m-out-of-n protocols, each collaborator has its own $collabA$ event corresponding to its participation message. For example, the event-based property has the following form for

the 3-out-of-n protocol:

$$endA \implies (beginA \wedge collabA_1 \wedge collabA_2).$$

The one exception to encoding *P2* in this way is for the second run of the all-public-channel protocols (Figures 3.4– 3.7), where the attacker is given either K_{AC}, or K_{AC} and K_{RC}. In this case, the attacker may use K_{RC} to obtain, and/or K_{AC} to collaborate with, legitimate authentication requests, thus helping legitimate authentications succeed, which we do not consider an attack. Therefore, for the second run of the Figures 3.4– 3.7 protocols, we specify property *P2* as only requiring

$$endA \implies beginA,$$

that is, if the authenticator believes it shares the session key with the requestor then the requestor was indeed its interlocutor (but the attacker, rather than the collaborator, may have collaborated).

4.1.3.3 P3: Authentication of A to R

This property is symmetric to *P2* and, with one exception, requires that if the requestor sends an acknowledgment of a session key (and therefore believes it shares the session key with the authenticator) then the authenticator was indeed its interlocutor and the collaborator indeed collaborated. This property has the form

$$endR \implies (beginR \wedge collabR),$$

where $endR$ refers to R sending the acknowledgment, $beginR$ to A receiving the authentication request, and $collabR$ to C sending its participation message. Similarly to *P2*, each

Table 4.2: Formal verification results.

Protocol	*P1: Secrecy of the K_{SK}*			*P2: Authentication of R to A*			*P3: Authentication of A to R*		
	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
Figure 3.1	✓	✓	✓	✓	✓	✓	✓	✓	✓
Figure 3.2	✓	✓	✓	✻	✻	✻	✓	✓	✓
Figure 3.3	✓	✓	✓	✻	✻	✻	✓	✓	✓
Figure 3.4	✓	✓	✓	✓	✓	✓	✓	✓	✓
Figure 3.5	✓	✓	✓	✻	✓	✓	✓	✓	✓
Figure 3.6	✓	✓	✓	✻	✓	✓	✓	✓	✓
Figure 3.7	✓	✓	✓	✓	✓	✓	✓	✓	✓
3-out-of-n	✓	✓	✓	✓	✓	✓	✓	✓	✓
4-out-of-n	✓	✓	✓	✓	✓	✓	✓	✓	✓

collaborator in the m-out-of-n protocols has its own *collabR* event. For example, the event-based property has the following form for the 3-out-of-n protocol:

$$endR \implies (beginR \wedge collabR_1 \wedge collabR_2).$$

As with *P2*, the one exception to encoding *P3* in this way is for the second run of the all-public-channel protocols (Figures 3.4– 3.7), in which case *P3* only requires

$$endR \implies beginR,$$

for the same reason explained for property *P2*.

4.1.4 Verification Results

Table 4.2 shows the verification results. Cells with "✓" denote ProVerif proved the property. Cells with "✻" denote ProVerif outputted "cannot be proved" for the property. ProVerif found no attacks on any of properties *P1–P3* in any runs of any of the protocols. That is, ProVerif did not refute any of *P1–P3* in any runs of any of the protocols.

ProVerif did prove *P1* and *P3* for all 3 runs of all 10 protocols, and it proved *P2* for all 3 runs of the full coauthentication protocol (Figure 3.1), the all-public channel variation

of the full protocol (Figure 3.4), the asymmetric-key protocol (Figure 3.7), and all of the m-out-of-n protocols. It also proved *P2* for the second and third runs of the protocols shown in Figure 3.5 and 3.6.

For all runs of the protocols shown in Figures 3.2 and 3.3, and for the first runs of the protocols shown in Figure 3.5 and 3.6, ProVerif outputs that *P2* "cannot be proved". It produces traces in which a man-in-the-middle sits between A and R, and A and C, and simply collects and forwards all messages sent to and from A. This trace is not an attack because the authenticator completes the protocols with R having sent the original authentication request and C having sent its participation message, despite the fact that the attacker touched these messages while acting as an intermediary.

We also note that these results are for the stronger, injective-correspondence versions of properties *P2* and *P3*. The injective-correspondence versions require there to be a unique predecessor event for each end event [103, pp.19–22]; for example, the injective version of *P2* requires that for each *endA* event there exists a unique *beginA* predecessor event. The non-injective versions allow end events to have non-unique predecessor events. ProVerif was able to prove the weaker, non-injective version of property *P2* for all runs of all protocols.

4.2 Empirical Evaluation

We have implemented and measured the performance of full coauthentication (Figure 3.1) and the variations shown in Figures 3.2–3.7. To establish a baseline of performance, we also implemented and measured the performance of a basic password authentication system. In total, 8 authentication systems were evaluated.

4.2.1 Implementations

The password-authentication system only uses two devices (requestor and authenticator), while all of the coauthentication systems use three devices (authenticator, requestor, and collaborator).

To make performance comparisons more meaningful, the implementations were uniform to the extent possible. Each authenticator was implemented as a Java server application using Spring Boot [107], and each requestor and collaborator was implemented as an Android application. All nonces were 64-bit strings dynamically generated with Java's cryptographically strong random number generator class `java.security.SecureRandom`. All session keys in authentication-complete messages were 256-bit strings dynamically generated in the same way. The versions of coauthentication shown in Figures 3.1–3.3 also required two new keys to be generated dynamically, $\widehat{K_{AR}}$ and $\widehat{\widehat{K_{AR}}}$, again with Java's cryptographically strong random number generator. All other cryptographic keys were hardcoded, with shared keys assumed to have been shared before the implementations began running. All symmetric cryptographic operations were implemented with 256-bit CBC-mode AES, and all asymmetric cryptographic operations were implemented with HTTPS using 2048-bit RSA and self-signed certificates, through standard `javax.crypto` libraries.

To broadly mimic typical password-authentication systems, we implemented ours to run over HTTPS (again, using 2048-bit RSA and self-signed certificates). The requestor in our implementation sends the authenticator a username and password hardcoded in the requestor, with the username and password each being 8 characters because such length is common [108]. The authenticator receives and decrypts the username and password, adds salt to the password, hashes (with SHA-256), and verifies that the hash matches its hardcoded expected hash for the given username.

The public-key version of coauthentication shown in Figure 3.7 was also implemented to run over HTTPS configured in the same way. All the coauthentication protocols shown in Figures 3.1–3.6 sent all public-channel messages over TCP.

All public-channel messages, in all implementations, were sent through standard Wi-Fi channels. For communicating private-channel messages, that is, messages from the collaborator to the requestor in Figures 3.1–3.3, our implementations used Bluetooth, though it has known vulnerabilities [109].

Each run of each implementation opened new network connections, including a new Bluetooth connection in the implementations of Figures 3.1–3.3. Connections were never reused between runs of the implementations, and the Android applications were restarted for each run.

4.2.2 Experimental Setup and Results

The implementations were executed on the following devices. The authenticator was always a MacBook Pro laptop running macOS Sierra version 10.12.6 and having 16GB of memory and a 2.2GHz Intel quad-core i7 processor. Due to the popularity of mobile access to authentication services, the requestor was always a smartphone, a Samsung Galaxy s8 Plus running Android 8.0.0 and having 4GB of memory, a Qualcomm MSM 8998 octa-core (a 2.35GHz quad-core and a 1.9GHz quad-core) processor, and Bluetooth 5. The collaborator was always a Motorola Nexus 6 running Android 7.1.1 and having 3GB of memory, a 2.7GHz quad-core Qualcomm Snapdragon 805 processor, and Bluetooth 4.1.

Each of the implementations was run 100 times, in a uniform environment of normal (workday) university-network usage and standard loads of kernel and user-level applications running.

The following measurements were made for each run:

Table 4.3: Average performance of the authentication systems over 100 runs.

Implementation	Bytes Transmitted	Application-Layer Time (ms)				Authentication Time (ms)
		Authenticator	Requestor	Collaborator	Total	
Password	3212	0.28	1.50	—	1.80	136
Figure 3.1	1198	2.58	22.5	18.4	43.5	594
Figure 3.2	1088	1.36	20.1	20.9	42.4	475
Figure 3.3	885	1.16	17.2	19.4	37.8	473
Figure 3.4	835	1.88	6.18	23.5	31.6	142
Figure 3.5	1072	1.18	19.1	22.6	42.8	125
Figure 3.6	1075	0.94	14.9	23.3	39.1	131
Figure 3.7	7158	0.43	2.94	12.9	16.3	388

- The network usage, that is, the number of bytes transmitted over the course of the run. Due to unreliability in the communication channels, the number of bytes transmitted varied with each run. The network usage was measured with Android's standard network-monitoring class `android.net.TrafficStats`.

- The application-layer real time each device consumed. This measurement was made by starting a timer when beginning to process any newly received message or request, stopping the timer when finished preparing a response, taking the difference, and summing all of these times for each device. For example, the application-layer real time consumed by the authenticator in full coauthentication is the sum of the real times it consumes processing the requestor's and collaborator's messages, including generating new keys and a challenge nonce and performing the required encryptions and decryptions. Application-layer times exclude all time spent establishing connections and transmitting messages in the underlying TCP, HTTPS, and Bluetooth protocols.

- The total authentication time. This is the real time, measured on the requestor, from beginning to prepare an authentication request until finishing obtaining a plaintext session key.

Tables 4.3–4.4 summarize the results of running each implementation 100 times.

4.2.3 Performance Analysis

Many of the performance results are as expected. As shown in Table 4.3, protocols transmitting more or more complex messages, or using HTTPS, transmitted more bytes of data. Network (non-application-layer) activities dominated the performance of all implementations, consuming between 70% and 98.7% of the total authentication time on average.

As shown in Table 4.4 (CV refers to the coefficient of variation), these network activities also took a highly variable amount of time to complete, over different runs of the same implementation. The coefficients of variation (CVs) for total authentication time ranged up to 53%, indicating high variance. This variance explains the sometimes-substantial differences between median and average total authentication times observed for the same implementation. The coefficients of variation for total application-layer times were substantially smaller for all but the Figure-3.7 implementation, indicating much less variance.

In terms of application-layer performance, the password system was the most efficient and then the Figure-3.7 public-key system. Both of these systems benefit, at the application layer, from pushing all the cryptographic operations into the underlying HTTPS layer.

In terms of total authentication time, the Figure-3.6 system outperformed the others on average, and the outperformance was greater in the median case. The performance of this coauthentication system benefits from transmitting a minimal number of messages over the efficient (relative to HTTPS and Bluetooth) TCP.

Importantly, these performance results exclude human time, though it is known to be substantial for password-based authentication systems. Human entry of a password is expected to take on the order of several seconds [60, 52, 110].

Care should also be exercised when comparing the performance of the password-based system with the performance of the private-channel coauthentication systems (Figures 3.1–3.3), which update K_{AR} on every authentication. The advantages of updating K_{AR} are analogous

Table 4.4: Statistics on the performance of the authentication systems.

	Total Application-Layer Time			Total Authentication Time		
Implementation	Average (ms)	Median (ms)	CV	Average (ms)	Median (ms)	CV
Password	1.80	1.80	0.08	136	132	0.23
Figure 3.1	43.5	42.5	0.23	594	564	0.26
Figure 3.2	42.4	41.4	0.27	475	430	0.38
Figure 3.3	37.8	37.4	0.31	473	426	0.45
Figure 3.4	31.6	30.4	0.22	142	130	0.53
Figure 3.5	42.8	39.6	0.29	125	117	0.36
Figure 3.6	39.1	37.7	0.28	131	93.4	0.49
Figure 3.7	16.3	13.5	0.33	388	388	0.16

to the advantages of updating a password, so a better comparison would take into account the time required to update passwords. Password update is expected to take on the order of a minute of human time [111], significantly longer than an automatic coauthentication-key update.

We conclude from these results that coauthentication performs efficiently enough to be practical.

CHAPTER 5

SQL-IDENTIFIER INJECTION ATTACKS

This chapter [1] defines SQL-IDIAs and presents example SQL-IDIAs.

5.1 Definition of SQL-IDIAs

SQL-IDIA-vulnerable applications are applications that can form a valid SQL statement by concatenating a user-input identifier into the statement.

Definition 1. *An application is vulnerable to a* SQL-IDIA *iff the application constructs a SQL statement S by concatenating an untrusted input i into S and there exists an identifier x such that concatenating x into S in place of i causes S to be a valid SQL statement.*

For example, the application excerpted in Figure 5.1a is vulnerable to a SQL-IDIA because it can create a valid SQL statement by concatenating an identifier into the statement, as shown in Figure 5.1b.

A SQL-IDIA occurs when a SQL-IDIA-vulnerable application—which would produce a valid SQL statement by concatenating a user-input identifier into the statement—instead concatenates a non-identifier, or an invalid identifier, into the statement in place of a valid identifier.

Definition 2. *A* SQL-IDIA *occurs in a SQL-IDIA-vulnerable application iff the concatenated input i either is not an identifier or is an identifier that, when concatenated into S, makes S an invalid SQL statement.*

[1] Parts of this chapter is published in IEEE Conference on Communications and Network Security [5]. Permission to use the material is provided in Appendix A.

```
String sql = "SELECT * FROM Contact ORDER BY " + userInput;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

(a) An example Java program vulnerable to SQL-IDIAs.

```
SELECT * FROM Contact ORDER BY firstName
```

(b) The output SQL program when the `userInput` is a valid column name.

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic WHERE firstName='John' AND lastName
    ='Doe') > 0 THEN Contact.lastName ELSE Contact.firstName END)
```

(c) A malicious input through the `userInput` to perform a SQL-IDIA.

Figure 5.1: An order-by-based SQL-IDIA.

For example, a SQL-IDIA occurs when the SQL-IDIA-vulnerable application excerpted in Figure 5.1a is provided the input shown in Figure 5.1c. In this case the untrusted input (Figure 5.1c) is concatenated into the output SQL statement at a position in which an identifier could be valid, yet the untrusted input is not a valid identifier; hence, a SQL-IDIA has occurred.

Definition 2 also considers invalid-identifier injections to be SQL-IDIAs because such injections can leak sensitive database-schema information [112, 113]. For example, an attacker might input a nonexistent column name into the application shown in Figure 5.1a to cause the DBMS (Database Management System) to raise an exception when executing the generated invalid SQL statement. As with traditional SQLIAs, in cases in which the DBMS raises an exception, the application may output information contained in the exception object to leak database schema such as the database name or the SQL statement being executed.

```
String sql = "SELECT * FROM Customer WHERE " + userInput1 + " BETWEEN ? AND ?";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setInt(1, userInput2);
stmt.setInt(2, userInput3);
ResultSet rs = stmt.executeQuery();
```

(a) An example Java program vulnerable to SQL-IDIAs.

```
age BETWEEN ? AND ? UNION SELECT * FROM Admin--
```

(b) A malicious input through the `userInput1` to perform a SQL-IDIA.

```
SELECT * FROM Customer WHERE age BETWEEN ? AND ? UNION SELECT * FROM Admin --
    BETWEEN ? AND ?
```

(c) The output SQL program with the malicious input (b).

Figure 5.2: A column-name-based SQL-IDIA.

Although this dissertation focuses on SQL, identifier-injection attacks are possible in other languages such as XML, Javascript, PHP, and Python.

5.2 Additional Examples

To provide additional familiarity with SQL-IDIAs, we next consider two additional examples, shown in Figures 5.2 and 5.3. Both of these examples, as well as the example shown in Figure 5.1, are abbreviated and simplified versions of actual Java applications found by our automated GitHub analysis tool, described in Chapter 6.

Figure 5.2a shows a program that is vulnerable to a column-name-based SQL-IDIA. In this program, two user-inputs fill placeholders using prepared statements; therefore, SQLIAs are not possible through these inputs. However, a column-name parameter (i.e., `userInput1`)

```
String sql = "INSERT INTO " + userInput + "(isAdmin) VALUES ('False')";
Statement stmt = conn.createStatement();
stmt.executeUpdate(sql);
```

(a) A SQL-IDIA-vulnerable program through the table name.

```
Customer (name, isAdmin) VALUES ('Mallory', 'True'); DELETE FROM Admin; --
```

(b) A malicious input through the `userInput` to perform SQL-IDIAs.

```
INSERT INTO Customer (name, isAdmin) VALUES ('Mallory', 'True'); DELETE FROM
    Admin; -- (isAdmin) VALUES ('False')
```

(c) The output SQL program with the malicious input (b).

Figure 5.3: A table-name-based SQL-IDIA.

is concatenated into the SQL statement. In normal cases, this program expects the concatenated parameter to be a valid column name. However, attackers can perform SQL-IDIAs by injecting carefully crafted SQL statements. For instance, the program (Figure 5.2a) outputs the SQL code shown in Figure 5.2c with the malicious input shown in Figure 5.2b. This output program can maliciously return all entries from the Admin table (assuming the Customer and Admin tables have the same attributes). The malicious input is not a valid column in the Customer table, so Definition 2 correctly considers this input to be a SQL-IDIA.

Figure 5.3a shows a program that is vulnerable to a table-name-based SQL-IDIA. This program concatenates a table-name parameter (i.e., `userInput`) into a SQL statement and executes this statement using the standard JDBC (Java Database Connectivity) execute-update function. If an attacker injects the malicious input presented in Figure 5.3b through the table-name parameter, the program (Figure 5.3a) outputs the two consecutive SQL statements shown in Figure 5.3c. These statements cause two different attacks. The first

attack adds a new user to the Customer table as an administrator by changing the hardcoded admin value. The second attack—an example of piggy-backing attacks—deletes all entries from the Admin table; the malicious input presented in Figure 5.3b causes the Java program to execute multiple queries at once.

The existing JDBC API attempts to mitigate piggy-backing attacks by requiring the execute-update function to only execute one SQL statement at a time. The API provides different functions to execute multiple statements as a batch. However, in practice, some JDBC implementations do not faithfully follow the API specifications. We tested this SQL-IDIA with three different JDBC implementations: H2, SQLite, and MySQL JDBC drivers. Our results showed that these drivers, except the MySQL driver, are vulnerable to this attack. On the other hand, Definition 2 correctly classifies the input shown in Figure 5.3b as an attack because this input is not a valid table in the database.

CHAPTER 6

PREVALENCE OF SQL-INJECTION AND SQL-IDENTIFIER INJECTION ATTACKS

This chapter [1] presents an analysis of source files from GitHub and shows SQL-IDIAs on a deployed web application. This GitHub analysis investigates that how SQL statements are constructed in practice and how many of the files are vulnerable to SQLIAs and SQL-IDIAs.

6.1 Research Questions

To understand the prevalence of SQLIAs and SQL-IDIAs, we ask the following research questions:

R_1 What percentage of source files use prepared statements, string concatenation, or hard-coded strings for constructing SQL statements?

R_2 What percentage of source files use both prepared statements and string concatenation in the same file?

R_3 What percentage of source files use identifier concatenation for constructing SQL statements?

R_4 What type of identifiers are the most commonly concatenated?

Our GitHub analysis provides empirical answers to these questions.

[1] Parts of this chapter is published in IEEE Conference on Communications and Network Security [5]. Permission to use the material is provided in Appendix A.

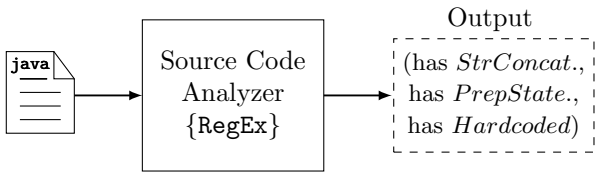


Figure 6.1: The operation of the source file analyzer.

6.2 Dataset Collection

We collected a dataset of source files from GitHub, which is the most popular platform to publish open source projects. We used Java source files because Java is one of the most commonly used programming languages [114]. We used GitHub Archive [115] because the GitHub website provides limited access to all source files. GitHub Archive is a public database that collects all public GitHub activities (e.g., source files, commits, pull requests) since 2011. Google BigQuery [116], which provides an interface to perform database operations on large data, was utilized to access GitHub Archive.

Each file in our dataset contains SQL statements. To determine the files that contain SQL statements, we filtered GitHub Archive with certain keywords (i.e., "executeQuery", "executeUpdate", "createQuery", and "createNativeQuery"). These keywords are used to execute SQL statements in Java. It is noteworthy that these keywords are the functions of the well-known database libraries [117]. To minimize redundancy and false positives, we only considered parent projects (i.e., the projects that are not forked from other projects) and eliminated unit-test files. Our final dataset contains 120,412 Java source files, obtained by filtering 56.7 million Java files.

6.3 Identifying SQL Usages

We developed an automated tool to determine SQL usages in each source file. Figure 6.1 depicts the source file analysis operation using this automated tool. As shown in the figure, this tool takes an individual Java source file as an input, matches regular expressions to

identify how SQL statements are constructed in the file, and outputs SQL-construction types in three categories. The first category is the dynamic SQL-statement construction using string concatenation (e.g., `"SELECT * FROM table WHERE id=" + userInput`). The second category is the dynamic SQL statement construction with prepared statements (e.g., `"SELECT * FROM table WHERE id=?"`). The last category is the static SQL-statement usage with hardcoded string literals (e.g., `"SELECT * FROM table WHERE id=5"`).

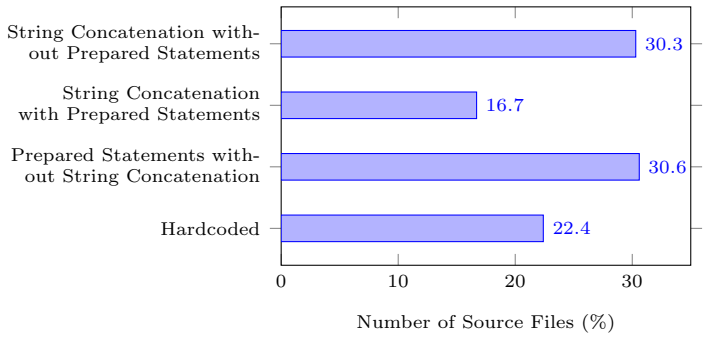
These SQL-construction categories are detected with rules that are encoded in regular expressions. If one of the following rules is matched, a file is classified as belonging to the string concatenation category.

- Concatenating a string literal with a Java identifier (e.g., a variable name) using the plus operator, where the string literal contains SQL keywords. For example, this rule matches `"SELECT * FROM " + userInput`.

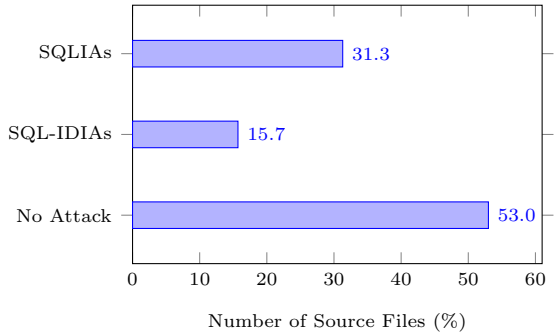
- Using at least two append functions, where at least one of the append function takes a Java identifier as an argument and the other takes a string literal containing SQL keywords. For example, this rule matches `append ("SELECT * FROM ") .append( userInput)`.

- Using a Java string-format function that contains at least one string literal with SQL keywords and string-format placeholders. For example, this rule matches `String. format ( "SELECT * FROM %s", userInput)`.

To determine prepared-statement usage, we encoded a rule that matches a string literal containing SQL keywords and prepared-statement placeholders (i.e., `?` and `:name`). A file is considered as using hardcoded SQL-statements if the file (1) does not contain a string concatenation to form SQL statements, (2) does not contain prepared statements, and (3) only contains string literals with SQL keywords.



(a) SQL-construction statistics of 120,412 Java source files.



(b) Injection-attack vulnerabilities of 120,412 Java source files.

Figure 6.2: SQL-construction and injection-attack vulnerability statistics.

These SQL-construction categories determine the files that are vulnerable to SQLIAs and SQL-IDIAs. Assume that the concatenated variables are propagated from untrusted inputs. Then, the source file is considered as vulnerable if it uses at least one string concatenation to construct SQL statements [15, 14]. If a source file does not use string concatenation and employs prepared statements and/or hardcoded SQL-statements, then the file is considered as not vulnerable.

6.4 Empirical Results

This section discusses the prevalence of SQLIAs and SQL-IDIAs by empirically answering each of the research questions.

R_1 What percentage of source files use prepared statements, string concatenation, or hardcoded strings for constructing SQL statements?

Figure 6.2a summarizes the SQL-usage statistics of 120,412 Java source files that contain SQL statements. 30.3% of the files construct SQL statements using string concatenation, 30.6% of the files employ prepared statements to form SQL statements, and 22.4% of the files only use hardcoded SQL statements. These results reveal that a significant portion of source files are vulnerable to SQLIAs through string concatenation.

R_2 What percentage of source files use both prepared statements and string concatenation in the same file?

Interestingly, 16.7% of the files have both string concatenation and prepared statements. The following answer to R_3 shows that one reason for using both string concatenation and prepared statements in the same file is the identifier limitation of the existing prepared-statement API.

One unusual reason for using both string concatenation and prepared statements in the same file is that the cases requiring extra inconvenient steps to apply prepared statements. For example, the usage of the `LIKE` operator. This operator is used for searching texts with specified patterns in columns. However, SQL syntax does not allow these patterns to be defined in the prepared statement. Instead, these patterns can be (1) dynamically concatenated with an untrusted input and then (2) the concatenated-input can be used to fill the placeholder in the prepared statement. Our analysis reveals that 1.4% of files concatenate strings to construct SQL clauses with `LIKE` operators.

R_3 What percentage of source files use identifier concatenation for constructing SQL statements?

Based on the GitHub analysis, 9.6% of the files use only identifier concatenation to form SQL statements. Additionally, 6.1% of the files concatenate identifiers and also employ prepared statements for values in the same file.

R_4 What type of identifiers are the most commonly concatenated?

We analyzed the types of identifiers (e.g., table, column, index, function, procedure names) being concatenated. Based on the analysis, 96% of the identifiers were table and column names.

As a result, the GitHub analysis revealed that identifier concatenation is a real problem. As shown in Figure 6.2b, 15.7% of the files are vulnerable to SQL-IDIAs. In addition, 31.3% of the files are vulnerable to SQLIAs through string concatenation.

6.5 Attacking a Deployed Application

We present SQL-IDIAs on a large-scale Electronic Medical Record software. As of March 17, 2019, this open-source software is actively maintained and has 21,844 total commits, 61 contributors, and 23 stars at GitHub. Features of this software include managing confidential patient and medication records, scheduling appointments, and managing hospital-related tasks. Our GitHub analysis showed that a Java file in this software concatenates an identifier to form a SQL statement. By manually inspecting the source code, we were able to verify that this software is indeed vulnerable to SQL-IDIAs. We set up an attack environment by running this Electronic Medical Record software on a local computer and creating a test database. Although we have disclosed the identified vulnerability, we do not name the software here because it is widely deployed, and because the vulnerability has not been resolved (as of March 17, 2019).

The software has a web page used to search for employees in a clinic. Users can type a search keyword through this web page, and the software runs the code shown in abbreviated form in Figure 6.3a. The `keyword` parameter is used with prepared statements; therefore, a SQLIA is not possible through this parameter. However, this program takes the `userInput` parameter for the order-by clause from a hidden form in the web page. Thus, attackers can

```
String sql = "SELECT * FROM Contact WHERE lastName LIKE ? ORDER BY " + userInput;
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setString(1, keyword);
ResultSet rs = stmt.executeQuery();
```

(a) An abbreviated version of actual SQL-IDIA-vulnerable code in Electronic Medical Record software.

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic WHERE firstName='John' AND lastName
    ='Doe') > 0 THEN Contact.lastName ELSE Contact.firstName END)
```

(b) A malicious input through the `userInput` parameter to test the patient name in the database.

```
(CASE WHEN (SELECT COUNT(*) FROM Demographic WHERE firstName='John' AND lastName
    ='Doe' AND demographicNo<5) > 0 THEN Contact.lastName ELSE Contact.firstName
    END)
```

(c) A malicious input through the `userInput` parameter to determine the unique identifier of the patient in the database.

```
(CASE WHEN (SELECT COUNT(*) FROM Appointment WHERE demographicNo=1 AND reasonCode
    <5) > 0 THEN Contact.lastName ELSE Contact.firstName END)
```

(d) A malicious input through the `userInput` parameter to determine the patient's doctor-visit reason.

Figure 6.3: SQL-IDIAs on Electronic Medical Record software.

perform SQL-IDIAs through this order-by parameter by changing the source of the web page in a web browser.

In our first example attack, we injected the SQL expression shown in Figure 6.3b through the order-by parameter to determine whether a person named John Doe is a patient in the clinic. If John Doe appears in the Demographic table, the search result from the Contact

table was sorted by the last name; otherwise, the result was sorted by the first name. This enabled us to determine that John Doe is a patient in the clinic.

To access other confidential information, we need to determine the unique identifier (i.e., `demographicNo`) of John Doe from the Demographic table. This identifier, which manages the relations between tables, can be extracted by injecting the code shown in Figure 6.3c. This code tests whether the unique identifier of John Doe is less than 5. Performing a binary search allowed us to determine the actual value (i.e., 1) of the unique identifier.

The extracted unique identifier of John can be used to obtain additional confidential information from other tables. For example, the code presented in Figure 6.3d maliciously detects John's doctor-visit reason. We determined, via binary search, that the reason code for John's doctor visit was 7. With manual inspection of the reason codes in the open source project, we were able to determine that the visit was made for HIV testing.

This technique is not limited to the attacks above. The same technique can be used to extract different confidential information about patients such as medication history, laboratory test results, patient address, or the room that a patient is occupying.

6.6 Threats to Validity

Our GitHub-analysis methodology is based on pattern matching in single files. This approach relies on the assumption that concatenated variables are propagated from untrusted inputs. This assumption was made because determining the input source for concatenated variables would require techniques such as data flow analysis [118] and compiler optimization [119]. These techniques require source files to be compiled, thus they cannot be applied to our analysis. This limitation exists because our dataset only contains independent source files, not the whole projects and their dependencies.

To estimate the performance of our GitHub-analysis results, we randomly selected and manually inspected 200 source files. The false positive is determined when the tool says

there is a string concatenation in the file, but in reality (1) the string concatenation is not used to form a SQL statement, (2) the concatenated variable is inside of the file (e.g., a SQL statement is concatenated with a static field), or (3) the concatenated variable is commented out. The false negative is determined when the tool says there is no string concatenation in the file, but there is actually at least one SQL statement that is created using string concatenation, and the concatenated variable is coming from outside of the source file. To estimate the accuracy, we calculated the proportion of true results (i.e., true positive and negative rates) in all results.

This methodology was used to estimate the performance of our analysis. The false positive rate, false negative rate, and the accuracy of our GitHub analysis are 18%, 7%, and 87%, respectively. The obtained results are promising. However, these results, similarly to existing research relying on GitHub data, may suffer from different threats as discussed in [120].

CHAPTER 7

PREVENTING SQL-IDENTIFIER INJECTION ATTACKS

In this chapter [1], we introduce a new extended prepared-statement API. This API

- enables prepare statements to fill placeholders with table and column names,
- prevents SQL-IDIAs,
- does not leak schema information, and
- does not have limitations that the input-sanitation-based approaches have.

This section also demonstrates the implementability, efficiency, and effectiveness of the extended API by presenting an empirical evaluation of a prototype implementation.

7.1 API Functions

To prevent SQL-IDIAs, we introduce the following two functions that can be added to the prepared-statement APIs (e.g., Java JDBC [121], PHP PDO [122]).

- `setColumnName(int parameterIndex, String columnName)`: takes a column name and its index as arguments,
- `setTableName(int parameterIndex, String tableName)`: takes a table name and its index as arguments,

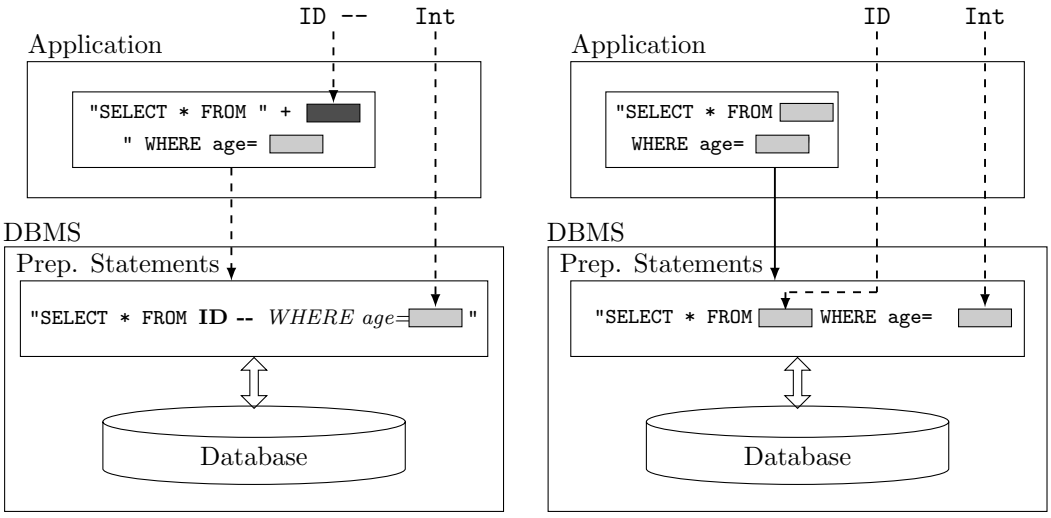
[1] Parts of this chapter is published in IEEE Conference on Communications and Network Security [5]. Permission to use the material is provided in Appendix A.

A possible implementation of these functions consists of three main steps. First, these functions can be added to the prepared-statement API and its corresponding database driver (e.g., the MySQL JDBC Driver). The implementation of these new functions in this step is similar to the existing prepared-statement functions, such as `setString`. Typically, when these new functions are called, their parameters can be stored in an array with parameter indices. These indices indicate the placeholder positions in SQL statements.

Second, the SQL-statement preparation phase for identifiers can be implemented in the DBMS. The standard preparation phase contains two main steps: (1) parsing the SQL statement, and (2) generating an execution plan. The implementation of the parsing step may require changing the SQL syntax of the database in some cases. For example, the syntax does not need to be changed if the databases allow placeholders anywhere in the SQL-statement. The syntax has to be modified to allow placeholders for table and columns if the database syntax only allows certain clauses to have placeholders.

The execution-plan-generation step can include schema verification and statement optimization. In the schema verification, the DBMS checks whether the table and column names in the SQL-statements are valid. For example, given the statement `SELECT id FROM Customer`, the DBMS checks whether the Customer table is in the database and id is an attribute of the Customer table. Although the DBMS can still verify and optimize the non-parameterized table and column names in this step, the verification of parameterized table and column names must be performed while executing the prepared statement.

The last step can involve filling the placeholders with identifiers while executing the prepared statement. This step starts by checking whether dynamic identifiers belong to the schema. The checking operation is straightforward in the case of column names because the DBMS only needs to ensure that a given column belongs to an appropriate table. Dynamically checking table names requires further verification including verifying whether the table belongs to the schema as well as ensuring that the already existing attributes in the SQL



(a) SQL-IDIAs with the existing prepared-statement APIs.

(b) Preventing SQL-IDIAs with the extended prepared-statement API that has identifier capabilities.

Figure 7.1: Preventing SQL-IDIAs with the extended prepared-statement API.

statement belong to the given table. Once the verification is complete, the DBMS can create an expression for each parameterized identifier and place these expressions into the prepared statement.

We only considered table and column names in our extended API because our GitHub analysis showed that 96% of the concatenated identifiers were table and column names.

7.1.1 Benefits of the Extended API

An illustration of the systems that are vulnerable to SQL-IDIAs due to the usage of existing prepared-statement APIs is shown in Figure 7.1a. As can be seen, an application (1) takes literals and identifiers as inputs, (2) fills placeholders with literals using prepared statements, and (3) concatenates identifiers to construct SQL statements. The identifier concatenation causes applications to have SQL-IDIA vulnerabilities.

As illustrated in Figure 7.1b, the extended API prevents SQL-IDIAs by filling placeholders with identifiers using prepared statements. Applications can create placeholders for

identifiers using the extended prepared-statement API, and the API only allows these placeholders to be filled with valid identifiers. Thus, attackers are not able to perform SQL-IDIAs.

The extended API prevents DBMSs from leaking sensitive schema information by performing a default operation when the input column or table name does not exist in the database. For example, if a parameterized column name is used in an order-by clause and the column name is invalid, the DBMS will order the results by the first column in the table. This operation prevents the information-leakage attack described in Chapter 5.

In addition, these extended API functions do not suffer from the drawbacks of input-sanitation-based approaches. For example, as described in Chapter 2, incorrect updates to whitelists or blacklists may introduce false positives or false negatives. The extended API functions eliminate such false positives or negatives by dynamically verifying given table and column names in databases before filling placeholders.

7.2 Empirical Evaluation

A prototype of the extended prepared-statement API was implemented, and the implementation was compared with an existing equivalent prepared-statement function as well as ad hoc whitelisting solutions.

7.2.1 Implementation

We implemented a prototype of the `setColumnName` function into the H2 JDBC library [123]. H2 is an open-source relational database management system that is written in Java.

The implementation enables order-by clauses to have column names through the new `setColumnName` function. In our implementation, we have not modified H2's SQL syntax because it allows order-by clauses to have placeholders for values; in fact, order-by clauses can take numerical column indices as parameters with prepared statements.

```
String sql = "SELECT * FROM TestTable WHERE col2 < 100 ORDER BY ? ASC";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setColumnName(1, userInput);
ResultSet rs = stmt.executeQuery();
```

Figure 7.2: Usage of the new `setColumnName` function.

Figure 7.2 shows a program that employs the implemented `setColumnName` function. This program selects entries from a table and orders them by the given column name. At prepare time, when the prepare-statement function is executed, the H2 DBMS parses the SQL statement and creates a query structure having a placeholder for the order-by parameter. When the `setColumnName` is executed, the DBMS stores the column name parameter with its index in an array. Once the execute-query function is executed, the DBMS first validates the column name. If the given column name is invalid, i.e., does not belong to the table, the DBMS sorts the results by the first column in the table to prevent information-leakage attacks through error messages. If the column is valid, the DBMS (1) dynamically creates a column expression, (2) appends this expression to the query structure, and (3) executes the query.

7.2.2 Experimental Setup

We compared our `setColumnName` implementation with three different implementations: an existing prepared-statement function and two different ad hoc implementations. Our implementation executes the query shown in Figure 7.2, and the three other implementations execute equivalent queries. Hence, all of the implementations return the same result-set in the same order when the input is the same.

To establish a baseline, we used the existing prepared-statement API's `setInt` function that takes an int-literal as a parameter. By filling the placeholder shown in Figure 7.2 with a column index using the `setInt` function, we were able to create an equivalent query with

`setColumnName`. We could not use other existing prepared-statement functions because an equivalent query cannot be created with any other functions. We also compared our implementation with two different ad hoc solutions. The first ad hoc implementation uses a static-whitelist (i.e., a hash set that contains all column names in the table). The second ad hoc implementation employs a dynamic-whitelist by first querying whether the given column name exists in the database and then executing the actual query.

The `setColumnName` and `setInt` implementations prepare a statement once and execute the statement 100 times. The ad hoc implementations prepare and execute the statement 100 times because column names had to be concatenated into queries. In each execution, we measured the execution time, that is, the real-time. For the first two prepared-statement implementations, the real-time is measured from beginning to setting placeholders and executing the query until finishing obtaining a result-set from the database. For the ad hoc implementations, the real-time is measured from beginning to preparing a statement and executing the query until finishing obtaining a result-set from the database.

We tested all four implementations using a uniform environment. The testing database has a table that contains 100 columns and 1000 rows. Each cell of the table was filled with a random number between 0 and 1000. These random numbers were generated using the standard Java random number library. We used the H2 DBMS to implement the database-relevant operations. All experiments were performed on a MacBook Pro laptop that runs macOS Sierra version 10.12.6 with 16GB of memory and a 2.2GHz Intel quad-core i7 processor.

We conducted three sets of experiments to test the performance of the implementations. In the first experiment, each implementation was given the same column name or column index. In the second experiment, a randomly chosen valid column name or index was given to each implementation in each run, to eliminate caching. In the last experiment, each

Table 7.1: Average execution times of the implementations over 100 runs.

Implementation	Execution Time (ms)		
	Same Input	Random Input	Bad Input
New `setColumnName`	2.11	2.25	2.04
Existing `setInt`	2.13	2.29	1.11
Static Whitelist	2.08	2.18	2.29
Dynamic Whitelist	2.37	4.73	4.07

implementation was given a "bad" input, meaning a randomly chosen column that is not an attribute of the table.

7.2.3 Experimental Results

Table 7.1 summarizes the performance results of the four implementations. Our implementation has no extra performance overhead over the existing prepared-statement `setInt` function when the input is the same or a random input is provided. For the bad inputs, `setInt` outperformed `setColumnName` because `setInt` does not retrieve a result set from the table and instead throws an exception containing sensitive schema information. In contrast, our implementation returns a result set that is sorted by the first column in the table to prevent information-leakage attacks.

In all experiments, the new `setColumnName` function outperformed the dynamic-whitelist implementation. The static-whitelist implementation slightly outperformed the new set column name function in two experiments. Although this ad hoc approach has a slight performance advantage, whitelisting approaches may introduce nontrivial complexities into application code and may lead to false positives (see Chapter 2).

To test the effectiveness of our implementation, we mounted the order-by-based SQL-IDIAs described in Figures 5.1 and 6.3, and the information-leakage attack described in Chapter 5. The new `setColumnName` function successfully prevented all of these attacks.

To summarize, filling placeholders with column names (1) is practical and efficient as compared to the existing ad hoc approaches, (2) does not introduce extra performance overheads as compared to the existing prepared-statement functions, and (3) is effective against SQL-IDIAs.

CHAPTER 8
CONCLUSIONS

This dissertation addressed two "most critical" web-application security vulnerabilities by presenting (1) a novel authentication technique called coauthentication and (2) a class of SQL-Injection attacks called SQL-Identifier injection attack [1].

8.1 Summary

The coauthentication protocols and system designs have several potential benefits. Coauthentication:

- protects against compromise of any one authentication secret, similar to multi-factor techniques but without the inconveniences of having to enter passwords (including OTPs) or scan biometrics;
- requires little, and in some implementations no, interaction from users;
- mitigates phishing, replay, and man-in-the-middle attacks (there are no passwords to phish, and the attack models assume active attackers);
- bases authentications on high-entropy secrets that can be generated, exchanged, stored, updated, and used automatically and efficiently (in contrast with password and biometric secrets);
- can implement advanced functionalities, including m-out-of-n, shared-device, continuous, group, and anonymous authentications;

- has formally verified security properties;
- has been implemented and found to perform efficiently enough to be practical;
- can be combined with additional authentication factors;
- provides protocols that may benefit existing multi-device authentication systems, such as those based on OTPs.

Another benefit of coauthentication is its ability to reset secrets automatically. With existing systems, if a nonuser does obtain the required secrets, then resetting the secrets is laborious (e.g., for the victim to reset a password), expensive (e.g., to send the user a new physical token), or impractical (e.g., to give a user new fingerprints, retinas, vocal profile, etc). In contrast, coauthentication secrets may be cryptographic keys stored on registered devices; these keys may be reset, and periodically updated, automatically. These keys can also be generated to have high entropy, without concern for whether users can create, memorize, or enter the high-entropy secrets.

Because users never enter coauthentication secrets, these secrets cannot be phished by convincing users to enter them. In contrast, passwords are often obtained by convincing users to enter them as part of phishing attacks [59].

Given these benefits, coauthentication, or a multi-factor authentication with coauthentication as the physical-token factor, may be advantageous for some authentication applications.

This dissertation has also defined SQL-IDIAs and demonstrated example SQL-IDIAs on deployed software. To prevent SQL-IDIAs, a new extended prepared-statement API was proposed. This API

- extends the safe use of prepared statements by filling placeholders with table and column names,

- prevents SQL-IDIAs,
- does not leak schema information on invalid inputs,
- does not have drawbacks that existing input-sanitation-based solutions have,
- has been prototyped, and found to perform efficiently and effectively, and
- can be utilized by the existing automatic prepared-statement-generation tools.

The prevalence of SQL-IDIAs was determined by GitHub SQL-construction analysis. The GitHub analysis showed that 15.7% of the SQL-constructing Java source files considered are vulnerable to SQL-IDIAs. These SQL-IDIA vulnerabilities can be prevented with successful adoption of the proposed extended prepared-statement API.

8.2 Future Work

This section addresses several possible research directions.

8.2.1 Quantum Coauthentication

Quantum Key Distribution (QKD) is a technique for sharing secret keys using quantum mechanics [124, 125]. QKD systems use classical and quantum channels to distribute secret keys between two distant devices. The security of QKD techniques relies on the no-cloning theorem [126, 127]. Based on the no-cloning theorem, any measurement on a quantum channel will disturb the original message. Therefore, passive attacks such as eavesdropping attempts become noticeable by receivers [128]. However, QKD schemes are still vulnerable to active attacks such as man-in-the-middle [128].

To mitigate active attacks many quantum authentication schemes have been proposed (e.g., [129, 130, 131]). These schemes can be implemented using many different approaches

including quantum entanglement, polarization states, or combination of quantum and classical cryptographic techniques. As far as we are aware, all of these approaches still rely on the no-cloning theorem to mitigate active attacks. Although these techniques mitigate active attacks, an eavesdropping attempt may disturb the original message, therefore resulting in a denial-of-service attack.

One avenue for future work may be to investigate the quantum coauthentication techniques that can mitigate denial-of-service attacks using multiple registered quantum devices.

8.2.2 Parallel Security-Protocol Verification

Automated protocol verification plays a vital role to verify security properties of network protocols [132]. In this dissertation, we used ProVerif, which is one of the most popular tools to verify protocols, to verify coauthentication protocols. However, ProVerif verified the complex m-out-of-n coauthentication protocols in approximately 6 weeks. Based on our observation, ProVerif only uses a single processing core to verify a protocol property.

Another possible avenue for future work may focus on verifying security protocols in parallel. That is, multiple processing cores may work together to verify a single security property. A plug-in to ProVerif, that enables the usage of multiple cores, may improve the complex-security-protocol verification performance.

8.2.3 Additional Modifications to the Prepared-Statements APIs

Another future direction can be the additional modifications to prepared statements for mitigating SQLIAs. For example, our Github analysis showed that 1.4% of files concatenate strings to construct SQL clauses with `LIKE` operators. Although the existing prepared-statement functions can be used with `LIKE` operators, as discussed in Chapter 6, using prepared statements with `LIKE` operators requires extra inconvenient development-steps. Another avenue for future work may investigate developers' behaviors on possible inconve-

nient steps, and introduce additional functions to the prepared statements that eliminate these extra inconvenient steps.

This future work can also consider implementing the new API functions and extensively evaluating their performance. The performance of each new function may be different. For example, dynamically setting a table name may take more time than a column name. The performance difference between API functions may exist because dynamically setting table name requires more checks (e.g., checking all attributes belong to the table) than dynamically setting column name. Additionally, each SQL clause may have different performance results. For example, using the `setColumnName` function might be faster for `order by` than `select` clauses. Such a performance analysis can be used for fine-tuning SQL queries.

8.2.4 Compiling Source Files Without Dependencies

Our GitHub-analysis results rely on pattern matching in single files. To reduce false positive and negatives, data flow analysis and compiler optimization techniques can be implemented in single source files. To implement these techniques, the source files are needed to be compiled. As far as we are aware, there is no such a technique that can compile a single source file without having its dependencies. It would be useful to develop such a system that can compile single source files and perform SQL-construction analysis based on data flow analysis.

LIST OF REFERENCES

[1] OWASP, "Owasp top 10 – 2017 The ten most critical web application security risks." https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2017.

[2] J. A. Ligatti, D. Goldgof, C. Cetin, and J.-B. Subils, "Systems and methods for anonymous authentication using multiple devices," June 28 2016. US Patent 9,380,058.

[3] J. A. Ligatti, D. Goldgof, C. Cetin, and J.-B. Subils, "System and methods for authentication using multiple devices," May 23 2017. US Patent 9,659,160.

[4] J. Ligatti, C. Cetin, S. Engram, J.-B. Subils, and D. Goldgof, "Coauthentication," in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Apr. 2019.

[5] C. Cetin, J. Ligatti, and D. Goldgof, "SQL-Identifier injection attacks," in *2019 IEEE Conference on Communications and Network Security (CNS) (IEEE CNS 2019)*, 2019.

[6] J. Ligatti, C. Cetin, S. Engram, J.-B. Subils, and D. Goldgof, "Coauthentication," Tech. Rep. Technical Report CSE-SEC-092418, University of South Florida, Department of Computer Science and Engineering, Sept. 2018. http://www.cse.usf.edu/ ligatti/papers/CoauthTR-092418.pdf.

[7] J. Ligatti, C. Cetin, S. Engram, J.-B. Subils, and D. Goldgof, "Coauthentication," Tech. Rep. Auth-7-17-17, University of South Florida, Department of Computer Science and Engineering, July 2017. http://www.cse.usf.edu/ ligatti/papers/coauth-TR.pdf.

[8] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings of the IEEE Computer Security Foundations Workshop*, pp. 82–96, June 2001.

[9] B. Blanchet, "ProVerif: Cryptographic protocol verifier in the formal model," 2016. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[10] D. Watson, "Web application attacks," *Network Security*, vol. 2007, no. 10, pp. 10–14, 2007.

[11] W. G. Halfond, J. Viegas, A. Orso, *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, vol. 1, pp. 13–15, IEEE, 2006.

[12] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," in *Proceedings of the International Conference on Applied Cryptography and Network Security*, pp. 292–302, 2004.

[13] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 372–382, 2006.

[14] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 179–190, 2012.

[15] D. Ray and J. Ligatti, "Defining injection attacks," in *Proceedings of the 17th International Information Security Conference*, pp. 425–441, 2014.

[16] R. J. Anderson, "Attack on server assisted authentication protocols," *Electronics Letters*, vol. 28, no. 15, p. 1473, 1992.

[17] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond, "Chip and pin is broken," in *2010 IEEE Symposium on Security and Privacy*, pp. 433–446, IEEE, 2010.

[18] H.-B. Chen, T.-H. Chen, W.-B. Lee, and C.-C. Chang, "Security enhancement for a three-party encrypted key exchange protocol against undetectable on-line password guessing attacks," *Computer Standards & Interfaces*, vol. 30, no. 1-2, pp. 95–99, 2008.

[19] D. P. Jablon, "Extended password key exchange protocols immune to dictionary attack," in *Proceedings of IEEE 6th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 248–255, IEEE, 1997.

[20] D. S. Dakun Shen, Ian Markwood and Y. Liu, "Virtual safe: Unauthorized walking behavior detection for mobile devices," *IEEE Transactions on Mobile Computing*, 2018.

[21] D. S. Dakun Shen, Ian Markwood and Y. Liu, "Virtual safe: Unauthorized movement detection for mobile devices," in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016.

[22] J. Grossman, S. Fogie, R. Hansen, A. Rager, and P. D. Petkov, *XSS attacks: cross site scripting exploits and defense*. Syngress, 2007.

[23] J. Fonseca, M. Vieira, and H. Madeira, "Testing and comparing web vulnerability scanning tools for sql injection and xss attacks," in *13th Pacific Rim international symposium on dependable computing (PRDC 2007)*, pp. 365–372, IEEE, 2007.

[24] P. Bisht and V. Venkatakrishnan, "Xss-guard: precise dynamic prevention of cross-site scripting attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–43, Springer, 2008.

[25] S. Gupta and B. B. Gupta, "Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512–530, 2017.

[26] B. Eshete, A. Villafiorita, and K. Weldemariam, "Early detection of security misconfiguration vulnerabilities in web applications," in *2011 Sixth International Conference on Availability, Reliability and Security*, pp. 169–174, IEEE, 2011.

[27] F. Cuppens, N. Cuppens-Boulahia, and J. Alfaro, "Misconfiguration management of network security components," in *IASTED International Conference on Communication, Network, and Information Security (CNIS 2005)*, pp. 1–10, 2005.

[28] I. M. Khalil, A. Khreishah, S. Bouktif, and A. Ahmad, "Security concerns in cloud computing," in *2013 10th International Conference on Information Technology: New Generations*, pp. 411–416, IEEE, 2013.

[29] C. Joshi and U. K. Singh, "Security testing and assessment of vulnerability scannersin quest of current information security landscape," *International Journal of Computer Applications*, vol. 145, no. 2, pp. 1–7, 2016.

[30] Y. L. Ian Markwood, Dakun Shen and Z. Lu, "PDF mirage: Content masking attack against information-based online services," in *26th USENIX Security Symposium (USENIX Security 17)*, pp. 833–847, 2017.

[31] P. Ferguson, "Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing," 2000.

[32] J. Bellardo and S. Savage, "802.11 denial-of-service attacks: Real vulnerabilities and practical solutions.," in *USENIX security symposium*, vol. 12, pp. 2–2, Washington DC, 2003.

[33] A. Hussain, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 99–110, ACM, 2003.

[34] F. Lau, S. H. Rubin, M. H. Smith, and L. Trajkovic, "Distributed denial of service attacks," in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, vol. 3, pp. 2275–2280, IEEE, 2000.

[35] T. Wang, Y. Liu, and A. V. Vasilakos, "Survey on channel reciprocity based key establishment techniques for wireless systems," *Wireless Networks*, vol. 21, no. 6, pp. 1835–1846, 2015.

[36] T. Wang, Y. Liu, and J. Ligatti, "Fingerprinting far proximity from radio emissions," in *European Symposium on Research in Computer Security*, pp. 508–525, Springer, 2014.

[37] T. Wang and Y. Liu, "Secure distance indicator leveraging wireless link signatures," in *2014 IEEE Military Communications Conference*, pp. 222–227, IEEE, 2014.

[38] T. Wang, Y. Liu, Q. Pei, and T. Hou, "Location-restricted services access control leveraging pinpoint waveforming," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 292–303, ACM, 2015.

[39] S. Fang, Y. Liu, W. Shen, H. Zhu, and T. Wang, "Virtual multipath attack and defense for location distinction in wireless networks," *IEEE Transactions on Mobile Computing*, vol. 16, no. 2, pp. 566–580, 2016.

[40] T. Wang, Y. Liu, T. Hou, Q. Pei, and S. Fang, "Signal entanglement based pinpoint waveforming for location-restricted service access control," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 853–867, 2018.

[41] S. Fang, T. Wang, Y. Liu, S. Zhao, and Z. Lu, "Entrapment for wireless eavesdroppers,"

[42] L. O'Gorman, "Comparing passwords, tokens, and biometrics for user authentication," *Proceedings of the IEEE*, vol. 91, pp. 2021–2040, Dec. 2003.

[43] H. Zhao and X. Li, "S3pas: A scalable shoulder-surfing resistant textual-graphical password authentication scheme," in *IEEE 21st International Conference on Advanced Information Networking and Applications Workshops*, vol. 2, pp. 467–472, Aug. 2007.

[44] Y. Ding and P. Horster, "Undetectable on-line password guessing attacks," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 4, pp. 77–86, 1995.

[45] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 364–372, ACM, 2005.

[46] S. W. Shin, M.-K. Lee, D. Moon, and K. Moon, "Dictionary attack on functional transform-based cancelable fingerprint templates," *ETRI journal*, vol. 31, no. 5, pp. 628–630, 2009.

[47] C.-L. Lin, H.-M. Sun, and T. Hwang, "Attacks and solutions on strong-password authentication," *IEICE transactions on communications*, vol. 84, no. 9, pp. 2622–2627, 2001.

[48] C.-L. Lin and T. Hwang, "A password authentication scheme with secure password updating," *Computers & Security*, vol. 22, no. 1, pp. 68–72, 2003.

[49] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *Proceedings of the 11th international conference on World Wide Web*, pp. 293–304, ACM, 2002.

[50] B. Pinkas and T. Sander, "Securing passwords against dictionary attacks," in *Proceedings of the 9th ACM conference on Computer and communications security*, pp. 161–170, ACM, 2002.

[51] P. G. Inglesant and M. A. Sasse, "The true cost of unusable password policies: Password use in the wild," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 383–392, April 2010.

[52] W. Melicher, D. Kurilova, S. M. Segreti, P. Kalvani, R. Shay, B. Ur, L. Bauer, N. Christin, L. F. Cranor, and M. L. Mazurek, "Usability and security of text passwords on mobile devices," in *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 527–539, May 2016.

[53] A. J. Aviv, J. T. Davin, F. Wolf, and R. Kuber, "Towards baselines for shoulder surfing on mobile authentication," in *ACM Proceedings of the 33rd Annual Computer Security Applications Conference*, pp. 486–498, Dec. 2017.

[54] C. S. Weir, G. Douglas, M. Carruthers, and M. Jack, "User perceptions of security, convenience and usability for ebanking authentication tokens," *Computers & Security*, vol. 28, no. 1, pp. 47–62, 2009.

[55] N. I. of Standards and Technology, "Back to basics: Multi-factor authentication (MFA)," Nov. 2016. https://www.nist.gov/itl/tig/back-basics-multi-factor-authentication.

[56] P. Grassi, J. Fenton, E. Newton, R. Perlner, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, Y.-Y. Choong, K. Greene, and M. Theofanos, "NIST special publication 800-63B digital authentication guideline," June 2017. https://doi.org/10.6028/NIST.SP.800-63b.

[57] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "TOTP: Time-based one-time password algorithm," RFC 6238, May 2011. http://www.rfc-editor.org/rfc/rfc6238.txt.

[58] M. A. Sasse, S. Brostoff, and D. Weirich, "Transforming the 'weakest link'–a human/computer interaction approach to usable and effective security," *BT Technology Journal*, vol. 19, pp. 122–131, July 2001.

[59] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the International Conference on World Wide Web*, pp. 657–666, May 2007.

[60] F. Schaub, R. Deyhle, and M. Weber, "Password entry usability and shoulder surfing susceptibility on different smartphone platforms," in *Proceedings of the International Conference on Mobile and Ubiquitous Multimedia*, pp. 13:1–13:10, Dec. 2012.

[61] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, pp. 612–613, Nov. 1979.

[62] M. Bellare and G. Neven, "Identity-based multi-signatures from RSA," in *Proceedings of the Cryptographers' Track at the RSA Conference*, pp. 145–162, Springer, Feb. 2007.

[63] R. K. Konoth, V. van der Veen, and H. Bos, "How anywhere computing just killed your phone-based two-factor authentication," 2016. http://fc16.ifca.ai/preproceedings/24_Konoth.pdf.

[64] C. McColgan, "Issues with SMS deprecation rationale," Sept. 2016. https://github.com/usnistgov/800-63-3/issues/351.

[65] T. F. L. Lecube, J. Miller, T. Jaeger, S. Oh, W. Zhao, and E. Min, "Multi-device authentication," 2015. US Patent Application 2017/0093846 A1.

[66] M. D. Corner and B. D. Noble, "Zero-Interaction authentication," in *Proceedings of the ACM International Conference on Mobile Computing and Networking*, pp. 1–11, Sept. 2002.

[67] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *IEEE Symposium on Security and Privacy*, pp. 553–567, May 2012.

[68] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 1181–1192, 2013.

[69] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: preventing SQL injection attacks using dynamic candidate evaluations," in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 12–24, 2007.

[70] W. G. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks," in *Proceedings of the ACM International Conference on Automated Software Engineering*, pp. 174–183, 2005.

[71] C. Nagy and A. Cleve, "A static code smell detector for SQL queries embedded in java code," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 147–152, 2017.

[72] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," in *IEEE Software*, vol. 25, pp. 22–29, 2008.

[73] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis.," in *Proceedings of the USENIX Security Symposium*, vol. 14, 2005.

[74] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proceedings of the IEEE International Conference on Software Engineering*, pp. 672–681, 2013.

[75] "The open web application security project (OWASP): Sql injection prevention cheat sheet.." https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, 2018.

[76] L. K. Shar and H. B. K. Tan, "Defending against cross-site scripting attacks," in *IEEE Computer*, vol. 45, pp. 55–62, 2012.

[77] J. Clarke-Salt, *SQL Injection Attacks and Defense*. Elsevier, 2009.

[78] S. Friedl, "SQL injection attacks by example." http://unixwiz.net/techtips/sql-injection.html, 2017.

[79] A. Douglen, "Sql smuggling or, the attack that wasn't there, Comsec Consulting Research." http://www.it-docs.net/ddata/4954.pdf.

[80] A. Ginsberg and C. Yu, "Rapid homoglyph prediction and detection," in *Proceedings of the IEEE International Conference on Data Intelligence and Security*, pp. 17–23, 2018.

[81] C. Anley, "Advanced SQL injection in SQL server applications, an NGSSoftware Insight Security Research Publication." https://crypto.stanford.edu/cs155old/cs155-spring06/sql_injection.pdf.

[82] P. Bisht, A. P. Sistla, and V. Venkatakrishnan, "Automatically preparing safe SQL queries," in *Proceedings of the International Conference on Financial Cryptography and Data Security*, pp. 272–288, 2010.

[83] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities," in *Information and Software Technology*, vol. 51, pp. 589–598, 2009.

[84] S. Thomas and L. Williams, "Using automated fix generation to secure SQL statements," in *Proceedings of the Third IEEE International Workshop on Software Engineering for Secure Systems*, p. 9, 2007.

[85] "Oracle Community: PreparedStatement and order by." https://community.oracle.com/thread/1340727.

[86] "Using prepared statements to set table name." https://stackoverflow.com/a/1208477/701325.

[87] M. Chen, S. Gonzalez, A. Vasilakos, H. Cao, and V. C. Leung, "Body area networks: A survey," *Mobile networks and applications*, vol. 16, pp. 171–193, Apr. 2011.

[88] B. Latré, B. Braem, I. Moerman, C. Blondia, and P. Demeester, "A survey on wireless body area networks," *Wireless Networks*, vol. 17, pp. 1–18, Jan. 2011.

[89] International Standards Organization, "Information technology – Trusted platform module library – Part 1: Architecture," tech. rep., Aug. 2015. ISO/IEC 11889-1:2015. https://www.iso.org/standard/66510.html.

[90] O. Staff, "Three arrested in Hillsborough County burglary," Oct. 2016. http://www.plantcityobserver.com/article/three-arrested-hillsborough-county-burglary.

[91] J. Milian, "Burglars using stolen garage-door openers in boynton beach," Aug. 2015. http://www.mypalmbeachpost.com/ZU0JluARHfHTpRnGA6A4ZJ/.

[92] B. News Staff, "Mercedes 'relay' box thieves caught on CCTV in Solihull," Nov. 2017. http://www.bbc.com/news/uk-england-birmingham-42132689.

[93] G. Belford, S. Bunch, J. Day, P. Alsberg, D. Brown, E. Grapa, D. Healy, and J. Mullen, "A state-of-the-art report on network data management and related technology," Tech. Rep. 150, Center for Advanced Computation, University of Illinois at Urbana-Champaign, Apr. 1975. Page 132. https://archive.org/details/stateoftheartrep150belf.

[94] D. Irvine, A. Zemke, G. Pusateri, L. Gerlach, R. Chun, and W. M. Jay, "Tablet and smartphone accessibility features in the low vision rehabilitation," *Neuro-ophthalmology*, vol. 38, pp. 53–59, June 2014.

[95] M. T. Islam, B. Islam, and S. Nirjon, "Soundsifter: Mitigating overhearing of continuous listening devices," in *ACM Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 29–41, June 2017.

[96] International Standards Organization, "Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification," tech. rep., Feb. 2015. ISO/IEC 18004:2015. https://www.iso.org/standard/62021.html.

[97] Department of Defense, *Nuclear Weapon Accident Response Procedures (NARP)*, Sept. 1990. DoD 5100.52-M. https://fas.org/nuke/guide/usa/doctrine/dod/5100-52m/chap15.pdf.

[98] M. Woodward, "Air force instruction 91-104," Apr. 2013. https://fas.org/irp/doddir/usaf/afi91-104.pdf.

[99] M. Reed, P. Syverson, and D. Goldschlag, "Anonymous connections and onion routing," *IEEE Journal on Selected areas in Communications*, vol. 16, no. 4, pp. 482–494, 1998.

[100] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[101] H. Corrigan-Gibbs, D. J. Wu, and D. Boneh, "Quantum Operating Systems," in *HotOS*, pp. 76–81, 2017.

[102] C. Cheng, R. Lu, A. Petzoldt, and T. Takagi, "Securing the Internet of Things in a quantum world," *IEEE Communications Magazine*, vol. 55, pp. 116–120, Feb. 2017.

[103] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "ProVerif 1.98pl1: Automatic cryptographic protocol verifier, user manual and tutorial," Dec. 2017. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf.

[104] M. Arapinis, J. Phillips, E. Ritter, and M. D. Ryan, "StatVerif: Verification of stateful processes," *Journal of Computer Security*, vol. 22, pp. 743–821, July 2014.

[105] C. Cetin and J. Ligatti, "ProVerif coauthentication files," Dec. 2018. https://github.com/Coauthentication/FormalModels/tree/journal.

[106] T. Y. Woo and S. S. Lam, "A semantic model for authentication protocols," in *Proceedings of IEEE Symposium on Research in Security and Privacy*, pp. 178–194, 1993.

[107] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis, S. Deleuze, and M. Simons, *Spring Boot Reference Guide 1.5.4.RELEASE*, June 2017. https://docs.spring.io/spring-boot/docs/1.5.4.RELEASE/reference/pdf/spring-boot-reference.pdf.

[108] M. Dell'Amico, P. Michiardi, and Y. Roudier, "Password strength: An empirical analysis," in *Proceedings of IEEE INFOCOM*, pp. 1–9, March 2010.

[109] J. Dunning, "Taming the blue beast: A survey of bluetooth-based threats," *IEEE Security & Privacy*, vol. 8, no. 2, pp. 20–27, 2010.

[110] R. Shay, S. Komanduri, A. L. Durity, P. S. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor, "Can long passwords be secure and usable?," in *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 2927–2936, Apr. 2014.

[111] R. Shay, L. Bauer, N. Christin, L. F. Cranor, A. Forget, S. Komanduri, M. L. Mazurek, W. Melicher, S. M. Segreti, and B. Ur, "A spoonful of sugar?: The impact of guidance and feedback on password-creation behavior," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pp. 2903–2912, April 2015.

[112] B. Smith, L. Williams, and A. Austin, "Idea: Using system level testing for revealing SQL injection-related error message information leaks," in *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pp. 192–200, 2010.

[113] D. Litchfield, "Web application disassembly with ODBC error messages." http://www.davidlitchfield.com/WebApplicationDisassemblywith ODBCErrorMessages.pdf, 2001.

[114] T. software BV, "Tiobe programming community index.." https://www.tiobe.com/tiobe-index/, 2018.

[115] "GitHub Archive." https://www.gharchive.org/, 2018.

[116] "Google BigQuery," 2018. https://bigquery.cloud.google.com/, Accessed: 2018-07-30.

[117] M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in Java projects," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pp. 551–555, 2015.

[118] A. Sanyal, B. Sathe, and U. Khedker, *Data flow analysis: theory and practice*. CRC Press, 2009.

[119] S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann Publishers, 1997.

[120] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the ACM 11th Working Conference on Mining Software Repositories*, pp. 92–101, 2014.

[121] "Java SE 7 PreparedStatement Interface," 2018. https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html, Accessed: 2018-07-27.

[122] "PHP Data Objects," 2019. https://secure.php.net/manual/en/book.pdo.php, Accessed: 2018-07-27.

[123] "H2 Java SQL Database JDBC Driver." https://github.com/h2database/h2database, 2018.

[124] P. W. Shor and J. Preskill, "Simple proof of security of the bb84 quantum key distribution protocol," *Physical review letters*, vol. 85, no. 2, p. 441, 2000.

[125] H.-K. Lo, X. Ma, and K. Chen, "Decoy state quantum key distribution," *Physical review letters*, vol. 94, no. 23, p. 230504, 2005.

[126] J. L. Park, "The concept of transition in quantum mechanics," *Foundations of Physics*, vol. 1, no. 1, pp. 23–33, 1970.

[127] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, no. 5886, p. 802, 1982.

[128] V. Scarani, H. Bechmann-Pasquinucci, N. J. Cerf, M. Dušek, N. Lütkenhaus, and M. Peev, "The security of practical quantum key distribution," *Reviews of modern physics*, vol. 81, no. 3, p. 1301, 2009.

[129] H. Barnum, C. Crépeau, D. Gottesman, A. Smith, and A. Tapp, "Authentication of quantum messages," in *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pp. 449–458, IEEE, 2002.

[130] M. Dušek, O. Haderka, M. Hendrych, and R. Myška, "Quantum identification system," *Physical Review A*, vol. 60, no. 1, p. 149, 1999.

[131] X. Li and H. Barnum, "Quantum authentication using entangled states," *International Journal of Foundations of Computer Science*, vol. 15, no. 04, pp. 609–617, 2004.

[132] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *Proceedings of the First international conference on Principles of Security and Trust*, pp. 3–29, Springer-Verlag, 2012.

APPENDIX A

COPYRIGHT PERMISSIONS

The following permission is for to use the content in Chapters 1–4, and 8.

ACM (Association for Computing Machinery) LICENSE TERMS AND CONDITIONS

Jun 05, 2019

This is a License Agreement between University of South Florida -- Cagri Cetin ("You") and ACM (Association for Computing Machinery) ("ACM (Association for Computing Machinery)") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by ACM (Association for Computing Machinery), and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	4602550100662
License date	Jun 04, 2019
Licensed content publisher	ACM (Association for Computing Machinery)
Licensed content title	Proceedings, Association for Computing Machinery
Licensed content date	Jan 1, 1900
Type of Use	Thesis/Dissertation
Requestor type	Author of requested content
Format	Electronic
Portion	chapter/article
The requesting person/organization is:	Cagri Cetin / University of South Florida
Title or numeric reference of the portion(s)	Coauthentication
Title of the article or chapter the portion is from	Coauthentication
Editor of portion(s)	N/A
Author of portion(s)	Cagri Cetin, Jay Ligatti, Shamaria Engram, Jean-Baptiste Subils, Dmitry Goldgof
Volume of serial or monograph.	N/A
Page range of the portion	
Publication date of portion	April 08, 2019
Rights for	Main product
Duration of use	Life of current edition
Creation of copies for the disabled	no
With minor editing privileges	yes
For distribution to	Worldwide
In the following language(s)	Original language of publication
With incidental promotional use	no
The lifetime unit quantity of new product	More than 2,000,000
Title	Authentication and SQL-Injection Prevention Techniques in Web Applications
Institution name	University of South Florida

The following permission is for to use the content in Chapters 1–2 and 5–8.

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

SQL-Identifier Injection Attacks
Mr. Cagri Cetin, Prof. Jay Ligatti and Dr. Dmitry Goldgof
2019 IEEE Conference on Communications and Network Security (CNS)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Cagri Cetin	19-03-2019
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the