Secure Coding Practices in Java: Challenges and Vulnerabilities*

Na Meng Stefan Nagy Danfeng (Daphne) Yao Wenjie Zhuang Gustavo Arango Argoty
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
{nm8247,snagy2,danfeng,kaito,gustavo1}@vt.edu

ABSTRACT

The Java platform and its third-party libraries provide useful features to facilitate secure coding. However, misusing them can cost developers time and effort, as well as introduce security vulnerabilities in software. We conducted an empirical study on StackOverflow posts, aiming to understand developers' concerns on Java secure coding, their programming obstacles, and insecure coding practices.

We observed a wide adoption of the authentication and authorization features provided by Spring Security—a third-party framework designed to secure enterprise applications. We found that programming challenges are usually related to APIs or libraries, including the complicated cross-language data handling of cryptography APIs, and the complex Java-based or XML-based approaches to configure Spring Security. In addition, we reported multiple security vulnerabilities in the suggested code of accepted answers on the StackOverflow forum. The vulnerabilities included disabling the default protection against Cross-Site Request Forgery (CSRF) attacks, breaking SSL/TLS security through bypassing certificate validation, and using insecure cryptographic hash functions. Our findings reveal the insufficiency of secure coding assistance and documentation, as well as the huge gap between security theory and coding practices.

CCS CONCEPTS

• **General and reference → Empirical studies**;

KEYWORDS

Secure coding, Spring Security, CSRF, SSL/TLS, certificate validation, cryptographic hash functions, authentication, authorization, StackOverflow, cryptography

ACM Reference Format:
Na Meng Stefan Nagy Danfeng (Daphne) Yao Wenjie Zhuang Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180201

*This work was supported by NSF Grant CCF-1565827 and ONR Grant N00014-17-1-2498.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180201

1 INTRODUCTION

The Java platform and third-party libraries (e.g., BouncyCastle [9]) provide useful features to support secure coding. Misusing these libraries and frameworks not only slows down code development, but also leads to security vulnerabilities in software [19, 74, 80, 98].

Prior research has mostly focused on the specific topics of misusing cryptography and secure socket layer (SSL) APIs [23, 26, 29, 60, 72]. For example, Nadi et al. investigated the obstacles introduced by Java cryptography APIs, developers' usage of the APIs, and desired tool support [68]. Lazar et al. manually examined 269 published cryptographic vulnerabilities in the CVE database, and found that 83% of them were resulted from cryptography API misuse [60]. Fahl et al. [26] and Georgiev et al. [29] separately reported vulnerable Android applications and software libraries that misuse SSL APIs and demonstrated how these vulnerabilities cause man-in-the-middle attacks. Rahaman and Yao recently introduced cryptographic program analysis (CPA) [72], which detects cryptographic coding errors in C/C++ programs with static taint analysis. All these studies on improving cryptography and SSL coding security are timely and important.

In this work, we conducted an in-depth investigation on Java secure coding problems. Our analysis is not limited to cryptography or SSL APIs. We inspected 503 StackOverflow (SO) posts that are related to Java security. The majority (87%) of the posts are about non-crypto libraries. For each post, we thoroughly examined the entire thread, including the question and all the responses. We chose StackOverflow [91] because (1) it is an extremely popular online platform for developers to share and discuss programming issues and solutions, and (2) SO plays an important role in educating developers and shaping their daily coding practices.

Our analysis was conducted at the code level, as code-level investigation has the potential to bring deeper insights. The technical challenge is *how to interpret the short and brief posts within the appropriate programming context in order to understand the security impact.* To comprehend each post's *program* context, we studied the context related to the source code, configuration files, and execution environments. We aimed to identify the root causes and solutions of each problem. To comprehend each post's *security* context, we inferred developers' implementation intents from their problem descriptions and the involved security libraries. We also leveraged our security expertise to assess whether the accepted solutions fulfilled their original intents. These analysis and reasoning tasks require expertise in both software engineering and cyber security.

In our analysis of the 503 StackOverflow posts, we investigated the following three research questions (RQs):

RQ1 *What are the common concerns in Java secure coding?* We aimed to identify the libraries and functionalities (e.g., [3,

32, 45, 47, 69, 85]) that were most frequently asked about by developers. Our scope covers all topics related to Java security, not limited to cryptography and SSL.

RQ2 *What are the common programming challenges?* We aimed to identify the common obstacles that hinder secure coding. Such information can provide software engineering researchers actionable insights for designing tools and help close the gap between correct API usage and the practice.

RQ3 *What are the common security vulnerabilities?* The high popularity of StackOverflow may cause insecure code to be shared and used in real-world implementations. This effort helps raise the security awareness among software developers.

Our work provides empirical evidences for many significant secure coding issues that have not been previously reported. The major findings are summarized as follows.

- *There were security vulnerabilities in the recommended code of some accepted answers.* For example, when encountering errors during implementing Spring Security authentication, developers were suggested a workaround to disable the default security protection against Cross-Site Request Forgery (CSRF) attacks. Also for example, some posts advised developers to trust all incoming SSL/TLS certificates as a fix to certificate verification errors. Such a setup completely destroys the security guarantees of SSL/TLS. Although this insecure practice was reported by security researchers in 2012 [26, 29], some SO users still view this option as acceptable. In addition, MD5 or SHA-1 algorithms was repeatedly suggested, even though these cryptographic hashing algorithms are weak and should not be used for hashing passwords. For the 17 problematic posts (5 on CSRF, 9 on SSL/TLS, and 3 on password hashing), the total view count is 622,922 [1].
- *Various programming challenges were related to security library usage.* For instance, developers became stuck with using cryptography APIs due to clueless error messages, complex cross-language data handling (e.g., encryption in Python and decryption in Java), and delicate implicit API usage constraints. When using Spring Security, developers struggled with the two alternative ways of configuring security: Java-based or XML-based.
- *Since 2012, developers have increasingly relied on the Spring Security for secure coding.* 267 of the 503 examined posts (53%) are about the Spring Security, specifically on the authentication and authorization operations in enterprise applications. However, security and usability studies about Spring Security have not been reported in the literature.

Developers have pragmatic goals (i.e., getting the code to run) and security goals. Some of the choices made by developers indicate that the pragmatic goals can take priority over security, if a developer cannot satisfy both of them. In addition, cybersecurity decisions may be influenced by the social factors (such as reputation scores, votes, and accept labels) on the StackOverflow forum. We also found one instance of cyberbullying, where condescending comments were directed at a security-conscious user [103]. We briefly report the social behavioral findings in Section 4.3.4. Our data set is available at http://people.cs.vt.edu/nm8247/icse18.xlsx.

[1] As of August 2017

2 BACKGROUND

The examined posts cover three topics on Java security: Java platform security, Java EE security, and third-party frameworks. This section introduces the key terminologies used throughout the paper.

2.1 Java Platform Security

The platform defines APIs spanning major security areas, including cryptography, access control, and secure communication [54]. The *Java Cryptography Architecture (JCA)* contains APIs for **cryptographic hashes**, **keys and certificates**, **digital signatures**, and **encryption** [47]. Nine cryptographic engines are defined to provide either cryptographic operations (encryption, digital signatures, hashes), generators or converters of cryptographic material (keys and algorithm parameters), or objects (keystores or certificates) that encapsulate the cryptographic data. The *access control architecture* protects the access to sensitive resources (e.g., local files) or sensitive application code (e.g., methods in a class). All access control decisions are mediated by a **security manager**. By default, the security manager uses the `AccessController` class for access control operations and decisions. *Secure communication* ensures that the data traveling across a network is sent to the appropriate party, without being modified during the transmission. The Java platform provides API support for standard secure communication protocols like **SSL/TLS**. HTTPS, or "HTTP secure", is an application-specific implementation that is a combination of HTTP and SSL/TLS.

2.2 Java EE Security

Java EE is a standard specification for enterprise Java extensions [59]. Various application servers are built to implement this specification, such as JBoss or WildFly [104], Glassfish [31], WebSphere [101], and WebLogic [4]. A Java EE application consists of components deployed into various containers. Containers secure components by supporting features like authentication and authorization.

Specifically, **authentication** defines how communicating entities (i.e., a client and a server), prove to each other their identities. An authenticated user is issued a *credential*, which includes information like usernames/passwords or tokens. **Authorization** ensures that users have permissions to perform operations or access data. When accessing a certain resource, a user is authorized if the server can map this user to a security role permitted for the resource. Java EE applications' security can be implemented in two ways:

- **Declarative Security** expresses an application component's security requirements using either **deployment descriptors** or **annotations**. A deployment descriptor is an XML file external to the application. This XML file expresses an application's security structure, including security roles, access control, and authentication requirements. Annotations are used to specify security information in a class file. They can be either used or overridden by deployment descriptors.
- **Programmatic Security** is embedded in an application and is used to make security decisions, when declarative security alone is not sufficient to express the security model.

2.3 Third-Party Security Frameworks

Several frameworks were built to provide authentication, authorization, and other security features for enterprise applications, such as

Spring Security (SS) [82]. Different from the Java EE security APIs, these frameworks are container independent, meaning that they do not require containers to implement security. For example, SS handles requests as a single **filter** inside a container's filter chain. There can be multiple security filters inside the SS filter. Developers can choose between **XML-based** and **Java-based** security configurations, or a hybrid of the two. Similar to Java EE security, the XML-based configuration implements security requirements with deployment descriptors and source code, while the Java-based approach expresses security with annotations and code.

3 METHODOLOGY

We used the open source python library Scrapy [77] to crawl posts from the StackOverflow (SO) website. Figure 1 presents the format of a typical SO post. Each post mainly contains two regions: the question and answers.

① **Question region** contains the question description and some metadata. The metadata includes a **vote** for the question (e.g., 3), indicating whether the question is well-defined or well-representative, and a **favorite count** (e.g., 1) showing how many people liked the question.

② **Answer region** contains all answer(s) provided. When one or more answers are provided, the asker decides which answer to **accept**, and marks it with (✓).

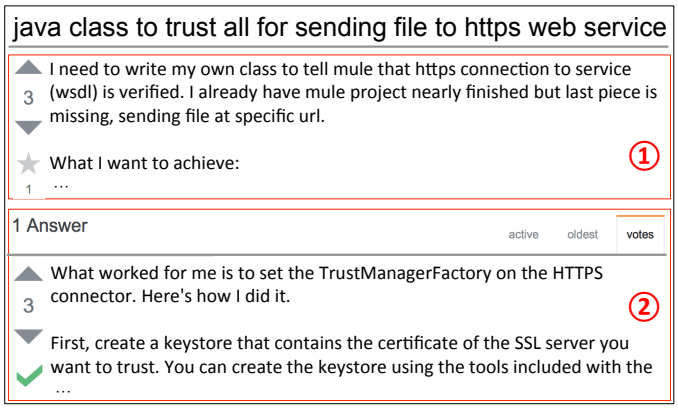

Figure 1: A highly viewed post (viewed 662 times as of January 16, 2018) asking about HTTPS workarounds to bypass key checking and allow all host names [46]

We obtained 22,195 posts containing keywords "Java" and "security". After extracting the question, answers, and relevant metadata for each post, we refined the data in three ways.

1) Filtering less useful posts. We automatically refined posts by removing duplicated posts, posts without accepted answers, and posts whose questions received negative votes (usually because the questions were ill-formed or confusing).

2) Removing posts without code snippets. To better understand the questions within the program context, we only focused on posts containing code snippets. Since our crawled data did not include any metadata describing the existence of code snippets, we developed an intuitive filter to search for keywords "public" and "class" in each post. Based on our observation, a post usually contains these two keywords when it includes a code snippet.

3) Discarding irrelevant posts. After applying the above two filters, we manually examined the remaining posts, and decided whether they were relevant to Java secure coding, or simply contained the checked keywords accidentally.

With the above three filters, we finally included 503 posts in our dataset asked between 2008-2016. We manually characterized relevant posts according to their *security concerns*, *programming challenges*, and *security vulnerabilities*. Based on this characterization, we classified the posts. We aim to answer the following three research questions (RQs):

RQ1: What are the common security concerns of developers? We aimed to investigate: (1) what are the popular security libraries or functionalities that developers frequently asked about, and (2) how have developers' security concerns shifted over the years? Because we had no prior knowledge of developers' security concerns, we adopted an open coding approach to classify posts. Specifically, Author 4 initially categorized posts based on the software libraries and security concepts discussed. Author 1 (an SE professor) then iteratively reviewed posts to create and adjust the identified security concerns. Next, Author 2 examined around 150 posts suggested by Author 1 to identify security vulnerabilities in their answers. To ensure high quality of the findings, the two authors cross checked results, and resolved disagreement with Author 3 (a cybersecurity professor).

We also classified posts into three categories based on the number of positive votes and favorite counts that their questions received:

- Neutral: A question does not have any positive vote or favorite count.
- Positive: A question receives at least one positive vote but zero favorite count.
- Favorite: A question obtains at least one favorite vote.

Thus, the post in Figure 1 is classified as "Favorite", because its favorite count is 1. By combining this categorization with the security concerns, we inferred developers' attitudes towards these coding issues. Questions that are project-specific or seemingly complicated may receive low favorite counts, as other developers may not learn or benefit from them.

RQ2: What are the common programming challenges? For each identified security concern, we further characterized each post based on its problem (buggy source code, wrongly implemented configuration files, improperly configured execution environment), the problem's root cause, and the accepted solution. We then clustered posts that have similar characteristics. For the post in Figure 1, we identified its problem as SSL verification workaround. The developer seemed unaware that SSL should not be bypassed. The recommended solution was to first create a keystore that contains the certificates of all trusted SSL servers, and then use this keystore to instantiate a `TrustManagerFactory` for establishing (unverified) connections.

RQ3: What are the common security vulnerabilities? To analyze each post's security impact, we inspected the entire thread, including unaccepted answers and conversational comments between the question asker and others. Based on recommended secure coding practices and the post's security context, we decided whether the accepted solution was vulnerable. The post shown in

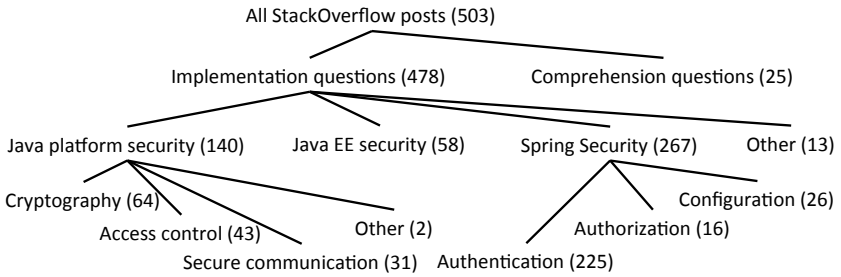

Figure 2: Taxonomy of StackOverflow posts

Figure 1 has a secure accepted answer, although the asker originally asked for a vulnerable solution as an easy fix.

4 MAJOR FINDINGS

We present our investigation results for the research questions separately in Section 4.1-4.3.

4.1 Common Concerns in Security Coding

Figure 2 presents our classification hierarchy among the 503 posts. At the highest level, we created two categories: **implementation questions** vs. **comprehension questions**. The majority (478 posts) were about implementing security functionalities or resolving program errors. Only 25 questions were asked to understand why specific features were designed in certain ways (e.g., "How does Java string being immutable increase security?" [37]). Because our focus is on secure coding practices, our further classification expands on the 478 implementation-relevant posts.

At the second level of the hierarchy, we clustered posts based on the *major security platforms or frameworks* involved in each post. Corresponding to Section 2, we identified posts relevant to **Java platform security**, **Java EE security**, **Spring Security**, and **other** third-party security libraries or frameworks.

At the third level, we classified the posts belonging to either Java platform security or Spring Security, because both categories contained many posts. Among the Java platform security posts, in addition to **cryptography** and **secure communication**, we identified a third major concern – **access control**. Among the Spring Security posts, the majority (225) are related to **authentication**, with the rest on **authorization** and **configuration**.

> **Finding 1:** *56%, 29%, and 12% of the implementation-relevant posts are on Spring Security, Java platform security, and Java EE security, respectively. This finding indicates that developers need more help with Java Spring Security.*

Based on the second- and third-level classifications, we identified seven major security topics: cryptography, access control, secure communication, Java EE security, authentication, authorization, and configuration. The first three topics correspond to Java platform security, while the last three correspond to Spring Security. To reveal trends in developers' security concerns over time, we clustered posts based on the year each question was asked.

Figure 3 presents the post distribution among 2008-2016. The total number of posts increased over the years, indicating that more

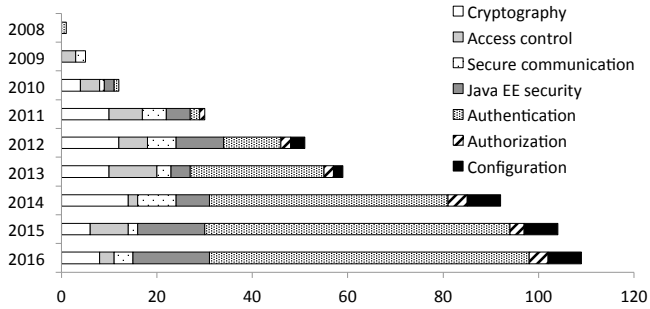

Figure 3: The post distribution during 2008-2016

developers became involved in secure coding and encountered problems. Specifically, there was only 1 post created in 2008, whereas 107 posts were created in 2016. During 2009-2011, most posts were about Java platform security. However, since 2012, the major security concern has shifted to securing Java enterprise applications (including both Java EE security and Spring Security). Specifically, Spring Security has taken up over 50% of the posts published every year since 2013.

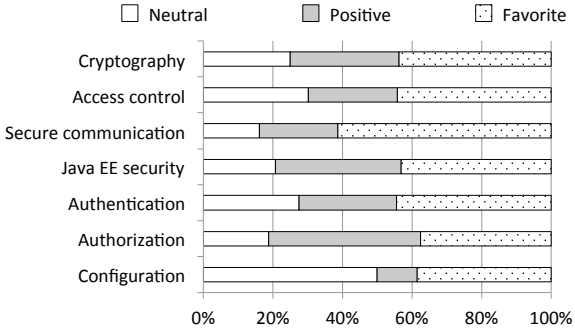

Figure 4: The post distribution among developers' attitudes: neutral, positive, and favorite

As shown in Figure 4, we also clustered posts based on developers' attitudes towards the questions for each security concern. The *configuration* posts received the highest percentage of neutral opinions (50%). One possible reason is that these posts mainly focused on problems caused by incorrect library versions and library dependency conflicts. Since such problems are usually specific to software development environments, they are not representative or relevant to many developers' security interests. In comparison, *secure communication* posts received the lowest percentage of neutral

opinions (16%), but the highest percentage of favorite (61%), indicating that the questions were more representative, focusing more on security implementation, instead of environment configuration.

> **Finding 2:** *Over time, developers' major security concern has shifted from securing Java platform to enterprise applications, especially the Spring Security framework. Secure communication posts received the highest percentage (61%) of favorite votes, indicating that these questions are both important and representative.*

4.2 Common Programming Challenges

To understand the common challenges developers faced, we examined the posts from the top five most popular categories, namely authentication (225), cryptography (64), Java EE security (58), access control (43), and secure communication (31). We identified posts with similar questions and related answers, and further investigated why developers asked these common questions. This section presents our key findings for each category.

4.2.1 Authentication. Most posts were related to (1) integrating Spring security with different application servers (e.g., JBoss) [87] or frameworks (e.g., Spring MVC) [83] (35 posts), (2) configuring security in an XML-based [84] or Java-based method [42] (145 posts), or (3) converting XML-based configurations to Java-based ones [15] (18 posts). Specifically, we observed three challenges.

Challenge 1: There is much variation in integrating Spring Security (SS) with different types of applications. Although SS can be used to secure enterprise applications no matter whether the applications are Spring-based or not, the usage varies with the application settings [86]. What's worse is that some SS-relevant implementations may exhibit different dynamic behaviors in different application contexts. As shown in Listing 1, by following a standard tutorial example [100], a developer defined two custom authentication filters—`apiAuthenticationFilter` and `webAuthenticationFilter`—to secure two sets of URLs of his/her Spring Boot web application.

Listing 1: An example of code working unexpectedly in Spring Boot applications [18]

```
1  @EnableWebSecurity
2  public class SecurityConfiguration {
3    @Configuration @Order(1)
4    public static class ApiConfigurationAdapter
5        extends WebSecurityConfigurerAdapter {
6      @Bean // define the 1st authentication filter
7      public GenericFilterBean
8          apiAuthenticationFilter() {...}
9      @Override
10     protected void configure(HttpSecurity http)
11         throws Exception {
12       http.antMatcher("/api/**") // URL pattern match
13         .addFilterAfter(apiAuthenticationFilter()...)
14         .sessionManagement()...;  } }
15   @Configuration @Order(2)
16   public static class WebSecurityConfiguration
17       extends WebSecurityConfigurerAdapter {
18     @Bean // define the 2nd authentication filter
19     public GenericFilterBean
20         webAuthenticationFilter() {...}
21   @Override
22   protected void configure(HttpSecurity http)
23       throws Exception {
24     http.antMatcher("/") // URL pattern match
25         .addFilterAfter(webAuthenticationFilter()...)
26         .authorizeRequests()...; } } }
```

In Listing 1, lines 3-14 correspond to `ApiConfigurationAdapter`, a security configuration class that specifies `apiAuthenticationFilter` to authenticate URLs matching the pattern "`/api/**`". Lines 15-26 correspond to `WebSecurityConfiguration`, which configures `webAuthenticationFilter` to authenticate the other URLs. Ideally, only one filter is invoked given one URL, however in reality, both filters were invoked. The root cause is that each filter is a bean (annotated with @Bean on lines 6 and 18). Spring Boot detects the filters and adds them to a regular filter chain, while SS also adds them to its own filter chain. Consequently, both filters are registered twice and can be invoked twice. To solve the problem, developers need to enforce each bean to be registered *only once* by adding specialized code. Unfortunately, this issue is not documented in the tutorial.

Challenge 2: The two security configurations (Java-based and XML-based) are difficult to implement correctly. Take the Java-based configuration for example. There are lots of annotations and APIs of classes, methods, and fields available to specify different configuration options. For example, `HttpSecurity` has 10 methods, each of which can be invoked on an `HttpSecurity` instance and then produces another `HttpSecurity` object. If developers are not careful about the invocation order between these methods, they may get errors [40]. As shown in Listing 1, the method `antMatcher("/api/**'')` must be invoked *before* `addFilterAfter(...)` (lines 12-13), so that the filter is only applied to URLs matching the pattern "`/api/**`". Unfortunately, such implicit constraints and subtle requirements are not well documented.

Challenge 3: Converting from XML-based to Java-based configurations is tedious and error-prone. The semantic conflicts between annotations, deployment descriptors, and code implementations are difficult to locate and resolve. Such problems become more serious when developers express security in a Java-XML hybrid form. Since Spring Security 3.2, developers can configure SS in a pure Java-based approach. There is documentation describing how to migrate from XML-based to Java-based configurations [85]. However, manually applying migration rules is still time-consuming and error-prone.

> **Finding 3:** *Spring Security authentication posts were mainly about configuring security for various enterprise applications in different approaches (namely, Java-based or XML-based), and converting between them. The challenges were due to incomplete documentation, as well as missing tool support for automatic configuration checking and converting.*

4.2.2 Cryptography. 45 of the 64 posts were about key generation and usage. For instance, some posts discussed how to create a key from scratch [55], and how to generate or retrieve a key from a random number [41], a byte array [17], a string [48], a certificate [30], BigIntegers [7], a keystore [6], or a file [97]. Other

posts are on how to compare keys [14], print key information [96], or initialize a cipher for encryption and decryption [52]. Specifically, we observed three common challenges of correctly using the cryptography APIs.

Challenge 1: The error messages did not provide sufficient useful hints about fixes. We found five posts on the same problem: "get InvalidKeyException: Illegal key size", while the solutions were almost identical: (1) download the "Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files", "local_policy.jar", and "US_export_policy.jar"; and (2) place the policy files in proper folders [2]. Developers got the same exception because of missing either of the two steps. Providing a checklist of these necessary steps in the error message would help developers quickly resolve the problem. However, the existing error messages did not provide any constructive suggestion.

Challenge 2: It is difficult to implement security with multiple programming languages. Three posts were about encrypting data with one language (e.g. PHP or Python) and decrypting data with another language (e.g., Java). Such cross-language data encryption & decryption is challenging, because the format of the generated data by one language requires special handling in another language. Listing 2 is an example to generate an RSA key pair and encrypt data in PHP, and to decrypt data in Java [24].

Listing 2: Encryption in PHP and decryption in Java [24]

```
1  // *****keypair.php *****
2  if(file_exists('private.key')) {
3      echo file_get_contents('private.key'); }
4  else {
5      include('Crypt/RSA.php');
6      $rsa = new Crypt_RSA();
7      $res = $rsa->createKey();
8      $privateKey = $res['privatekey'];
9      $publicKey  = $res['publickey'];
10     file_put_contents('public.key',$publicKey);
11     file_put_contents('private.key',$privateKey); }
12 // *****encrypt.php *****
13 include('Crypt/RSA.php');
14 $rsa = new Crypt_RSA();
15 $rsa->setEncryptionMode(CRYPT_RSA_ENCRYPTION_OAEP);
16 $rsa->loadKey(file_get_contents('public.key'));
17 // *****MainClass.java *****
18 BASE64Decoder decoder=new BASE64Decoder();
19 String b64PrivateKey=getContents(
20   "http://localhost/api/keypair.php").trim();
21 byte[] decodedKey=decoder.decodeBuffer(b64PrivateKey);
22 BufferedReader br=new BufferedReader(
23   new StringReader(new String(decodedKey)));
24 PEMReader pr=new PEMReader(br);
25 KeyPair kp=(KeyPair)pr.readObject();
26 pr.close();
27 PrivateKey privateKey=kp.getPrivate();
28 Cipher cipher=Cipher.getInstance(
29   "RSA/None/OAEPWithSHA1AndMGF1Padding","BC");
30 cipher.init(Cipher.DECRYPT_MODE, privateKey);
31 byte[] plaintext=cipher.doFinal(cipher);
```

In this example, when a key pair is generated in PHP (lines 2-11), the public key is easy to retrieve in PHP (lines 13-16). However, retrieving the private key in Java is more complicated (lines 18-30). After reading in the private key string (lines 19-20), the Java implementation first uses `Base64Decoder` to decode the string into a byte array (line 21), which corresponds to an OpenSSL PEM encoded stream (line 22-23). Because OpenSSL PEM is not a standard data format, the Java code further uses a PEMReader to convert the stream to a PrivateKey instance (lines 24-27) before using the key to initialize a cipher (lines 28-30). Existing documentation seldom describes how the security data format (e.g., key) defined in one language corresponds to that of another language. Unless developers are experts in both languages, it is challenging for them to figure out the security data processing across languages.

Challenge 3: Implicit constraints on API usage cause confusion. Two posts were about getting "InvalidKeySpecException: algid parse error, not a sequence", when obtaining a private key from a file [44]. The problem is that the key should be in PKCS#8 format when used to create a `PKCS8EncodedKeySpec` instance, as shown below:

Listing 3: Consistency between the key format and spec [44]

```
1 //privKey should be in PKCS#8 format
2 byte[] privKey=...;
3 PKCS8EncodedKeySpec keySpec=
4   new PKCS8EncodedKeySpec(privKey);
```

The tricky part is that a private key retrieved from a file always has the data type `byte[]` even if it is not in PKCS#8 format. If developers invoke the API `PKCS8EncodedKeySpec(...)` with a non-PKCS#8 formatted key, they would be stuck with the clueless exception. Three solutions were suggested to get a PKCS#8 format key: (1) to implement code to convert the byte array, (2) to use an OpenSSL command to convert the file format, or (3) to use the `PEMReader` class of BouncyCastle to generate a key from the file. Such implicit constraints between an API and its input format are delicate.

> **Finding 4:** *The cryptography posts were mostly about key generation and usage. Developers asked these questions mainly due to clueless error messages, cross-language data handling, and implicit API usage constraints.*

4.2.3 Java EE security. 33 of the 58 posts were on authentication and authorization. The APIs of these two security features were defined differently on different application servers (e.g., WildFly and Glassfish). Developers might use these servers in combination with diverse third-party libraries [75]. As a result, these posts rarely shared solutions or code implementation.

One common challenge we identified is the usage of declarative security and programmatic security. When developers misunderstood annotations, they could use incorrect annotations that conflict with other annotations [49], deployment descriptors [105], code implementation [16], or file paths [71]. Nevertheless, existing error reporting systems only throw exceptions. There is no tool helping developers identify or resolve conflicting configurations.

> **Finding 5:** *Java EE security posts were mainly about authentication and authorization. One challenge is the complex usage of declarative security and programmatic security, and the complicated interactions between the two.*

4.2.4 Access Control. 43 posts mainly discussed how to *restrict* or *relax* the access permission(s) of a software application for certain resource(s).

Specifically, 21 questions asked about restricting untrusted code from accessing certain packages [53], classes [56], or class members (i.e., methods and fields) [35]. Two alternative solutions were commonly suggested for these questions: (1) to override the `checkXXX()` methods of `SecurityManager` to disallow invalid accesses, or (2) to define a custom policy file to grant limited permissions. Another nine posts were on how to allow applets to perform privileged operations [79]. Applets are executed in a security sandbox by default and can only perform a set of safe operations. One commonly recommended solution was to digitally sign the applet. Although it seems that there exist common solutions to the most frequently asked questions, the access control implementation is not always intuitive. We identified two common challenges associated with correctly implementing access control.

Challenge 1: The effect of access control varies with the program context. We identified two issues that were frequently asked about. First, the RMI tutorial [43] suggested that a security manager is needed *only* when RMI downloads code from a remote machine. For the RMI program that does not download any code, including a `SecurityManager` instance causes an AccessControlException [51]. Second, although a signed applet is allowed to perform sensitive operations, it loses its privileges when being invoked from Javascript [36]. As a result, the invocation to the signed applet should be wrapped with an invocation of `AccessController.doPrivileged(...)`.

Challenge 2: The effect of access control varies with the execution environment. `SecurityManager` can disallow illegal accesses via reflection only when the program is executed in a controlled environment (i.e., on a trusted server) [10]. Nevertheless, if the program is executed in an uncontrolled environment (e.g. on an untrusted client machine), where hackers can control how to run the program or manipulate the jar file, the security mechanisms become voided.

> **Finding 6:** *The access control posts were mainly about* `SecurityManager`, `AccessController`, *and the policy file. Configuring and customizing access control policies are challenging.*

4.2.5 Secure Communication. Among the 31 examined posts, 22 posts were about SSL/TLS-related issues, discussing how to create [88], install [94], find [58], or validate an SSL certificate [90], how to establish a secure connection [50], and how to use SSL together with other libraries, such as JNDI [38] and PowerMock [102].

In particular, six posts focused on the problem of unable to find a valid server certificate to establish an SSL connection with a server [58]. Instead of advising to install the required certificates, two accepted answers suggested a highly insecure workaround to disable the SSL verification process, so that *any* incoming certificate can pass the validation [89]. Although such workarounds effectively remove the error, they fail to secure the communications. In Section 4.3, we further explain the security vulnerability due to such workarounds. Developers likely accepted the vulnerable answers because they found it challenging to implement the entire process of creating, installing, finding, and validating an SSL certificate.

> **Finding 7:** *Security communication posts mainly discussed the process of establishing SSL/TLS connections. This process contains so many steps that developers were tempted to accept a broken solution to simply bypass the security verification.*

4.3 Common Security Vulnerabilities

Among the five categories listed in Section 4.2, we identified security vulnerabilities in the accepted answers of three frequently discussed topics: Spring Security's `csrf()`, SSL/TLS, and password hashing.

4.3.1 Spring Security's `csrf()`. **Cross-site request forgery (CSRF)** is a serious attack that tricks a web browser into executing privileged actions (e.g., transferring victim's money to attacker's account) in a web application (e.g., a bank website), without the victim's awareness [107]. The root cause is that the browser does not attempt to distinguish the attacker's forged requests from legitimate ones. It automatically appends the victim's credential (e.g., session ID stored in a cookie) to all these outgoing requests. Thus, forged requests can pass the authentication.

By default, Spring Security provides the CSRF protection by defining a function `csrf()` and implicitly enabling the function invocation. Correspondingly, developers should include the CSRF token in all PATCH, POST, PUT, and DELETE methods to leverage the protection [57]. However, among the 12 examined posts on `csrf()`, 5 posts discussed program failures, while all the accepted answers suggested an insecure solution: disabling the CSRF protection by invoking `http.csrf().disable()`. In one instance, after accepting the vulnerable solution, an asker commented "*Adding csrf().disable() solved the issue!!! I have no idea why it was enabled by default*" [62]. Unfortunately, the developer happily disabled the security protection without realizing that such workaround would expose the resulting software to exploits.

> **Finding 8:** *In 5 of the 12* `csrf()`*-relevant posts, developers took the suggestion to irresponsibly disable the default CSRF protection. Developers were unaware of the threats associated with disabling CSRF tokens.*

4.3.2 SSL/TLS. We examined the 10 posts on SSL/TLS, and observed two important security issues.

Problem 1: Many developers opted to trust all SSL certificates and permit all hostnames with the intent of quickly building a prototype in the development environment. SSL is the standard security technology for establishing an encrypted connection between a web server and browser. There are mainly four steps involved to securely enable SSL connections [76]. First, a web service's developers request for an SSL certificate for their website by providing the website's identity information (e.g., its public key and host name) to a Certification Authority (CA). Second, the CA validates the website's information, and issues a digitally signed SSL certificate. Third, when a client or browser attempts to connect the website, the server sends over its certificate. Fourth, the client conducts several checks, including (1) whether the certificate is issued by a CA the browser trusts, (2) whether the requested hostname matches the

hostname associated with the certificate, and (3) whether the server has the knowledge of the private key corresponding to the certified public key. If all these checks are passed, the SSL connection can be established successfully.

The safest practice is to enable SSL after obtaining a signed certificate from a certificate authority (CA). However, many developers implement and test certificate verification code before obtaining the certificate. A common workaround without CA-signed certificates is to create a local *self-signed* certificate for use in implementing certificate verification [88]. However, 9 of the 10 examined posts accepted an insecure solution to bypass security checks entirely by trusting *all* certificates and/or allowing *all* hostnames, as demonstrated by Listing 4.

Listing 4: A typical implementation to disable SSL certificate validation [78]

```
// Create a trust manager that does not validate certificate chains
TrustManager[] trustAllCerts = new TrustManager[]{
  new X509TrustManager() {
    public java.security.cert.X509Certificate[]
        getAcceptedIssuers() {return null;}
    public void checkClientTrusted(...) {}
    public void checkServerTrusted(...) {} }};
// Install the all-trusting trust manager
try {
  SSLContext sc = SSLContext.getInstance("SSL");
  sc.init(null, trustAllCerts,
      new java.security.SecureRandom());
  HttpsURLConnection.setDefaultSSLSocketFactory(
      sc.getSocketFactory());
} catch (Exception e) {}
// Access an https URL without any certificate
try {
  URL url=new URL("https://hostname/index.html");
} catch (MalformedURLException e) {}
```

Disabling the SSL certificate validation process completely destroys the secure communication protocol, leaving clients susceptible to **man-in-the-middle (MITM) attacks** [29]. In the MITM attack, by secretly relaying and possibly altering communication (e.g., through DNS poisoning) between client and server, an attacker can fool the SSL-client to connect to an attacker-controlled server [29]. Although the insecurity of this coding practice was highlighted in 2012 [29], three examined posts that were created since then still discussed this dangerous workaround [13, 46, 89]. This observation indicates a significant gap between security theory and coding practices. A developer justified the verification-bypassing choice by stating "*I want my client to accept any certificate (because I'm only ever pointing to one server)*" [95]. [2] This statement indicates the lack of understanding about the man-in-the-middle attack. Another developer stated "*Because I needed a quick solution for debugging purposes only. I would not use this in production due to the security concerns ...*" [95]. However, as pointed by another SO user [95] and demonstrated by prior research [26, 29], many of these implementations find their way into production software, and have yielded radically insecure systems as a result.

[2] That is, in this developer's application, a client only needs to communicate to one server.

Problem 2: Developers were unaware of the best usage of SSL/TLS. TLS is SSL's successor. TLS is so different from SSL that the two protocols do not interoperate. To maintain the backward compatibility with SSL 3.0, most SSL/TLS implementations allow protocol version negotiation: if a client and a server cannot connect via TLS, they will fall back to using the older protocol SSL 3.0. In 2014, Möler et al. reported the POODLE attack which exploits the SSL 3.0 fallback [67]. Specifically, there is a design vulnerability in the way SSL 3.0 handles block cipher mode padding, which can be exploited by attackers to decrypt ciphertext. With the POODLE attack, a hacker can intentionally trigger a TLS connection failure and force to use SSL 3.0.

Since 2014, researchers have recommended developers to disable SSL 3.0 support and configure systems to prevent the SSL 3.0 fallback [67]. The US government (NIST) mandates ceasing SSL usage in the protection of Federal information [33]. None of the 10 posts mentioned this security issue. The most recent post [89] (created in 2016) still discussed about the use of the obsolete SSL.

> **Finding 9:** *9 of 10 SSL/TLS-relevant posts discussed insecure code to bypass security checks. We observed two important security threats: (1) StackOverflow contains a lot of obsolete and insecure coding practices; and (2) developers are unaware of the state-of-the-art security knowledge.*

4.3.3 Password Hashing. We found 6 posts on hashing passwords with MD5 or SHA-1 to store user credentials in databases. However, these cryptographic hashing functions were found insecure [93, 99]. They are vulnerable to offline **dictionary attacks** [22] – after obtaining a password hash H from a compromised database, a hacker can use brute-force methods to enumerate a list of password guesses, until finding the password P whose hash value matches H. Impersonating a valid user at login allows an attacker to conduct malicious behavior. Researchers recommended **key-stretching algorithms** (e.g., PBKDF2, bcrypt, and scrypt) as the best practice for secure password hashing, as these algorithms are specially crafted to slow down hash computation by orders of magnitude [8, 28, 92], which substantially increases the difficulty of dictionary attacks.

Unfortunately, only 3 of the 6 posts (50%) mentioned the best practice in their accepted answers. One post asked about using MD5 hashing in Android [64]. Although subsequent discussion between developers revealed recommendations of avoiding MD5, the asker kept justifying his/her choice of using MD5. The asker even shared a completely wrong understanding of secure hashing: "*The security of hash algorithms really is MD5 (strongest) > SHA-1 > SHA-256 > SHA-512 (weakest)*", although the opposite is true, which is MD5 < SHA-1 < SHA-256 < SHA-512. Among these posts, some developers misunderstood security APIs and ignored the potential consequences of their API choices. Such posts conveying incorrect information on such a popular platform can have a profound negative impact on software security.

Finding 10: *3 of 6 hashing-relevant posts accepted vulnerable solutions as correct answers, indicating that developers were unaware of best secure programming practices. Incorrect security information may propagate among StackOverflow users and negatively influence software development.*

4.3.4 Social Aspects of StackOverflow. Among the 17 SO posts that either discussed or recommended insecure coding practices relevant to CSRF (5 posts described in Section 4.3.1), SSL/TLS (9 posts described in Section 4.3.2), or password hashing (3 posts described in Section 4.3.3), we observed a few interesting facts.

The total view count of these posts is 622,922 [3]. Such a large viewcount means that many developers have read these posts. It is conceivable that some developers may have heeded the erroneous advice and incorporated the vulnerable code in their projects.

Influential answers are not necessarily secure. In one post [39], the insecure suggestion by a user with a higher reputation (55.6K reputation score) was selected as the accepted answer, as opposed to the correct fix by a user with a lower reputation (29K reputation score). In another post [78], one insecure "quick fix" answer received 5 votes, probably because it indeed eliminated the error messages. The positive indicators for insecure solutions (e.g., high reputation and positive votes on StackOverflow) can mislead developers to implement insecure practices.

Also for example, a user with zero reputation score pointed out that trusting all certificates is very dangerous. Another user with a higher reputation score (6.3K) made condescending and discouraging remarks, such as "*Once you have sufficient reputation, you will be able to comment*" [103].

Finding 11: *Highly viewed posts may inadvertently promote insecure coding practices. This problem may be further aggravated by misleading indicators such as accepted answers, answers' positive votes, and responders' high reputation.*

5 RELATED WORK

We describe three categories of related work on analyzing, detecting, and preventing security vulnerabilities due to library API misuse.

5.1 Analyzing Security Vulnerabilities

Prior studies showed API misuse caused many security vulnerabilities [60, 63, 98, 106]. For instance, Long identified several Java features (e.g., the reflection API) whose misuse or improper implementation can compromise security [63]. Lazar et al. manually examined 269 published cryptographic vulnerabilities in the CVE database, and observed 83% of them were caused by the misuse of cryptographic libraries [60]. Veracode reported that 39% of all applications used broken or risky cryptographic algorithms [98].

Barua et al. automatically extracted latent topics in SO posts [5]. These topics are not specific to security. Nadi et al. reported the obstacles of using cryptography APIs by examining 100 SO posts and 48 developers' survey inputs [68]. Acar et al. focused on the vulnerabilities in Android code [1]. The studies by Yang et al. [106] and Rahman [73] are the most relevant to our research. They automatically extracted security-relevant topics from SO questions, and identified high-frequency keywords like "Password" and "Hash" for post categorization.

Our work belongs to this category of analyzing security vulnerabilities. Compared with the prior research, our selection of posts covers Java security, not limited to cryptography, SSL, or Android. This broad coverage enables us to obtain new insights on secure coding practices, including complex security configurations in Java Spring Security and cross-language data handling (e.g., encryption in Python and decryption in Java).

5.2 Detecting Security Vulnerabilities

Researchers have proposed tools to detect security vulnerabilities caused by API misuse [12, 23, 26, 27, 29, 34, 61, 70, 72]. For instance, Egele et al. implemented a static checker for six well-defined Android cryptographic API usage rules (e.g., "Do not use ECB mode for encryption"). They analyzed 11,748 Android applications for any rule violation [23]. They found 88% of the applications violated at least one checked rule. Fischer et al. extracted Android security-related code snippets from SO, and manually labeled a subset of the data as "secure" or "insecure" [27]. The labeled data is used to train a classifier that determines whether or not a code snippet is secure. The authors then searched for code clones of the snippets in 1.3 million Android apps, and found many clones of the insecure code. In 2012, Fahl et al. [26] and Georgiev et al. [29] separately reported vulnerable Android applications and software libraries that misuse SSL APIs and demonstrated how these vulnerabilities cause man-in-the-middle attacks. We found three posts created after 2012 that still discussed the highly insecure practice of trusting all certificates (in Section 4.3.2). He et al. developed SSLINT, an automatic static analysis tool, to identify the misuse of SSL/TLS APIs in client-side applications [34]. Rahaman and Yao presented a static taint analysis approach to enforce a wide range of cryptographic properties in C/C++ code [72].

5.3 Preventing Security Vulnerabilities

Researchers proposed approaches to prevent developers from implementing vulnerable code and misusing APIs [20, 21, 25, 65, 66, 81]. For example, Mettler et al. designed Joe-E – a security-oriented subset of Java – to support secure coding by removing any encapsulation-breaking features from Java (e.g., reflection), and by enforcing the least privilege principle [65]. Keyczar is a library designed to simplify the cryptography usage, and thus to prevent API misuse [21]. Below shows how to decrypt data with Keyczar:

Listing 5: Simple decryption with Keyczar APIs

```
1 Crypter crypter=new Crypter("/rsakeys");
2 String plaintext=crypter.decrypt(ciphertext);
```

Compared with the decryption code shown in Listing 2 (lines 18-31), this implementation is much simpler and more intuitive. All details about data format conversion and cipher initialization are hidden, while a default strong block cipher is used to properly decrypt data.

[3] As of August 2017

Formal verification techniques can analyze the security properties of cryptographic protocol specifications [20, 66] and cryptographic API implementations [25, 81]. For instance, Protocol Composition Logic (PCL) is a logic for proving security properties, e.g., on network protocols that use public and symmetric key cryptography [20]. The logic is designed around a process calculus with actions for possible protocol steps, including generating new random numbers and sending and receiving messages. The proof system consists of axioms about individual protocol actions and inference rules that yield assertions about protocols composed of multiple steps.

6 OUR RECOMMENDATIONS

Our work reveals the gap between the intended use and the actual use of Java security APIs. This gap may result in serious software vulnerabilities. In addition, it also impacts the productivity. Some developers reported spending substantial effort on learning about the correct API usage (e.g., two weeks as mentioned in [83]). These findings lead us to give the following recommendations.

For Developers. Conduct security testing to check whether the implemented features work as expected. Do not disable security checks (e.g., CSRF check) to implement a temporary fix in the testing or development environment. Be cautious when following SO's accepted or reputable answers to implement secure code, because some of these solutions may be insecure and outdated. For SO administrators, they may consider adding warnings to the posts with known vulnerable code, as these posts may mislead developers.

For Library Designers. Deprecate the APIs whose security guarantees are broken (e.g., MD5). Design clean and helpful error reporting interfaces which show not only the error, but also possible root causes and solutions. Design simplified APIs with strong security defenses implemented by default.

For Tool Builders. Develop automatic tools to diagnose security errors, locate buggy code, and suggest security patches or solutions. Build vulnerability prevention techniques, which compare peer applications that use the same set of APIs to infer and warn potential misuses. Explore approaches that check and enforce the semantic consistency between security-relevant annotations, code, and configurations. Build new approaches to transform between the implementations of declarative security and programmatic security.

7 THREATS TO VALIDITY

This study is based on our manual inspection of Java security posts, so the observations may be subject to human bias. To alleviate the problem, the first author of the paper conducted multiple rounds of careful inspection of all the posts relevant to implementation questions, and the second author examined the posts related to security vulnerabilities (mentioned in Section 4.3) multiple times.

To remove the posts without any code snippets, we defined a filter to search for keywords "public" and "class". If a post does not contain both words, the filter automatically removes the post from our dataset. This filter may incorrectly remove some relevant posts that contain code. One may improve the crawling technique to keep the <code> tags around code snippets in the raw data, and then use these tags to filter posts. One can also leverage Cerulo et al.'s approach [11] to automatically extract source code from text.

We chose to report the posts whose accepted answers *will* cause security vulnerabilities. There exist other posts whose accepted answers could potentially be insecure and might lead to vulnerabilities. However, due to the limited program and environment information in these posts, it is difficult for us to confirm the vulnerabilities. Therefore, we decided not to report them.

8 CONCLUSION

Our work aimed at assessing the current secure coding practices, and identifying the potential gaps between security theory and practice, and between specification and implementation. Our analysis of hundreds of posts on the popular developer forum (StackOverflow) revealed a worrisome reality in the software development industry.

- A substantial number of developers do not appear to understand the security implications of coding options, showing a lack of cybersecurity training. This situation creates frustration in developers, who sometimes end up choosing insecure-but-easy fixes. Examples of such easy fixes include *i)* disabling CSRF protection, *ii)* trusting all certificates to enable SSL/TLS, *iii)* using obsolete cryptographic hash functions, or *iv)* using obsolete communication protocols. These insecure coding practices, if used in production code, will seriously compromise the security of software products.
- We provided empirical evidence showing that (1) Spring Security usage is overly complicated and poorly documented; (2) the error reporting systems of Java platform security APIs cause confusion; and (3) the multi-language support for securing data is rather weak. These issues seriously hinder developers' productivity, resulting in frustration and confusion.
- Interestingly, we found that the social dynamics among askers and responders may impact people's security choices. Highly viewed posts may wrongly promote vulnerable code. Metadata like accepted answers, responders' reputation scores, and answers' positive vote counts can further mislead developers to take insecure advices. We also found an instance where cyberbullying comments were directed at a person who pointed out the danger of trusting all certificates.
- Developers' security concerns have shifted from cryptography APIs to Spring Security over time. However, researchers have not provided solutions to resolve the programming challenges in this new framework.

We described several possible solutions to improve secure coding practices in the paper. Efforts (e.g., workforce retraining) to correct these alarming security issues may take a while to take effect. Our future work is on building automatic or semi-automatic security bug detection and repair tools.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments.

REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You get where you're looking for: The impact of information sources on code security.

In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, May 2016.
[2] AES-256 implementation in GAE. https://stackoverflow.com/questions/12833826/aes-256-implementation-in-gae.
[3] Apache Shiro documentation. https://shiro.apache.org/documentation.html.
[4] Application Server - Oracle WebLogic Server. https://www.oracle.com/middleware/weblogic/index.html.
[5] A. Barua, S. W. Thomas, and A. E. Hassan. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654, Jun 2014.
[6] Basic Program for encrypt/Decrypt : javax.crypto.BadPaddingException: Decryption error. https://stackoverflow.com/questions/39518979/basic-program-for-encrypt-decrypt-javax-crypto-badpaddingexception-decryption.
[7] BigInteger to Key. https://stackoverflow.com/questions/10271164/biginteger-to-key.
[8] S. Boonkrong. Security of passwords. *Information Technology Journal*, 8(2):112–117, 2012.
[9] Bouncy castle. https://www.bouncycastle.org.
[10] Can a secret be hidden in a 'safe' Java class offering access credentials? https://stackoverflow.com/questions/5761519/can-a-secret-be-hidden-in-a-safe-java-class-offering-access-credentials.
[11] L. Cerulo, M. D. Penta, A. Bacchelli, M. Ceccarelli, and G. Canfora. Irish: A hidden Markov model to detect coded information islands in free text. *Science of Computer Programming*, 105(Supplement C):26 – 43, 2015.
[12] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis. Evaluation of cryptography usage in Android applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies*, pages 83–90, 2015.
[13] Communication with server that support SSL in Java. https://stackoverflow.com/questions/21156929/java-class-to-trust-all-for-sending-file-to-https-web-service.
[14] Compare two Public Key values in Java (duplicate). https://stackoverflow.com/questions/37439695/compare-two-public-key-values-in-java.
[15] Configure Spring Security without XML in Spring 4. https://stackoverflow.com/questions/20961600/configure-spring-security-without-xml-in-spring-4.
[16] @Context injection in Stateless EJB used by JAX-RS. https://stackoverflow.com/questions/29132547/context-injection-in-stateless-ejb-used-by-jax-rs.
[17] Converted secret key into bytes, how to convert it back to secret key? https://stackoverflow.com/questions/5364338/converted-secret-key-into-bytes-how-to-convert-it-back-to-secrect-key.
[18] Custom Authentication Filters in multiple HttpSecurity objects using Java Config. https://stackoverflow.com/questions/37304211/custom-authentication-filters-in-multiple-httpsecurity-objects-using-java-config.
[19] CWE-227: Improper fulfillment of API contract (API abuse). https://cwe.mitre.org/data/definitions/227.html.
[20] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311 – 358, 2007.
[21] A. Dey and S. Weis. *Keyczar: A Cryptographic Toolkit.*
[22] Dictionary Attacks 101. https://blog.codinghorror.com/dictionary-attacks-101/.
[23] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, pages 73–84, New York, NY, USA, 2013. ACM.
[24] Encryption PHP, Decryption Java. https://stackoverflow.com/questions/15639442/encryption-php-decryption-java.
[25] L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 73–82, New York, NY, USA, 2008. ACM.
[26] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS, pages 50–61, New York, NY, USA, 2012. ACM.
[27] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack Overflow considered harmful? The impact of copy&paste on Android application security. In *38th IEEE Symposium on Security and Privacy*, 2017.
[28] C. Gackenheimer. Implementing security and cryptography. In *Node. js Recipes*, pages 133–160. Springer, 2013.
[29] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, pages 38–49, New York, NY, USA, 2012. ACM.
[30] Get public and private key from ASN1 encrypted pem certificate. https://stackoverflow.com/questions/30392114/get-public-and-private-key-from-asn1-encrypted-pem-certificate.
[31] GlassFish. https://javaee.github.io/glassfish/.
[32] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation.* Pearson Education, 2nd edition, 2003.
[33] Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf.
[34] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534, May 2015.
[35] Hiding my security key from Java reflection. https://stackoverflow.com/questions/14903318/hiding-my-security-key-from-java-reflection.
[36] How can I get a signed Java Applet to perform privileged operations when called from unsigned Javascript? https://stackoverflow.com/questions/1006674/how-can-i-get-a-signed-java-applet-to-perform-privileged-operations-when-called.
[37] How does Java string being immutable increase security? https://stackoverflow.com/questions/15274874/how-does-java-string-being-immutable-increase-security.
[38] How to accept self-signed certificates for JNDI/LDAP connections? https://stackoverflow.com/questions/4615163/how-to-accept-self-signed-certificates-for-jndi-ldap-connections.
[39] How to add MD5 or SHA hash to Spring security? https://stackoverflow.com/questions/18581463/how-to-add-md5-or-sha-hash-to-spring-security.
[40] How to apply spring security filter only on secured endpoints? https://stackoverflow.com/questions/36795894/how-to-apply-spring-security-filter-only-on-secured-endpoints.
[41] How to generate secret key using SecureRandom.getInstanceStrong()? https://stackoverflow.com/questions/37244064/how-to-generate-secret-key-using-securerandom-getinstancestrong.
[42] How to override Spring Security default configuration in Spring Boot. https://stackoverflow.com/questions/35600488/how-to-override-spring-security-default-configuration-in-spring-boot.
[43] Implementing a Remote Interface. http://docs.oracle.com/javase/tutorial/rmi/implementing.html.
[44] InvalidKeySpecException : algid parse error, not a sequence. https://stackoverflow.com/questions/31941413/invalidkeyspecexception-algid-parse-error-not-a-sequence.
[45] Java authentication and authorization service (JAAS) reference guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html.
[46] Java class to trust all for sending file to HTTPS web service. https://stackoverflow.com/questions/21156929/java-class-to-trust-all-for-sending-file-to-https-web-service.
[47] Java cryptography architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.
[48] Java - Edit code sample to specify DES key value. https://stackoverflow.com/questions/22858497/edit-code-sample-to-specify-des-key-value.
[49] Java EE 7 EJB Security not working. https://stackoverflow.com/questions/30504131/java-ee-7-ejb-security-not-working.
[50] Java Mail get mails with pop3 from exchange server, Exception in thread "main" javax.mail.MessagingException. https://stackoverflow.com/questions/25017050/java-mail-get-mails-with-pop3-from-exchange-server-exception-in-thread-main.
[51] Java RMI / access denied. https://stackoverflow.com/questions/36570012/java-rmi-access-denied.
[52] Java security init Cipher from SecretKeySpec properly. https://stackoverflow.com/questions/14230096/java-security-init-cipher-from-secretkeyspec-properly.
[53] Java Security Manager completely disable reflection. https://stackoverflow.com/questions/40218973/java-security-manager-completely-disable-reflection.
[54] Java security overview. http://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html.
[55] Java Security - RSA Public Key & Private Key Code Issue. https://stackoverflow.com/questions/18757114/java-security-rsa-public-key-private-key-code-issue.
[56] Java security: Sandboxing plugins loaded via URLClassLoader. https://stackoverflow.com/questions/3947558/java-security-sandboxing-plugins-loaded-via-urlclassloader.
[57] Java - Simple example of Spring Security with Thymeleaf. https://stackoverflow.com/questions/25692735/simple-example-of-spring-security-with-thymeleaf.
[58] Java SSL - InstallCert recognizes certificate, but still "unable to find valid certification path" error? https://stackoverflow.com/questions/11087121/java-ssl-installcert-recognizes-certificate-but-still-unable-to-find-valid-c.
[59] JSR-000366 Java platform, enterprise edition 8 public review specification. http://download.oracle.com/otndocs/jcp/java_ee-8-pr-spec/.
[60] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail? A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 7:1–7:7, New York, NY, USA, 2014. ACM.
[61] Y. Li, Y. Zhang, J. Li, and D. Gu. iCryptoTracer: Dynamic analysis on misuse of cryptography functions in iOS applications. In M. H. Au, B. Carminati, and C.-C. J. Kuo, editors, *Proceedings of the 8th International Conference on Network and System Security*, pages 349–362, 2014.
[62] Logout call - Spring security logout call. https://stackoverflow.com/questions/24530603/spring-security-logout-call.

[63] F. Long. Software vulnerabilities in Java. Technical Report CMU/SEI-2005-TN-044, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.

[64] MD5 hashing in Android. https://stackoverflow.com/questions/4846484/md5-hashing-in-android.

[65] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium*. Internet Society, 2010.

[66] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Mur/spl phi/. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 141–, Washington, DC, USA, 1997. IEEE Computer Society.

[67] B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback, 2014.

[68] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE, pages 935–946, New York, NY, USA, 2016. ACM.

[69] S. Oaks. *Java Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[70] L. Onwuzurike and E. De Cristofaro. Danger is my middle name: Experimenting with SSL vulnerabilities in Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, pages 15:1–15:6, New York, NY, USA, 2015. ACM.

[71] PicketLink / Deltaspike security does not work in SOAP (JAX-WS) layer (CDI vs EJB?). https://stackoverflow.com/questions/32392702/picketlink-deltaspike-security-does-not-work-in-soap-jax-ws-layer-cdi-vs-ej.

[72] S. Rahaman and D. Yao. Program analysis of cryptographic implementations for security. In *IEEE Security Development Conference (SecDev)*, pages 61–68, 2017.

[73] M. S. Rahman. An empirical case study on Stack Overflow to explore developers' security challenges. Master's thesis, Kansas State University, 2016.

[74] F. Y. Rashid. Library misuse exposes leading Java platforms to attack. http://www.infoworld.com/article/3003197/security/library-misuse-exposes-leading-java-platforms-to-attack.html, 2017.

[75] Resteasy Authorization design - check a user owns a resource. https://stackoverflow.com/questions/34315838/resteasy-authorization-design-check-a-user-owns-a-resource.

[76] RF 6101 - The Secure Sockets Layer (SSL) Protocol Version 3.0. https://tools.ietf.org/html/rfc6101.

[77] Scrapy – A Fast and Powerful Scraping and Web Crawling Framework. https://scrapy.org.

[78] Security - Allowing Java to use an untrusted certificate for SSL/HTTPS connection. https://stackoverflow.com/questions/1201048/allowing-java-to-use-an-untrusted-certificate-for-ssl-https-connection.

[79] Security exception when loading web image in jar. https://stackoverflow.com/questions/2011407/security-exception-when-loading-web-image-in-jar.

[80] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modeling analysis and auto-detection of cryptographic misuse in Android applications. In *Proceedings of the IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, DASC, pages 75–80, Washington, DC, USA, 2014. IEEE Computer Society.

[81] E. Smith and D. L. Dill. Automatic formal verification of block cipher implementations. In *Formal Methods in Computer-Aided Design*, pages 1–7, Nov 2008.

[82] Spring security. https://projects.spring.io/spring-security/.

[83] Spring Security 4 XML configuration UserDetailsService authentication not working. https://stackoverflow.com/questions/41321176/spring-security-4-xml-configuration-userdetailsservice-authentication-not-workin.

[84] Spring security JDK based proxy issue while using @Secured annotation on Controller method. https://stackoverflow.com/questions/35860442/spring-security-jdk-based-proxy-issue-while-using-secured-annotation-on-control.

[85] Spring Security Reference. http://docs.spring.io/spring-security/site/docs/3.2.4.RELEASE/reference/htmlsingle/#jc-httpsecurity.

[86] Spring Security Tutorial. http://www.mkyong.com/tutorials/spring-security-tutorials/.

[87] Spring Security using JBoss <security-domain>. https://stackoverflow.com/questions/28172056/spring-security-using-jboss-security-domain.

[88] SSL Certificate Verification: javax.net.ssl.SSLHandshakeException. https://stackoverflow.com/questions/25079751/ssl-certificate-verification-javax-net-ssl-sslhandshakeexception.

[89] SSL handshake fails with unable to find valid certification path to requested target. https://stackoverflow.com/questions/40977556/ssl-handshake-fails-with-unable-to-find-valid-certification-path-to-requested-ta.

[90] SSL Socket Connection working even though client is not sending certificate? https://stackoverflow.com/questions/26761966/ssl-socket-connection-working-even-though-client-is-not-sending-certificate.

[91] StackOverflow. https://stackoverflow.com.

[92] J. Steven and J. Manico. Password storage cheat sheet. https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet.

[93] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, 2017. https://eprint.iacr.org/2017/190.

[94] The Webserver I talk to updated its SSL cert and now my app can't talk to it. https://stackoverflow.com/questions/5758812/the-webserver-i-talk-to-updated-its-ssl-cert-and-now-my-app-cant-talk-to-it.

[95] Trusting all certificates using HttpClient over HTTPS. https://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https.

[96] Use of ECC in Java SE 1.7. https://stackoverflow.com/questions/24383637/use-of-ecc-in-java-se-1-7.

[97] Using public key from authorized_keys with Java security. https://stackoverflow.com/questions/3531506/using-public-key-from-authorized-keys-with-java-security.

[98] State of software security. https://www.veracode.com/sites/default/files/Resources/Reports/state-of-software-security-volume-7-veracode-report.pdf, 2016. Veracode.

[99] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD, 2004. http://eprint.iacr.org/2004/199.

[100] Web Security Samples. https://github.com/spring-projects/spring-security-javaconfig/blob/master/samples-web.md#sample-multi-http-web-configuration.

[101] WebSphere Application Server - IBM. http://www-03.ibm.com/software/products/en/appserv-was.

[102] When a TrustManagerFactory is not a TrustManagerFactory (Java). https://stackoverflow.com/questions/14654639/when-a-trustmanagerfactory-is-not-a-trustmanagerfactory-java.

[103] When I try to convert a string with certificate, exception is raised. https://stackoverflow.com/questions/10594000/when-i-try-to-convert-a-string-with-certificate-exception-is-raised.

[104] WildFly. http://wildfly.org.

[105] Wildfly 9 security domains won't work. https://stackoverflow.com/questions/37425056/wildfly-9-security-domains-wont-work.

[106] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun. What security questions do developers ask? A large-scale study of Stack Overflow posts. *Journal of Computer Science and Technology*, 31(5):910–924, Sep 2016.

[107] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. https://www.cs.utexas.edu/~shmat/courses/library/zeller.pdf, 2008.