Ensuring Correct Cryptographic Algorithm and Provider Usage at Compile Time

Weitian Xing
University of Waterloo
Waterloo, Canada
w23xing@uwaterloo.ca

Yuanhui Cheng
University of Waterloo
Waterloo, Canada
y82cheng@uwaterloo.ca

Werner Dietl
University of Waterloo
Waterloo, Canada
wdietl@uwaterloo.ca

ABSTRACT

Using cryptographic APIs to encrypt and decrypt data, calculate digital signatures, or compute hashes is error prone. Weak or unsupported cryptographic algorithms can cause information leakage and runtime exceptions, such as a NoSuchAlgorithmException in Java. Using the wrong cryptographic service provider can also lead to unsupported cryptographic algorithms. Moreover, for Android developers who want to store their key material in the Android Keystore, misused cryptographic algorithms and providers make the key material unsafe.

We present the Crypto Checker, a pluggable type system that detects the use of forbidden algorithms and providers at compile time. For typechecked code, the Crypto Checker guarantees that only trusted algorithms and providers are used, and thereby ensures that the cryptographic APIs never cause runtime exceptions or use weak algorithms or providers. The Crypto Checker is easy-to-use: it allows developers to determine which algorithms and providers are permitted by writing specifications using type qualifiers.

We implemented the Crypto Checker for Java and evaluated it with 32 open-source Java applications (over 2 million LOC). We found 2 issues that cause runtime exceptions and 62 violations of security recommendations and best practices. We also used the Crypto Checker to analyze 65 examples from a public benchmark of hard security issues and discuss the differences between our approach and a different static analysis in detail.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; • **Software and its engineering** → **Automated static analysis**.

KEYWORDS

Java, cryptography, pluggable type system, static analysis

ACM Reference Format:
Weitian Xing, Yuanhui Cheng, and Werner Dietl. 2021. Ensuring Correct Cryptographic Algorithm and Provider Usage at Compile Time. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21), June 13, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3464971.3468418

FTfJP '21, June 13, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8543-5/21/06…$15.00
https://doi.org/10.1145/3464971.3468418

1 INTRODUCTION

Cryptographic APIs are hard to understand and use for developers who are not cryptographers [24], which causes significant security vulnerabilities. This paper focuses on one aspect of cryptographic API misuse: the inadvertent use of weak, unsupported, or disallowed cryptographic algorithms or providers.

A key method to select cryptographic algorithms in Java is `Cipher.getInstance(String transformation)`[1]. The transformation can be in one of two formats: `"algorithm/mode/padding"` or simply `"algorithm"`. In the latter case, default mode and padding values will be used. This method uses the installed providers, which offer the implementation of algorithms and other security services, to find the implementation for the requested transformation. Alternatively, developers can specify the provider via `Cipher.getInstance(String transformation, String provider)`.

A `NoSuchAlgorithmException` occurs when an algorithm that is unavailable in the environment is requested [19]. This can be caused by a misspelling or an incorrect assumption about the execution environment. For instance, `Cipher.getInstance("AESS/GCM/NoPadding")` throws a `NoSuchAlgorithmException` at run time, because there is no `"AESS"` algorithm. The Java cryptographic APIs could have used fixed enum constants to guarantee that only valid algorithms, modes, and paddings are used. However, strings were likely chosen because they allow much more flexibility in the evolution and customization of the APIs and their independent implementation by hardware and software providers.

When specifying the provider, developers must use algorithms that are supported by this provider. The following code compiles successfully, but throws a `NoSuchAlgorithmException` at run time since `PKCS7PADDING` is not a valid padding for the provider `SunJCE`. Developers should use the BouncyCastle (BC) provider instead.

```
// runtime error
Cipher.getInstance("AES/CBC/PKCS7PADDING", "SunJCE");
```

For Android developers who want to store the key material in the Android Keystore, when generating security keys, `AndroidKeyStore` needs to be used explicitly as the cryptographic service provider. Otherwise, the Android Keystore system cannot protect the key material from unauthorized use. Also, only a subset of algorithms is supported by the Android Keystore, which means that a wrong algorithm will lead to a runtime exception.

Similar to `NoSuchAlgorithmException`, `NoSuchProviderException` occurs at run time when the requested provider does not exist in the environment. For example:

```
// runtime error
KeyPairGenerator.getInstance("RSA", "WrongProvider");
```

[1] https://docs.oracle.com/javase/8/docs/api/javax/crypto/Cipher.html

Unsupported algorithms or providers result in runtime exceptions, which can be hard to reproduce and fix. Using weak algorithms is even worse, as applications continue to operate and there will be no errors at compile or run time. Using weak algorithms may cause the exposure of sensitive information [8]. Some common symmetric ciphers such as DES, IDEA, and RC4 are considered very insecure, because their 64-bit keys are too short and susceptible to brute-force attacks [1]. Similarly, hash functions MD5, MD4, SHA-1, and the ECB operation mode, are prone to vulnerabilities. The compiler will not warn against using weak algorithms and, as there is no runtime exception when a weak algorithm is used, an application can use weak algorithms for a long time. Therefore, finding the use of weak cryptographic primitives at compile time is essential to protect sensitive information.

In this paper, we present the Crypto Checker, which validates the possible values used for cryptographic algorithms and providers at compile time. It gives a strong guarantee that no forbidden algorithms or providers are used in an application, which helps developers keep sensitive information safe and avoids runtime exceptions. The Crypto Checker enforces several default security rules for algorithms and providers. Users can also indicate their own rules to meet their particular requirements by adding type annotations, which is very convenient and easy to understand. Part of our work, weak algorithm detection, was inspired by the AWS Crypto Policy Compliance Checker [22]. Our work provides security rules for the Android Keystore [16], supports provider checking, and presents a thorough evaluation. Furthermore, to handle programs that read cryptographic parameters from property files, we designed a property file handler that performs type refinements for Java property file APIs. The source code of the checker and the evaluation are openly available[2].

2 TYPE SYSTEM

In this section, we present the Crypto Checker type system, which guarantees that only allowed algorithms and providers are used. The presented ideas can be applied to any language, but we use Java for examples. Section 2.1 describes the type qualifiers of the type system. Section 2.2 discusses the qualifier hierarchy. Section 2.3 defines the type rules for assignments and pseudo-assignments.

The type system performs a modular, conservative over-approximation of all possible executions of a program. The type system reports a false positive when it cannot guarantee a correct usage. In our case studies, there were no false positives.

2.1 Type Qualifiers

Type qualifiers are used to specify properties that cannot be expressed by the standard type system [4, 12]. Java's type annotation syntax can be used to represent type qualifiers [18]. Developers use the annotations in source code to specify properties of the program. In our type system, there are five type qualifiers: `@AllowedAlgorithms`, `@AllowedProviders`, and `@StringVal` provide information about allowed algorithms, providers, and string values, respectively; `@Unknown` and `@Bottom` complete the type lattice (see Section 2.2).

[2] https://github.com/vehiloco/crypto-checker

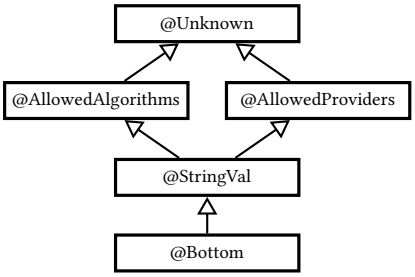

Figure 1: The basic qualifier hierarchy of the Crypto Checker's type system. Arrows represent the subtyping relationships between types.

Type qualifiers `@AllowedAlgorithms` and `@AllowedProviders` record the permitted algorithms and providers, as `String` arrays. For example, `@AllowedAlgorithms({"AES/GCM/NoPadding", "RSA"})` indicates that there are only two legal cipher transformations, `AES/GCM/NoPadding` and `RSA`. Developers can also use regular expressions to make the type qualifiers more expressive. For example, `@AllowedAlgorithms({"HmacSHA(1|224|256)"})` expresses that algorithms `HmacSHA1`, `HmacSHA224`, and `HmacSHA256` are allowed.

Type qualifier `@StringVal` expresses permitted String values, again as an `String` array. Most commonly, `@StringVal` is automatically determined for String literals in the program. This constant value propagation is provided by the Constant Value Checker [13] in the Checker Framework. For example, the String literal `"RSA"` has the type qualifier `@StringVal({"RSA"})`. In contrast to `@AllowedAlgorithms` and `@AllowedProviders`, `@StringVal` does not use regular expressions to describe possible values.

Type qualifier `@Unknown` is the top and default type qualifier in the type system. It indicates that no information about the algorithm or provider is known. The type system is conservative: when the type system cannot determine a more precise type, e.g., `@AllowedAlgorithms` or `@AllowedProviders`, the top type will be used. Type qualifier `@Bottom` is the bottom type and is used internally by the type system; developers do not need to use it explicitly.

2.2 Qualifier Hierarchy

These type qualifiers form an easy-to-understand qualifier hierarchy (type qualifier lattice), which is shown in Figure 1. `@AllowedAlgorithms` and `@AllowedProviders` are subtypes of `@Unknown` and supertypes of `@StringVal`, while `@Bottom` is the bottom type in the type system, subtype of `@StringVal`.

When determining the subtyping relation between two `@AllowedAlgorithms` or two `@AllowedProviders` type qualifiers, the `String[]` annotation type element also needs to be considered. The following rule applies: for two types τ_1 and τ_2, τ_1 is a subtype of τ_2 if and only if the element value of τ_2 contains the element value of τ_1. To make it more concrete, `@AllowedAlgorithms({"a"})` is a subtype of `@AllowedAlgorithms({"a", "b"})`. Two `@StringVal`'s subtyping relation is similar to the above rule [14].

These subtyping rules are sound but conservative: it is computationally hard to decide whether a regular expressions is subsumed by another regular expression, that is, whether the set of

strings accepted by two regular expressions are subsets. To resolve this, our type system only checks whether the supertype literally contains all the values in the subtype. For example, although the regular expression `SHA-(256|512)` matches `SHA-256`, `@AllowedAlgorithms({"SHA-(256|512)"})` is not a supertype of `@AllowedAlgorithms({"SHA-256"})` while `@AllowedAlgorithms({"SHA-256", "SHA-512"})` is.

For the subtyping relation between `@StringVal` and `@AllowedAlgorithms`, or `@StringVal` and `@AllowedProviders`, our type system has the following rule: `@StringVal` is a subtype of `@AllowedAlgorithms` or `@AllowedProviders` if and only if `@StringVal`'s element values match one of the regular expressions in `@AllowedAlgorithms` or `@AllowedProviders`. As the arguments to `@StringVal` do not use regular expressions, the type system simply needs to check whether the string values match the regular expressions.

2.3 Type Rules for Assignment and Pseudo-assignment

The subtyping rules from Section 2.2 are used wherever the type system performs subtype checks, in particular for assignments and pseudo-assignments.

For normal assignments, the type system checks whether the type of the right-hand side is a subtype of the left-hand side's type. An example is demonstrated below. As discussed before, `@StringVal` is the default type of String literals, and has the same element value as the String literal. Hence, the String literal `"SHA-256"` has type `@StringVal({"SHA-256"})`. As `SHA-256` matches the regular expression `SHA-(256|512)`, i.e., `@StringVal({"SHA-256"})` is a subtype of `@AllowedAlgorithms({"SHA-(256|512)"})`, this assignment typechecks.

```
@AllowedAlgorithms({"SHA-(256|512)"}) String algo;
algo = "SHA-256"; // correct
```

In contrast, the following assignment check fails because type qualifier `@StringVal({"SHA-384"})` is not a subtype of `@AllowedAlgorithms({"SHA-(256|512)"})`:

```
algo = "SHA-384"; // error
```

Pseudo-assignments have many forms, such as passing an argument to a method invocation. The type system checks whether the passed argument's type is a subtype of the parameter's type. For example, for `Cipher.getInstance("algorithm", "provider")`, it ensures the passed algorithm and provider argument types are subtypes of the specifications from the parameter types.

In the following code, we annotate the parameter of the method `KeyGenerator.getInstance(String a)` with `@AllowedAlgorithms` to specify that only `HmacSHA256` and `HmacSHA512` are accepted by the method. The passed arguments, String literals `"HmacSHA256"` and `"HmacSHA1"`, have type `@StringVal({"HmacSHA256"})` and `@StringVal({"HmacSHA1"})`, respectively. The former type matches the regular expression, while the latter one does not. Thus, `@StringVal({"HmacSHA1"})` is not a subtype of `@AllowedAlgorithms({"HmacSHA(256|512)"})`, and the type system reports an error.

```
class KeyGenerator {
  static KeyGenerator getInstance(
    @AllowedAlgorithms({"HmacSHA(256|512)"}) String a);
}
KeyGenerator.getInstance("HmacSHA256"); // correct
KeyGenerator.getInstance("HmacSHA1"); // error
```

This extended subtype checking applies everywhere the programming language performs subtype checks, e.g., to validate that a type argument is a subtype of a type parameter bound. We forego a soundness proof for this type system and instead rely on a standard type lattice and extended subtyping checks, which has been successfully used for other systems [9].

3 CRYPTO CHECKER

We present the Crypto Checker, a pluggable type system for Java, which implements the type system described in Section 2 and enforces the correct usage of algorithms and providers at compile time. The Crypto Checker is built using the Checker Framework [9, 27], which helps developers create pluggable type checkers. The Crypto Checker is written with only 376 non-blank, non-comment lines of Java code. Like other checkers based on the Checker Framework, the Crypto Checker performs modular type checking and flow-sensitive type refinement. Modular type checking analyzes each method and class independently, which makes it fast and light-weight. Flow-sensitive type refinement uses the control flow of the program to refine type information. The Crypto Checker is pluggable, it can be used together with other type checkers to enforce multiple properties, e.g., with the Checker Framework's built-in Nullness and Tainting Checkers. Moreover, specifications for binary-only code can be provided through stub files (minimal Java source files that contain the annotations for external APIs). The Crytpo Checker integrates into the normal Java build process and produces error messages in the standard Java format.

Enforced Cryptographic Rules. The Crypto Checker implements several default security rules extracted from security and static analysis papers [1, 6, 11, 26] and the cryptographic API documentation [15, 16, 25, 31]. These rules are normally considered safe and reasonable. Stub files contain the annotated code that indicates the allowed algorithms and providers. Developers can supply different stub files to the Crypto Checker to apply different rules:

- `cipher.astub` stores the security rules of `javax.crypto.Cipher`.
- `messagedigest.astub` stores the security rules of `java.security.MessageDigest`.
- `hardwarebacked.astub` stores the security rules of the Android Hardware-backed Keystore.
- `strongboxbacked.astub` stores the security rules of the Android Strongbox-backed Keystore.

Most developers only use cryptographic APIs and they can use the provided stub files to follow best practices. Organizations may want to create their own stub files or annotate their own cryptographic APIs with specifications. All the stub files are available with the Crypto Checker[3].

The specifications from stub files are trusted by the Crypto Checker, that is, there is no verification that the behavior of the methods matches their specifications. The Crypto Checker type checks Java source code, so it cannot verify whether stub files for bytecode-only libraries are correct. Care needs to be taken when encoding security rules in stub files and edits to stub files need to be audited by security experts. In contrast, the Crypto Checker ensures that type-checked source code follows all type rules, which

[3] https://vehiloco.github.io/crypto-checker/#stub-files

includes ensuring that additional specifications in source code are used correctly.

Regular expressions allow compact encoding of the security policies, in particular when compound algorithms are involved, where using a pattern of the form

```
PBEWith(SHA1|SHA2|SHA3)with(DES1|DES2|DES3)
```

is much simpler than enumerating all combinations. See stub file `cipher.astub` for examples.

Property File Handling. Cryptographic parameters such as algorithms and providers are commonly stored in property files. This makes enforcing a common standard easy and allows flexible reconfiguration. We found this pattern in many Java projects, e.g., Eclipse's jgit [10] and Apache's commons-cipher [2].

We added special handling in order to avoid false positive warnings caused by conservative over-approximation of the values returned by property files. A simple example is:

```
String CIPHER_ALGORITHM = "cipher.algorithm";
Cipher cipher = Cipher.getInstance(prop.getProperty(
     CIPHER_ALGORITHM));
```

As the Crypto Checker is conservative, it would warn about the use of the value read from the property as cryptographic algorithm. In the above example, `prop` is an instance of the `Properties` class which contains a set of properties. `prop.getProperty(String key)` searches the provided key in the property set and returns the corresponding value if this specific key exists. Otherwise, `null` will be returned. There is another method `prop.getProperty(String key, String defaultValue)` which will return the default value if the key does not exist.

A `Properties` instance is usually loaded from a property file, for example:

```
Properties prop = new Properties();
InputStream inputStream = getClass().getClassLoader().
     getResourceAsStream("config.properties");
prop.load(inputStream);
```

This kind of design establishes a barrier for static analysis tools which need to extract the corresponding algorithm from property files. To the best of our knowledge, there is no security tool that can detect security flaws by looking through cryptographic parameters in property files. SonarSource[29], a code analyzer for Java projects, has a test suite with Properties. However, SonarSource only checks the default value, not the value from a configuration file:

```
void usingJavaUtilProperties(Properties props) {
  Cipher.getInstance(props.getProperty("myAlgo", "DES/ECB/
       PKCS5Padding"));
}
```

The above code is compliant when the corresponding value of the key `myAlgo` in `props` is a safe cipher algorithm. If `myAlgo` does not exist, then the weak algorithm `DES/ECB/PKCS5Padding` will be used, which is unsafe and should be reported to the developers. SonarSource and other security tools do not analyse the properties file and only check the default value. If the default value conforms to the security rules, then it will pass the static analysis check while the value in the properties file is ignored by the security tools. This can lead to false positives or, even worse, false negatives: the algorithm in the property is unsafe while the default algorithm is safe, i.e., the unsafe algorithm will be used at run time, but the program can pass the checks because only the default value, which is a safe algorithm, is checked.

The Crypto Checker performs type refinement to handle reading cryptographic parameters from property files. Consider the following property file and the code snippet:

```
# a.properties
 cipher=DES

# PropertyFileRead.java
 Properties prop = new Properties();
 InputStream inputStream = getClass().getClassLoader().
      getResourceAsStream("a.properties");
 prop.load(inputStream);
 Cipher.getInstance(prop.getProperty("cipher"));
```

From the `getResourceAsStream()` call, the Crypto Checker propagates the property file's information to the return type of the method invocation. When loading properties from the input stream, the Crypto Checker keeps propagating the property file's name to the receiver, i.e., the object `prop` itself. Finally, when `prop.getProperty()` is called, the Crypto Checker will try to read from the properties file as well to identify the property value. Then, the property value will be added to the `@StringVal` annotation for the return type of `prop.getProperty()`. For the example above, the Crypto Checker views the code `Cipher.getInstance(prop.getProperty("cipher"));` as equal to:

```
@StringVal("DES") String cipher = prop.getProperty("cipher");
Cipher.getInstance(cipher);
```

Moreover, if a default value is provided when reading a property, the property file handler will also add the default value to the `@StringVal` annotation. This will propagate both the default value and the value determined from the file to the cryptographic APIs, ensuring that both values are secure. Let us add `"AES"` as default value to the example, resulting in:

```
@StringVal({"DES", "AES"}) String cipher = prop.getProperty(
   "cipher", "AES");
Cipher.getInstance(cipher);
```

When the key does not exist in the property file, only the default value will be added as `@StringVal` annotation type element. A "key-not-found" warning will also be reported from the Crypto Checker to help users correct their configuration files.

The property file handler is designed to be conservative to keep the Crypto Checker sound: if, for whatever reason, the Crypto Checker can not open and read a property file successfully, it will treat the result of `props.getProperty()` as an unknown String value. Thus, the Crypto Checker will issue an error if that unknown String value is used in a cryptographic API. Users need to make sure that the compile-time and run-time environments match, as the Crypto Checker cannot control the content of loaded property files at runtime. If property files might change at runtime, the conservative defaults must be used instead, by not using the property file handler.

4 CASE STUDIES

To evaluate the Crypto Checker's capability of detecting misuses of cryptographic algorithms and providers in Java applications, we ran the checker on 32 open-source projects consisting of 18 standard Java applications and 14 Android applications. These projects

were chosen based on their popularity and relevance to cryptography. The 18 standard Java applications use either or both of the `Cipher` and `MessageDigest` APIs; the 14 Android applications use either or both of the `KeyGenerator` or `KeyPairGenerator` APIs. In total, the Crypto Checker found security issues in 15 out of the 18 standard Java applications and 5 out of the 14 Android Applications. We reported the security issues to the top 5 Java applications that have the most issues and got responses from the Eclipse Californium team. We also used 65 test cases from the CRYPTOAPI-BENCH [1] to evaluate the Crypto Checker's performance. After type checking, we examined each error reported by the checker and added annotations where necessary to ensure the correct usages of cryptographic primitives.

For each standard Java project, we supplied two stub files (`cipher.astub` and `messagedigest.astub`). The results indicate insecure uses of `Cipher` and `MessageDigest` (see Section 4.1). For each Android project, we supplied the `hardwarebacked` stub file, and the results indicate unsupported uses of `KeyGenerator` or `KeyPairGenerator` by the Android Hardware-backed Keystore (see Section 4.2). Section 4.3 discusses the Crypto Checker's performance with CRYPTOAPI-BENCH.

4.1 Insecure Uses of Cryptographic APIs

The Crypto Checker issued 64 errors in 15 of the 18 analyzed standard Java applications and found no issues in the remaining 3 applications. Only 9 annotations were manually added to 3 of the 15 applications, to precisely specify the expected behavior (note that defaults, flow-sensitive type refinement, and property file handling are enough for most code; annotations are only needed when specifications cross method boundaries). The discovered errors include two bugs from Eclipse Californium where invalid arguments were passed as the cipher transformations, which can cause `NoSuchAlgorithmException`. The other 62 errors are all defects that could cause cryptographic vulnerabilities, and they can be further categorized into two types: 54 insecure cryptographic algorithms (see Section 4.1.1) and 8 unsafe public methods (see Section 4.1.2). No false positives are reported by the Crypto Checker. The evaluation results of the 15 projects are listed in Table 1 in the Appendix. The repository URLs for these 18 projects are listed in Table 2 in the Appendix.

4.1.1 Insecure Cryptography. Overall, the Crypto Checker found 54 insecure cryptographic algorithms in these 15 applications, which we classified into four categories:

- Category 1: 11 uses of insecure mode ECB for encryption;
- Category 2: 10 uses of cipher transformations without providing cipher mode or padding schema;
- Category 3: 11 uses of insecure ciphers; and
- Category 4: 32 uses of insecure hash functions.

The sum of the misuses in the list above is 64 rather than 54 since some cipher transformations break multiple rules, such as DES/ECB/PKCS5Padding insecurely applies ECB and an insecure cipher. We count both of these misuses as one defect.

The results for each app are summarized in Table 1 in the Appendix, and we discuss each category with examples next.

Category 1. Electronic Codebook (ECB) mode encrypts the same plaintext blocks to identical ciphertext blocks, which makes it possible to leak information. Hence, it should not be used as the mode of operation to encrypt data. Here is an example from class `EncryptUtil` in Apache Kylin that uses insecure ECB mode:

```
// insecure ECB mode
Cipher.getInstance("AES/ECB/PKCS5Padding");
```

However, RSA/ECB/OAEPPADDING is secure to use since ECB processes on blocks while RSA does not break the message into blocks, which indicates that RSA does not really apply the ECB mode [25, 32]. Hence, the Crypto Checker treats RSA/ECB/OAEPPADDING as a safe transformation.

Category 2. When generating a cipher instance, introducing the cipher algorithm without the mode of operation or the padding schema is discouraged. A default mode of operation and padding schema would be used at run time, which could result in a false sense of security. The following example is extracted from class `FileSystemConsumerStore` in Eclipse's lyo.server. The standalone cipher algorithm, `AES`, defaults to insecure ECB mode that triggers a misuse of the mode of operations:

```
// defaults to AES/ECB/..., which is insecure
Cipher.getInstance("AES");
```

However, there is an exception for RSA. It is allowed to only specify RSA in a cipher transformation, without providing the mode of operation and padding schema. RSA defaults to RSA/ECB/PKCS1Padding [3], which is secure. Therefore `Cipher.getInstance("RSA")` is secure, which is used frequently in real-world applications.

Category 3. Insecure ciphers such as DES, Blowfish, and RC4 make brute-force attacks possible and should be forbidden. The following example from class `DESUtils` in project whatsmars uses one of the insecure ciphers, `DES`:

```
private static final String PADDING = "DES/ECB/PKCS5Padding";
// use of insecure cipher algorithm
Cipher cipher = Cipher.getInstance(PADDING);
```

Category 4. An insecure hash function such as SHA1, MD4, and MD5 could cause collisions, which take different input but generate the same output. Hence, we only permit using strong hash functions to produce hash values or message digests. The cipher transformation could also apply an insecure hash function, such as PBEWithMD5AndDES, which uses an insecure hash function and insecure cipher simultaneously.

Here is an example, from class `MessageDigestUtils` in project async-http-client, that uses the insecure hash function `SHA-1`:

```
try {
  return MessageDigest.getInstance("SHA-1");
} catch (NoSuchAlgorithmException e) {
  throw new InternalError("SHA1 not supported");
}
```

For project smart, one developer opened an issue [20] to point out that an insecure hash function, MD5, is used. The Crypto Checker reports that MD5 is indeed used by the project and that such a use is insecure. For project flutter_secure_storage (one of the three projects that have no cryptographic misuses), developers have opened an issue [23] to discuss whether the cipher transformation RSA/ECB/PKCS1Padding is weak or not. They argued about

this issue and did not reach an agreement. Checking the whole project, the Crypto Checker reports that, according to our defined rules, RSA/ECB/PKCS1Padding is used securely.

4.1.2 Exposure of Cryptographic APIs through Public Methods. Sometimes cryptographic APIs are exposed by an application's public methods. These methods take the cryptographic algorithm as a parameter and are accessible to outside callers. In this situation, insecure algorithms might be used and make the program vulnerable to malicious attacks.

The Crypto Checker reported 8 occurrences of this vulnerability. Consider this example from class `Crypto` in project rapidoid:

```
public static Cipher cipher(String transformation) {
  try {
    return Cipher.getInstance(transformation);
  } catch (NoSuchAlgorithmException e) {
    ...
```

Calling `Cipher.getInstance(transformation)` raises security concerns, as correct use of cryptographic algorithms is not guaranteed. There are two possible solutions to this problem, depending on whether callers should be trusted or not. The Crypto Checker performs modular type checking, which allows developers to write specifications by adding annotations. If callers of the method can be trusted, for example, because they also use the Crypto Checker, developers can add `@AllowedAlgorithms` to the `transformation` parameter. This will ensure that the `transformation` parameter must take an allowed algorithm. For example, if only `AES/GCM/PKCS5Padding` should be allowed, the parameter can be annotated as `@AllowedAlgorithm({"AES/GCM/PKCS5Padding"})`.

If the public method can also be invoked by untrusted third parties, the method should perform a runtime check on the parameter to ensure a valid cryptographic algorithm is selected. Runtime checks are needed only in places where untrusted external invocations are possible. When annotating code, the developer can decide what the right solution is for each situation.

4.2 Android Keystore Case Study

To make it difficult to extract sensitive data from an Android device, Google introduced the Android Keystore System in Android 4.3. Keystore is used to keep key material in secure hardware, such as a Trusted Execution Environment (TEE) [7, 16]. This mechanism takes effect only if the following two conditions are satisfied: 1) `AndroidKeyStore` is used as the cryptographic service provider, and 2) the device's secure hardware supports the particular combination of transformations with which the key is authorized to be used [16].

However, developers might not use `AndroidKeyStore` as the provider even though their applications require high security. Developers might also use algorithms that are not supported by the `AndroidKeyStore` provider. For example, HmacMD5 is not supported by `AndroidKeyStore` but can be used with other providers.

To handle these cases, we used the Crypto Checker with `hardwarebacked.astub` to find three vulnerabilities: 1) `KeyGenerator.getInstance(algorithm)` where the provider is not specified, 2) `KeyGenerator.getInstance(algorithm, provider)` where the provider is not stated as `AndroidKeyStore`, and 3) `KeyGenerator.getInstance(algorithm, "AndroidKeyStore")` where the algorithm is not supported by `AndroidKeyStore`.

We tested the 14 security-sensitive Android applications listed in Table 3 in the Appendix. No extra annotations were needed. The Crypto Checker found that 4 out of the 14 projects were not using `AndroidKeyStore` as the provider when generating keys, which corresponds to vulnerability 1). We manually checked the source code of these 4 projects and observed that for two of them, `AndroidKeyStore` was never used across the whole program. In the remaining two projects, `AndroidKeyStore` was not used consistently: some of the cryptographic API uses designate `AndroidKeyStore` as the provider while some do not. In this case, the key material may not always be stored in the Android Hardware-backed Keystore. Both of the situations that miss the `AndroidKeyStore` could contribute to an insecure environment, which can lead to unauthorized uses of key material. For vulnerability 2), one project was found using a provider other than `AndroidKeyStore` to generate keys. For vulnerability 3), we did not find any violations among these 14 projects, which indicates that all the algorithms were used correctly.

The Crypto Checker guarantees correct usage only for checked source code. It gives no guarantees for sources it did not check, for example, third-party libraries. It also cannot control the environment in which the application is deployed. Users must make sure that their phone hardware and operating system support the Android Hardware-backed Keystore.

4.3 CRYPTOAPI-BENCH Case Study

The benchmark CRYPTOAPI-BENCH [1] consists of 171 test cases to evaluate the quality of cryptographic vulnerability detection tools. 65 out of the 171 test cases in the benchmark are about misuses of cipher and hash functions. We used these test cases as unit tests[4] to test the Crypto Checker's performance. The Crypto Checker found all the errors that are expected by the benchmark, and nine additional errors that we believe are noteworthy.

The benchmark covers field- and path-sensitive cases. As the Crypto Checker is a type system that performs modular type checking, we expect developers to add annotations to indicate the specifications. With 79 added annotations, the Crypto Checker can handle the field-sensitive cases. For path-sensitive cases, the Crypto Checker issues nine cryptographic misuses. Take this example:

```
method2(2); // only invocation of method2
public void method2(int choice) {
  Cipher cipher = Cipher.getInstance(insecureCipherAlgorithm);
  if (choice > 1) {
    cipher = Cipher.getInstance(secureCipherAlgorithm);
  }
}
```

The benchmark supposes that the above code is safe because the only observed call of the method uses a value that applies the secure cipher algorithm. This test case aims to evaluate whether a static analysis tool can properly perform whole-program value analysis and path-sensitive refinement. In contrast, the Crypto Checker treats this method as unsafe and it should not be trusted. Developers could, in a future version of the code, pass `values <= 1` to `method2`, which triggers the creation of an insecure cipher instance. Also, specifying the insecure cipher algorithm in the conditional branch is a code smell. Hence, we do not consider these 9 errors to be false

[4] https://github.com/vehiloco/crypto-checker/tree/master/tests/cryptoapibench

positives, but rather the result of different analysis goals. Moreover, it is rare for real-world applications to apply a secure or insecure cryptographic algorithm depending on the value of a parameter. This pattern did not come up in the 32 real-world applications.

5 RELATED WORK

There is a large body of work on static analyses for many different domains. In the following we can only review the most directly related work. We discuss work that focuses on detecting misuses of cryptographic APIs and compare them to the Crypto Checker. Since our approach focuses on algorithms and providers, this will be our main focus.

Algorithm Checking. The AWS Crypto Policy Compliance Checker (AWS Checker for short) is a type checker built on the Checker Framework [9], which checks if there are any usages of weak cipher algorithms in Java applications. Part of our work, weak algorithm detection, is based on the idea of this checker. The AWS Checker has two main type qualifiers, `@CryptoBlackListed` and `@CryptoWhiteListed`, to indicate the algorithms that are forbidden or allowed. It additionally has the `@SuppressCryptoWarning` annotation, which is used to suppress errors from non-whitelisted algorithms. This annotation can be used to document policy exceptions. For whitelisted algorithms, AWS Checker offers an option to issue warnings for algorithms that should not be used. For algorithm checking, compared to the AWS Checker, the Crypto Checker supports checking Android applications and also supplies a more comprehensive set of security rules to developers.

CRYPTOGUARD [28] uses on-demand slicing algorithms to detect cryptographic vulnerabilities. It can handle path and field-sensitive cases. CRYPTOGUARD additionally covers a large number of vulnerabilities, such as Rule 3 (Hardcoded Store Password), Rule 6 (Used Improper Socket), and Rule 7 (Used HTTP). For weak algorithm uses, it has the following related rules: Rule 14 (Symmetric Ciphers) and Rule 16 (Insecure Cryptographic Hash). CRYPTOAPI-BENCH, which we used as a case study in Section 4.3, also had an evaluation on CRYPTOGUARD. CRYPTOGUARD produced 10 false positives on the 65 test cases. By manually adding annotations to some of the test cases, the Crypto Checker achieved zero false positives. However, adding annotations may become a burden to programmers as 79 annotations were added to these test cases. Compared with CRYPTOGUARD's slicing algorithm, the Crypto Checker's modular type checking analyzes less information, which makes it more efficient but imprecise. Besides, modular type checking is more conservative: using an open-world assumption may find more potential vulnerabilities, such as exposures of public methods (Section 4.1.2).

Error Prone [17] is a widely-used open-source static analysis tool for Java. For cryptographic algorithm misuses, it has one bug pattern called `InsecureCryptoUsage` which includes three particular security rules: 1) Cipher instance should not be created with the insecure ECB mode, 2) Diffie-Hellman protocol is insecure and Elliptic Curves Diffie-Hellman (ECDH) should be used instead, and 3) do not use DSA for digital signatures.

Compared with Error Prone and CRYPTOGUARD, the Crypto Checker gives developers the freedom to set their permitted algorithm and provider rules easily using annotations. Users of Error Prone have to learn how to write a new checker to enforce new rules and CRYPTOGUARD has its rules hard-coded in the source code.

Some other static analysis tools support cryptographic algorithm checking, such as Coverity [30], SonarSource [29], SpotBugs [5], and LGTM [21]. SonarSource and SpotBugs are open-sourced, while Coverity and LGTM are not. SonarSource, SpotBugs, and LGTM support writing custom rules, but that requires learning internal APIs.

Provider Checking. We are not aware of any tools that support provider checking or security rules for the `AndroidKeyStore` provider.

Java Properties Handling. Compared with all the above tools, only the Crypto Checker can perform type refinement to Java Properties, which reduces both false positives and false negatives, making the static analysis more comprehensive and expressive.

6 CONCLUSION

One important cause of security vulnerabilities are cryptographic algorithm and provider misuses. To resolve this, we present a pluggable type system for Java-like programming languages and implement it for Java. It performs modular type checking to find forbidden algorithm and provider usages at compile time.

We evaluated the Crypto Checker pluggable type system on 32 open-source Java applications and found 2 bugs and 62 potential security vulnerabilities, including in well-maintained projects such as Apache Dubbo and Apache Kylin. More broadly, we demonstrate that pluggable type systems are an excellent option for source code analysis: sound and robust infrastructure for analysis designers and flexibility for tool users to use annotations to customize specifications, all while staying within a standard programming language. In the future, we plan to analyze more cryptographic misuses, e.g., insecure initialization vectors and incorrect pseudo-random number usage.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments and suggestions. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) (RGPIN-05799-2014, RGPIN-2020-05502, and CRDPJ-543583-2019) and an Early Researcher Award from the Government of Ontario. This research has also been supported by a grant from the WHJIL. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSERC, WHJIL, or the Governments of Ontario or Canada.

REFERENCES

[1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2020. A Comprehensive Benchmark on Java Cryptographic API Misuses. In *Data and Application Security and Privacy*. 177–178.

[2] Apache. 2021. *Apache Commons Crypto.* Retrieved April 24, 2021 from https://github.com/apache/commons-crypto

[3] Maarten Bodewes. 2016. *Stack Overflow: Java - Default RSA padding in SUN JCE/Oracle JCE.* Retrieved April 24, 2021 from https://stackoverflow.com/questions/21066902/default-rsa-padding-in-sun-jce-oracle-jce

[4] Gilad Bracha. 2004. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages.*

[5] Spot Bugs. 2021. *SpotBugs: Find bugs in Java Programs.* Retrieved April 24, 2021 from https://spotbugs.github.io/

[6] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2016. Evaluation of cryptography usage in Android applications. In *Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. 83–90.

[7] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. 2014. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. 11–20.

[8] The MITRE Corporation. 2021. *CWE-327: Use of a Broken or Risky Cryptographic Algorithm.* Retrieved April 24, 2021 from https://cwe.mitre.org/data/definitions/327.html

[9] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. 2011. Building and Using Pluggable Type-Checkers. In *Software Engineering in Practice Track, International Conference on Software Engineering (ICSE)*.

[10] Eclipse. 2021. *An implementation of the Git version control system in pure Java.* Retrieved April 24, 2021 from https://github.com/eclipse/jgit

[11] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *Computer and Communications Security (CCS)*. 73–84.

[12] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-sensitive type qualifiers. In *Programming Language Design and Implementation (PLDI)*. 1–12.

[13] Checker Framework. 2021. *Constant Value Checker.* Retrieved April 24, 2021 from https://checkerframework.org/manual/#constant-value-checker

[14] Checker Framework. 2021. *Constant Value Checker Qualifier Hierarchy.* Retrieved April 24, 2021 from https://checkerframework.org/manual/#fig-value-hierarchy

[15] Google. 2020. *Android Keystore Provider.* Retrieved April 24, 2021 from https://developer.android.com/training/articles/keystore#SupportedAlgorithms

[16] Google. 2020. *Android Keystore System.* Retrieved April 24, 2021 from https://developer.android.com/training/articles/keystore#HardwareSecurityModule

[17] Google. 2021. *Error Prone Bug Pattern: InsecureCryptoUsage.* Retrieved April 24, 2021 from https://errorprone.info/bugpattern/InsecureCryptoUsage

[18] JSR 308 Expert Group. 2021. *Type Annotations (JSR 308).* Retrieved April 24, 2021 from https://jcp.org/en/jsr/detail?id=308

[19] David Hook. 2005. *Beginning cryptography with Java.* John Wiley & Sons.

[20] Joe. 2020. *Issue: Cryptographic API misuse detected.* Retrieved April 24, 2021 from https://github.com/a466350665/smart/issues/47

[21] LGTM. 2021. *LGTM: Continuous security analysis.* Retrieved April 24, 2021 from https://lgtm.com/

[22] Martin Kellogg, Martin Schäf, Serdar Tasiran, Michael D. Ernst. 2020. *AWS Crypto-Policy Compliance Checker.* Retrieved April 24, 2021 from https://github.com/awslabs/aws-crypto-policy-compliance-checker

[23] mogol. 2021. *Issue: ECB Mode is Insecure.* Retrieved April 24, 2021 from https://github.com/mogol/flutter_secure_storage/issues/60

[24] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: Why do Java developers struggle with cryptography APIs?. In *International Conference on Software Engineering (ICSE)*. 935–946.

[25] Oracle. 2021. *Java Cryptography Architecture (JCA) Reference Guide.* Retrieved April 24, 2021 from https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

[26] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Inferring crypto API rules from code changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 450–464.

[27] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 201–212.

[28] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *Computer and Communications Security (CCS)*. 2455–2472.

[29] Sonar Source. 2021. *SonarSource builds world-class products for Code Quality & Security.* Retrieved April 24, 2021 from https://www.sonarsource.com/

[30] Synopsys. 2021. *Coverity Static Application Security Testing (SAST).* Retrieved April 24, 2021 from https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

[31] Carnegie Mellon University. 2020. *MSC61-J. Do not use insecure or weak cryptographic algorithms.* Retrieved April 24, 2021 from https://wiki.sei.cmu.edu/confluence/display/java/MSC61-J.+Do+not+use+insecure+or+weak+cryptographic+algorithms

[32] John R Vacca. 2013. *Cyber Security and IT Infrastructure Protection.* Syngress.

A CASE STUDY DETAILS

This section presents additional details of the case studies discussed in Section 4. Table 1 shows the case study statistics for Java applications. Tables 2 and 3 contain links to the repositories of the case studies.

Table 1: Case study statistics for standard Java applications. NCNB LOC stands for non-comment, non-blank lines of code. Manual Annotations is the number of annotations we added to each application. Columns C1 (Insecure ECB), C2 (Cipher Without Mode or Padding), C3 (Insecure Cipher), and C4 (Insecure Hash Function) are the four categories of insecure cryptography (Section 4.1.1). Unsafe Public Methods is the number of public methods that use cryptographic APIs (Section 4.1.2). C1 + C2 + C3 + C4 may not be equal to Total Defects, since some code violates multiple rules simultaneously.

Java Applications	NCNB LOC	Manual Annotations	Total Defects	C1	C2	C3	C4	Unsafe Public Methods
Apache Druid	639k	0	3	0	0	0	1	2
Apache Kylin	201k	0	4	2	0	0	2	0
Apache Dubbo	168k	0	2	0	0	0	2	0
redisson	149k	0	3	0	0	0	3	0
Eclipse Californium	87k	6	10	1	4	0	2	3
rapidoid	66k	2	4	0	0	0	2	2
NettyGameServer	34k	0	5	0	2	2	3	0
async-http-client	33k	0	9	4	0	5	4	0
whatsmars	28k	0	5	4	0	2	1	0
ha-bridge	18k	0	2	0	0	2	2	0
mongodb-rdbms-sync	15k	0	2	0	2	0	0	0
java-telegram-bot-api	11k	0	1	0	0	0	1	0
smart	5k	0	1	0	0	0	1	0
Eclipse Lyo Server	3k	0	2	0	2	0	0	0
aes-rsa-java	1k	1	9	0	0	0	8	1
Totals	1458k	9	62	11	10	11	32	8

Table 2: Repository URLs of Java applications listed in Table 1.

Java Applications	Repository URL
Apache Druid	https://github.com/apache/druid.git
Apache Kylin	https://github.com/apache/kylin.git
Apache Dubbo	https://github.com/apache/dubbo.git
redisson	https://github.com/redisson/redisson.git
Eclipse Californium	https://github.com/eclipse/californium.git https://github.com/xwt-benchmarks/californium.git
rapidoid	https://github.com/rapidoid/rapidoid.git https://github.com/xwt-benchmarks/rapidoid.git
NettyGameServer	https://github.com/jwpttcg66/NettyGameServer.git
async-http-client	https://github.com/AsyncHttpClient/async-http-client.git
whatsmars	https://github.com/javahongxi/whatsmars.git
ha-bridge	https://github.com/bwssytems/ha-bridge.git
mongodb-rdbms-sync	https://github.com/gagoyal01/mongodb-rdbms-sync.git
java-telegram-bot-api	https://github.com/pengrad/java-telegram-bot-api.git
smart	https://github.com/a466350665/smart.git
Eclipse Lyo Server	https://github.com/eclipse/lyo.server.git
aes-rsa-java	https://github.com/wustrive2008/aes-rsa-java.git https://github.com/xwt-benchmarks/aes-rsa-java.git
Elephent	https://github.com/jusu/Elephant.git
jpass	https://github.com/gaborbata/jpass.git
flutter_secure_storage	https://github.com/mogol/flutter_secure_storage.git

Table 3: Repository URLs of Android applications.

Android Applications	Repository URL
CacheManage	https://github.com/ronghao/CacheManage.git
fingerlock	https://github.com/aitorvs/fingerlock.git
wigle-wifi-wardriving	https://github.com/wiglenet/wigle-wifi-wardriving.git
FingerprintRecognition	https://github.com/PopFisher/FingerprintRecognition.git
LolliPin	https://github.com/omadahealth/LolliPin.git
PFLockScreen-Android	https://github.com/thealeksandr/PFLockScreen-Android.git
secure-quick-reliable-login	https://github.com/kalaspuffar/secure-quick-reliable-login.git
lock-screen	https://github.com/amirarcane/lock-screen.git
BiometricPromptDemo	https://github.com/gaoyangcr7/BiometricPromptDemo.git
Fingerprint	https://github.com/OmarAflak/Fingerprint.git
connectbot	https://github.com/connectbot/connectbot.git
revolution-irc	https://github.com/MCMrARM/revolution-irc.git
Secured-Preference-Store	https://github.com/iamMehedi/Secured-Preference-Store.git
jpico	https://github.com/mypico/jpico