Light-weight verification of cryptographic API usage

by

Weitian Xing

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Weitian Xing is the sole author for Chapters 1, 2, and 7. They are written under the supervision of Professor Werner Dietl, and are not yet published.

Chapter 3 of the thesis introduces a type system aiming at preventing the use of forbidden cryptographic algorithms and providers at compile time. It is co-authored with Yuanhui Cheng and Werner Dietl and not yet published. Chapters 4, 5, and 6 contain figures, tables, and text from this unpublished paper.

Abstract

A pluggable type system is a light-weight approach for compile-time program verification, which provides more powerful types to both developers and compilers. Developers use pluggable types to boost program understanding, while compilers leverage the new types to enforce interesting properties, *e.g.*, NullPointerException freedom. This thesis presents a new type system, the Crypto Checker, to help developers prevent cryptographic APIs misuse. In addition, this thesis presents the Property File Handler and the Network Request Checker as two type system extensions. The Property File Handler performs type refinement to Java Properties by reading property files at compile time, while the Network Request Checker reports all the possible network requests to prevent potential information leakage in Java and Android applications.

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. This thesis presents 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 type system consists of an easy-to-understand type qualifier hierarchy. The Crypto Checker is flexible and 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.

Malicious or unsafe applications collect and send users' data to untrusted external servers via network requests, *e.g.*, HTTP and socket requests, which will cause information leakage. Detecting the possible network requests on the source code level without running the applications is a light-weight approach to solve the problem. Application stores that have the source code of the uploaded apps can take advantage of this to ensure application security. Security teams in companies also use similar technologies to guarantee compliance. This thesis presents the Network Request Checker, a type system extension for Java to

detect and report all the possible network requests to developers at compile time. The Network Request Checker can be integrated into any other pluggable type system or be seen as a stand-alone type system depending on developers' needs. We evaluated this type system with 6 real-world Java and Android applications and discuss the experimental results.

To improve the Crypto Checker, the Network Request Checker, and other type systems' precision, *i.e.*, to obtain more valuable information from the program at compile time, this thesis presents the Property File Handler, a type system extension that reads property files to perform type refinement on Java Properties. When an application reads property files, the Property File Handler will also try to load, store, and propagate the information from property files. A simple type hierarchy is proposed to achieve this functionality. By using the Property File Handler, we found and fixed a potential false negative with Java Properties, while other static analysis tools, *e.g.*, SonarSource, did not handle that code correctly.

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Professor Werner Dietl, for his continuous support and guidance. He opened the door to a new world for me. I would also like to thank my readers, Professor Arie Gurfinkel and Professor Mahesh Tripunitara, for their time, questions, and feedback. I am also thankful to Professor Arie Gurfinkel for his advice in project meetings. I thank my friends in the research group, especially Tongtong Xiang and Lian Sun, for their help and for buying tons of Tim Hortons with me. I thank my former URA, Yuanhui Cheng, for working on the Crypto Checker with me.

Dedication

This is dedicated to my parents who give me endless love, support, and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Many developers are suffering from cryptographic APIs misuse: most developers are not cryptography experts, which means that it is easy for them to make mistakes when the documentation is not clear and easy enough. Some of the misuses, e.g., using weak cryptographic algorithms, will not cause runtime exceptions, which makes the misuses hard to be reproduced and fixed as the programs will not throw errors to alert the developers. There are many security tools, both static and dynamic, created to detect weak cryptographic algorithm misuses. But most of them cannot provide developers a convenient way to specify custom security rules: some security tools hardcode security rules in their source code; for the security tools that support custom rules, users always have to learn template models or write sub checkers themselves. Furthermore, currently, there are no such tools for detecting cryptographic algorithm providers, which is also important to avoid runtime exceptions and make it possible for developers to specify the security providers they want to use. Because static tools run at compile time, i.e., they do not put extra burdens on the runtime of the program, we want to create a static tool which not only supports detecting both forbidden cryptographic algorithms and providers but also offers developers the flexibility to indicate their own security rules easily. A pluggable type system meets our requirements as 1) it performs type checking at compile time, 2) its type qualifier can express developer-defined security rules easily, and 3) it also improves program understanding.

Information leakage can cause major harm to applications and users: malicious applications collect and send the users' or other applications' information to untrusted external servers by sending network requests, e.g., HTTP and socket requests. This helps attackers

learn more about applications. The leak of users' sensitive data causes a loss of money. We create a type system extension which can detect all the possible network requests at compile time, so that developers can leverage this type system extension to perform security and privacy checking when building applications. As a type system extension, it can be used by any arbitrary type systems.

Property files in applications always store valuable information. For instance, to enforce a common security standard and allow global reconfiguration easily, applications save cryptographic parameters such as cryptographic algorithms and providers in project property files. Similarly, the hostnames and ports for network requests are commonly stored in property files. Thus, getting such information at compile time helps static analysis tools perform more comprehensive and precise analysis. Unfortunately, the current existing static analysis tools cannot handle property files, which introduces false positives, and even false negatives. We create a type system extension that handles property files to provide more compile-time constant values.

1.2 Thesis Contributions and Organization

The thesis contains the following main contributions:

Firstly, we present the Crypto Checker, which is a pluggable type system aimed at ensuring that there are no uses of forbidden cryptographic algorithms and providers in a program. For the Crypto Checker, we provide several pre-defined security rules for Java and Android applications. We evaluate the Crypto Checker with over 2 million lines of code and performed a case study with a comprehensive cryptographic APIs misuse benchmark. See Chapter 3 for detailed information.

Secondly, we present the Property File Handler as a type system extension performing flow-sensitive type refinement to enhance the compile-time constant value inference in Chapter 4.1. The Property File Handler tries to read the programs' property files and propagates the values in the property files to Java properties. The Property File Handler can collaborate with any Checker Framework based type system. The Crypto Checker and the other type systems benefit from the Property File Handler as it improves precision and reduces false positives.

Thirdly, we present the Network Request Checker, another type system extension that reports all the possible network requests at compile time. The Network Request Checker supports several popular network request libraries by default, and it can also be extended easily to be compatible with more libraries. We evaluate the Network Request Checker

with 6 real-world Java and Android applications, and discuss the experimental results. See Chapter 4.2 for detailed information.

The rest of the thesis is structured as follows: Chapter 2 discusses the basic knowledge of type systems and the Checker Framework, which is the fundamental of the pluggable type system and extensions presented in the thesis. Chapter 3 introduces the type system and the implementation of the Crypto Checker. Chapter 4 describes two type system extensions, the Property File Handler and the Network Request Checker. Chapter 5 talks about case studies. Chapter 6 discusses the related work. Chapter 7 records possible future work and concludes the whole thesis.

Chapter 2

Background

This chapter discusses the background knowledge that is needed to understand the thesis.

2.1 Type Systems

Type systems are always an important component of statically-typed programming languages as the type systems provide a set of rules to the programs, which improves both the security and the performance of the programs. To ensure developers follow the provided rules, programming languages utilize type checking algorithms, which will report errors when the rules are violated. After passing the type checking, guarantees of the correct use of the types are given to the programs [34, 38, 46]. Compilers can also optimize the code after knowing that the type rules are respected.

As a statically typed language, Java's built-in type system is strong and helps prevent many errors during compilation, e.g., as the following code is shown, assigning a string literal to an integer type variable will cause an incompatible types error. This helps developers fix errors at compile time.

```
int i = "123"; // incompatible types error
```

However, Java's built-in type system is not strong enough to find all kinds of errors at compile time. For example, even though the Java compiler does not complain, developers still may face NoSuchAlgorithmException and NoSuchProviderException when using Java cryptographic APIs. We will discuss how to prevent these two exceptions in Chapter 3. To make Java's type system more powerful, the Checker Framework is introduced.

```
void foo(Object nn, @Nullable Object nbl) {
  nn.toString(); // OK
  nbl.toString(); // Error
}
```

Figure 2.1: An example of the Nullness Checker.

2.2 Checker Framework

The Checker Framework is a framework that enhances Java's original type system to perform extra type checking for interesting properties, e.g., null-pointer freedom and freedom of certain taints [7, 10, 17, 21]. The pluggable type system and extensions presented in this thesis are all built on the Checker Framework. For a specific property that passes pluggable type checking, the Checker Framework gives users the guarantee that there will be no runtime errors with this property. "Pluggable" here indicates that the type checking is optional: Users can freely choose which type checking they want to perform at every compilation. Also, the type annotations will be only validated at compile time, which will not put extra burdens on the run time. Thus, pluggable type checking is light-weight and flexible. The Checker Framework leverages Java type annotations which were introduced in JDK 8 to achieve the Checker Framework's core functionalities. Currently, the Checker Framework supports both JDK 8 and 11. An example of the Checker Framework's Nullness Checker extracted from the Checker Framework Live Demo[1] is shown in Figure 2.1. In this example, developers use the annotation `@Nullable` provided by the Nullness Checker to indicate that `null` can be passed as the argument to the parameter `Object nbl`. Thus, `nbl.toString()` is a dangerous operation that may cause NullPointerException at run time and should be avoided. The Nullness Checker will report this to developers at compile time.

The Checker Framework provides a good encapsulation of the Java's internal compiler APIs, i.e., for most of the time developers do not need to work with them. In addition, the Checker Framework also provides default implementations which reduces the code efforts to create a new checker, i.e., a new checker normally only needs to override some of the specific methods to meet its need. Thus, even the developers who do not have deep knowledge with type system and javac can easily create custom annotations and enforce certain type rules. To design and develop a new pluggable type system, the following 5 components are critical:

[1] http://eisop.uwaterloo.ca/live/

- Type qualifiers and hierarchy. Each type system has at least one type qualifier and one type hierarchy. You need to define the subtyping relationship among different type qualifiers and the default type qualifier in the type system.
- Interface to the compiler. This is the entry point of the type checker. The command-line options and the sub checkers can be indicated here.
- Type rules. Type rules represent type system semantics. The type checker will report errors to developers when type rules are broken.
- Type introduction rules. For each of the type locations, type introduction rules indicate the suitable type annotations to it. Developers can create a tree annotator to express their own introduction rules.
- Dataflow rules. The Checker Framework performs flow-sensitive type refinement following the provided dataflow rules. Chapter 4.1 gives an example of creating and implementing dataflow rules.

The Checker Framework's manual[2] provides plenty of documentation that guides developers to get started and dive into. The Checker Framework's community is also responsive and friendly.

[2] https://checkerframework.org/manual/

Chapter 3

Crypto Checker Type System

Cryptographic APIs are hard to understand and use for developers who are not cryptographers [36], which causes significant security vulnerabilities. This thesis 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 [24]. 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`

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

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. Currently, there are no tools that enforce the security rules for the Android Keystore.

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");
```

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 [16]. 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 [2]. Similarly, hash functions MD5, MD4, SHA-1, and the mode of operation ECB, 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.

Many companies have security policies that specify what cryptographic algorithms and providers must be used. Human code reviewers can easily miss the use of incorrect algorithms and providers. Many static analysis tools have hard-coded security rules, which makes them useless for this situation. Tools which support custom rules require writing rules in some rule language, which is an additional learning effort and source of possible errors.

In this chapter, 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 of allowed 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 [30]. Our work provides security rules for the Android Keystore [5] and supports provider checking. Furthermore, to handle the case that a program reads cryptographic parameters from property files, we designed a Property File Handler that performs type refinements for Java property file APIs. The Property File Handler will be discussed in 4.1.

The Crypto Checker is a pluggable type system [8] built on the Checker Framework [17]. To the best of our knowledge, this is the first open-source static analysis tool providing a comprehensive solution to ensuring the correct usage of both cryptographic algorithms and providers at compile time. The source code is available on GitHub: https://github.com/vehiloco/crypto-checker.

Overall, our contributions are:

- a type system to enforce the correct usage of algorithms and providers at compile time (Section 3.1),
- an implementation of the type system for Java (Section 3.2.1),
- a flexible way to define custom security rules (Section 3.2.2),
- well-defined security rules for Java and Android applications (Section 3.2.3),
- a property file handler performing type refinement (Section 4.1), and
- case studies on 2 million LOC of Java and Android applications (Section 5.1).

Section 6 reviews related work and Section 7.1 concludes.

3.1 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 3.1.1 describes the type qualifiers of the

type system. Section 3.1.2 discusses the qualifier hierarchy. Section 3.1.3 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.

3.1.1 Type Qualifiers

Type qualifiers are used to specify properties that cannot be expressed by the standard type system [8, 23]. Java's type annotation syntax can be used to represent type qualifiers [44]. 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 3.1.2).

Type qualifiers `@AllowedAlgorithms` and `@AllowedProviders` record the permitted algorithms and providers, in a `String[] value` annotation type element. 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. In addition, algorithm names in the Java Cryptography Architecture (JCA) are case-insensitive [28]. Our type system also supports that—for example, `HMACSHA224` is equal to `HmacSHA224` in the type system.

Type qualifier `@StringVal` expresses permitted String values, again as an `String[] value` annotation type element. Most commonly, `@StringVal` is automatically determined for String literals in the program. This constant value propagation is provided by the Constant Value Checker [12] 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.

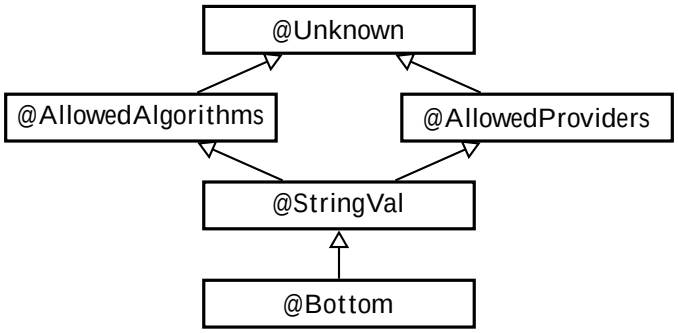

Figure 3.1: The basic qualifier hierarchy of the Crypto Checker's type system. Arrows represent the subtyping relationships between types. For example, `@Unknown` is the supertype of `@AllowedAlgorithms` and `@AllowedProviders`.

The type qualifiers are only used for `String` types in the program; uses on other types are forbidden. For example, `@Unknown int x` is an illegal use of the type annotation.

3.1.2 Qualifier Hierarchy

These type qualifiers form an easy-to-understand qualifier hierarchy (type qualifier lattice), which is shown in Figure 3.1. `@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 [13]. Figure 3.2 also illustrates the subtyping rules.

These subtyping rules are conservative but sound: 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.

$$\frac{A \subseteq B}{@AllowedProviders(A) <: @AllowedProviders(B)}$$

$$\frac{A \subseteq B}{@AllowedAlgorithms(A) <: @AllowedAlgorithms(B)}$$

Figure 3.2: The subtyping rules for two `@AllowedAlgorithms` or two `@AllowedProviders`.

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 value matches the regular expressions in `@AllowedAlgorithms` or `@AllowedProviders`. As the arguments to `@StringVal` do not use regular expression, the type system simply needs to check whether the regular expression matches the string value.

3.1.3 Type Rules for Assignment and Pseudo-assignment

The subtyping rules from Section 3.1.2 are used wherever the underlying programming language 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)"})`:

```
@AllowedAlgorithms({"SHA-(256|512)"}) String algo;
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 [17].

3.2 Crypto Checker

We present the Crypto Checker, a pluggable type system for Java, which implements the type system described in Section 3.1 and enforces the correct usage of algorithms and providers at compile time. In this section, we discuss the Crypto Checker's implementation and features in detail.

3.2.1 Implementation

The Crypto Checker is built using the Checker Framework [17], which helps developers create pluggable type checkers. The Crypto Checker is written with only 376 non-blank,

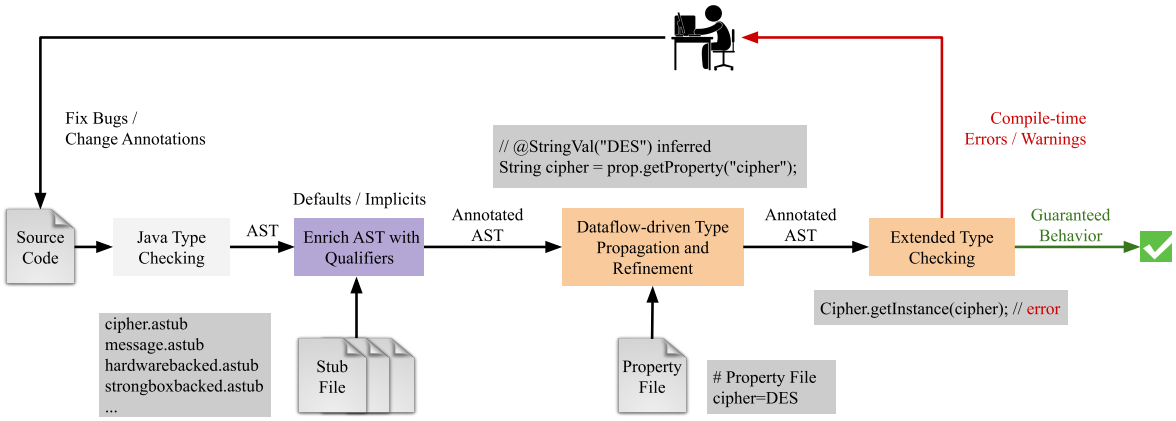

Figure 3.3: Workflow of the Crypto Checker.
Workflow of the Crypto Checker based on Checker Framework pluggable type checking.

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). Figure 3.3 illustrates the basic workflow of the Crypto Checker based on the Checker Framework:

- The Java source code for a project is the input.
- The Java compiler compiles the source code and performs its standard syntactic and semantic checks. The resulting attributed abstract syntax tree (AST) is the input to the Checker Framework.
- In the next step, specifications of cryptographic rules are read from stub files and incorporated into the AST.
- The Crypto Checker performs flow-sensitive type refinement (also called local type inference) on the annotated AST to enhance available type information. The Crypto Checker infers all compile-time constant values, e.g., the expression `"Hmac"`

`+ (high_security() ? "MD5" : "SHA512")` will result in type `@StringVal({"HmacMD5", "HmacSHA512"})`.

- To handle reading cryptographic parameters from property files, the Crypto Checker also refines types by reading property files at compile time (Section 4.1).
- Finally, the Crypto Checker traverses the annotated AST and applies the type rules, i.e., the subtyping rules from Section 3.1. If any forbidden algorithms or providers are found, the Crypto Checker will report them to the user. Otherwise, it gives a guarantee that no invalid algorithms or providers will be used at run time.

The Crytpo Checker integrates into the normal Java build process and produces error messages in the standard Java format.

3.2.2 Flexibility of Defining Cryptographic Rules

Several tools exist to detect Java cryptographic API misuses, e.g., Coverity [15], Eclipse CogniCrypt [31], and CryptoGuard [39]. But many of these tools use hard-coded cryptographic rules, thus forcing users to follow only these rules. However, organizations have preferences about particular cryptographic API usage; for instance, they intend to use one or more specific algorithms or providers, or they have different security rules than others. In such cases, the hard-coded security rules make these tools useless. For the tools that support writing custom rules, the learning curve is always steep: users are always requested to write a checker or detector or learn a specific template model [22,32,40], which is time-consuming.

With the Crypto Checker, users can write their custom rules by adding annotations to the method signatures in stub files and supply the stub files while running the checker so that the checker can read and apply the rules. Annotations make adding rules user-friendly. An example of a stub file which restricts to use `RSA` and `EC` as the algorithm and `AndroidKeyStore` as the provider for `java.security.KeyPairGenerator#getInstance(String algorithm, String provider)` is as follow:

```
# example.astub
package java.security;

class KeyPairGenerator {
  static KeyPairGenerator getInstance(
    @AllowedAlgorithms({"RSA", "EC"}) String a0,
```

```
    @AllowedProviders({"AndroidKeyStore"}) String a1);
}
```

Accepting the user-defined stub files that indicate the allowable algorithms and providers, the Crypto Checker extensively enhances the flexibility and convenience of choosing users' preferable security rules.

3.2.3 Enforced Cryptographic Rules

The Crypto Checker implements several default security rules extracted from security and static analysis papers [2, 9, 19, 37] and the cryptographic API documentation [4, 5, 27, 35] to meet the developers' requirements. 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 on GitHub: https://github.com/vehiloco/crypto-checker#stub-files.

Chapter 4

Type System Extensions

In this chapter, the thesis presents two type system extensions, for the Crypto Checker and other type systems.

4.1 Property File Handler

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 [3] and Apache's commons-cipher [6]. We added special handling for this pattern 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. The following code demonstrates the typical loading process from the property file `config.properties`:

```
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 [40], 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 checking 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 checking because only the default value, which is a safe algorithm, is checked.

Considering this dilemma, we designed the Property File Handler that performs type refinement for `Properties` types. The Property File Handler reads the property file, if the file can be determined at compile time, and then refines the return type of `prop.getProperty(String key)` and `prop.getProperty(String key, String defaultValue)` to use a `@StringVal` with the value from the property file. The Property File Handler is a type system extension, which can collaborate with any other type system. In this chapter, we used the Crypto Checker as an example. Note that the compile time property files must match run time property files, otherwise the Property File Handler may not works properly as expected.

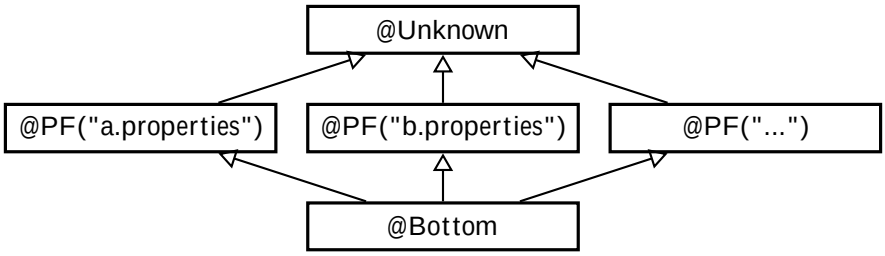

Figure 4.1: The qualifier hierarchy of the Property File Handler's type system.

4.1.1 Type System

In this section, we present the type qualifiers, the type hierarchy, and the type refinement rules of the Property File Handler.

A simple type hierarchy with three type qualifiers is needed for the Property File Handler:

- `@Unknown` is the top qualifier in the type hierarchy. This is also the default qualifier in the hierarchy.
- `@PropertyFile` indicates the information of a property file. If an object has type `@PropertyFile("a.property")`, then this object has type `java.io.InputStream` or `java.util.Properties` which loads the property file: `a.property`.
- `@Bottom` is the bottom qualifier in the type hierarchy in order to make the lattice complete. Developers usually do not need to work with this qualifier.

Figure 4.1 shows the qualifier hierarchy of the Property File Handler's type system.

We use the following property file and the code snippet as an example to illustrate the type refinement rules:

```
# 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 Property File Handler propagates the property file's information to the return type of the method invocation. Thus, `inputStream` will have type `@PropertyFile("a.properties")`.
- When loading properties from the input stream, the Property File Handler keeps propagating the `@PropertyFile("a.properties")` type to the receiver, i.e., the object `prop` itself.
- Finally, when `prop.getProperty()` is called, the Property File Handler 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 (with the Property File Handler enabled) 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's add `"AES"` as default value to the example. It is now equal to:

```
@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 Property File Handler to help users correct their configuration files.

4.1.2 Implementation

The Property File Handler is built on the Checker Framework with 583 newly-added lines of Java code. The source code is published on Github: https://github.com/opprop/checker-framework/pull/120.

To enable the Property File Handler with your checker, command-line option `handlePropertyFiles` should be passed to the Checker Framework:

```
javacheck -processor org.checkerframework.checker.crypto.CryptoChecker -Astubs=
    cipher.astub -AhandlePropertyFile A.java
```

The Property File Handler is designed to be conservative to keep the Crypto Checker and other type systems sound: if the checker can not open and read a property file successfully, for whatever reason, 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.

4.2 Network Request Checker

Previous work has shown that there are a large number of applications requiring information access for which they do not necessarily need. This increases the risk of information leakage. A key step to leak information is sending network requests, e.g., HTTP and socket requests without the user's consent. Thus, we want to create a security tool that can detect possible network requests on the source code level at compile time. This tool should support the popular third-party network request libraries and have the flexibility to add more libraries in the future by users. It also should be able to integrate with the build system and CI tools easily so that developers can use it without further study.

In this Chapter, we present the Network Request Checker, a simple type system extension that finds all the possible network request APIs usage when building the applications. The underlying principle is monitoring all the method invocations and constructor invocations in the program. It is implemented as a type system extension so that it can work with any arbitrary type system. It can also be seen as a stand-alone type system depending on developers' needs. The source code of the Network Request Checker is published on Github: https://github.com/vehiloco/network-request-checker.

4.2.1 Type System

Unlike the Crypto Checker we present in Section 3.2, the Network Request Checker uses a declaration annotation as its core annotation. A type annotation can be written on any use of a type, while a declaration annotation can be written on any class or method declaration. Normally, the users of the Network Request Checker do not need to write annotations in the source code manually. When developers want to use the Network Request Checker as a stand-alone type system, to make the type hierarchy complete, the Network Request

Checker provides a dummy annotation which has no practical effects. Overall, the Network Request Checker has two annotations: `@NetworkRequest` and `@NetworkRequestDummy`:

- `@NetworkRequest(String[] value)` is a declaration annotation. When it is found in the source code, the type system should report it to users as a possible network request. To get and report all the interesting information of the network request, the `String[] value` type element is used to indicate the kind of the information the type system will get. For instance, for Java network API `URL(String protocol, String host, int port, String file)`, we need to annotate it with annotation `@NetworkRequest("PROTOCOL", "HOST", "PORT", "FILE")` in the stub file, so that the Network Request Checker knows what each parameter means in the URL API. Then the type system can combine and report all the information properly.
- `@NetworkRequestDummy` is a dummy type qualifier, there is no specific type rules applying to it.

4.2.2 Implementation

The Network Request Checker is built on the Checker Framework with 254 non-blank, non-comment lines of Java code. It is implemented as a type system extension for Java. Like the Crypto Checker, the Network Request Checker performs modular type checking. The Network Request Checker can also leverage the Property File Handler to perform flow-sensitive type refinement.

The Network Request Checker currently supports popular third-party network request libraries via stub files:

- `httpclient4.astub` supports HttpGet in Apache HttpClient library.
- `springframework.astub` supports RestTemplate in Java Spring Framework.

It also allows developers to add other third-party libraries by putting the library method signatures into the new stub files. In Chapter 5.2, we discuss the case studies with the Network Request Checker using the above stub files.

Chapter 5

Experiments

In this chapter, we discuss the case studies with the Crypto Checker and the Network Request Checker. We did not perform case studies with the stand-alone Property File Handler. The Property File Handler is used by the Crypto Checker and the Network Request Checker in their case studies to improve precision.

5.1 Case Studies with the Crypto Checker

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 are found from GitHub 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 also used 65 test cases from the CRYPTOAPI-BENCH [2] 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.

We ran the Property File Handler with the Crypto Checker to improve performance. For each standard Java project, we supplied two stub files (`cipher.astub` and `message-digest.astub`). The results indicate insecure uses of `Cipher` and `MessageDigest` (see Section 5.1.1). 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 5.1.2). Section 5.1.3 discusses the Crypto Checker's performance with CRYPTOAPI-BENCH.

Choosing cryptographic algorithms and providers depends on many factors, e.g., the runtime environment, the hardware resources, and the communication protocols. Sometimes unsafe algorithms or providers are used intentionally and the errors reported by the Crypto Checker could be considered false positives. We view such reports as useful and valuable for improving the security of applications. Firstly, by reporting all the potential insecure uses, developers can double-check whether unsafe algorithms or providers are used deliberately or are actually misused. When developers think it is acceptable to continue using these algorithms or providers, they can suppress the corresponding errors. Secondly, it is good to add documentation to the purposely insecure uses to make the project more maintainable. Whenever a developer suppresses an error, they should document the reason for using an insecure algorithm to allow auditing the application and reasoning about possible security issues. Overall, as any sound static analysis, the Crypto Checker may introduce false positives, but it is still helpful to handle these errors. In our evaluation, we have not encountered false positives that are caused by a weakness of our static analysis.

5.1.1 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. These 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 5.1.1.1) and 8 unsafe public methods (see Section 5.1.1.2). Besides the 62 defects, no false positives are found by the Crypto Checker. The evaluation results of the 15 projects are listed in Table 5.1. The repository URLs of these 18 projects and the annotated versions of 3 projects are listed in Table 5.2.

5.1.1.1 Insecure Cryptography

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

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 5.1: 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 5.1.1.1). Unsafe Public Methods is the number of public methods that use cryptographic APIs (Section 5.1.1.2). C1 + C2 + C3 + C4 may not be equal to Total Defects, since some code violates multiple rules simultaneously.

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 5.2: Repository URLs of Java applications listed in Table 5.1.

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

Table 5.3: Repository URLs of Android applications.

- 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 and insecure cipher. We only count each of these misuses as one defect.

In Section 3.2.3, we presented the references used for the definition of the Crypto Checker rules in stub files. We use these same references to establish the four categories of violations.

The results for each app in each of the four categories are summarized in Table 5.1, and we discussion 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 [27, 45]. 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 not encouraged. The reason is that a default mode of operation and padding schema will 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 [42], which is secure. Therefore `Cipher.getInstance("RSA")` is secure, which is used frequently in real-world applications.

Category 3 The insecure ciphers should be forbidden when creating cipher objects since insecure ciphers such as DES, Blowfish, and RC4 could make brute-force attacks possible. The following example from class `DESUtils` in 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 insecure hash function and insecure cipher simultaneously.

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

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

For project smart, one developer opened an issue [25] to point out that one insecure hash function, MD5, is used in this project. The Crypto Checker reports that MD5 is indeed used by the project and that such a use is insecure. For another project flutter_secure_storage (one of the three projects that have no cryptographic misuses, which is listed in Table 5.2), developers have opened an issue [26] to question 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 secure.

5.1.1.2 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. Here we demonstrate one example from class `Crypto` in the project rapidoid:

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

```
}
```

In this case, calling `Cipher.getInstance(transformation)` could raise security issues, since there is no guarantee for the correct use of cryptographic algorithms. 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"}) String transformation`.

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. Annotating the parameter specifies the allowed algorithms for trusted users and allows the Crypto Checker to ensure valid usage from trusted code. 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.

5.1.2 Android Keystore Case Study

In Android, key material can be exposed unintentionally, which may cause information leakage. 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) [5, 14]. 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 [5].

However, developers might not use `AndroidKeystore` as the provider even though their applications require the high security and reliability. If they do not specify the provider, it is not guaranteed that `AndroidKeyStore` will be chosen as the provider; therefore, the key material may be unsafe. In another case, developers possibly use the legacy or general key algorithms that are not supported by the `AndroidKeyStore` provider. For example, HmacMD5 is not supported by `AndroidKeyStore` but can be used with other providers.

Aiming to handle the cases mentioned above, we supplied the Crypto Checker with

`hardwarebacked.astub` to find three underlying 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 used the Crypto Checker to test 14 security-sensitive Android applications listed in Table 5.3 without having to add any annotations. 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 further 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 violation among these 14 projects, which indicates that all the algorithms were used correctly.

The Crypto Checker can give a guarantee of 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. Consequently, the Crypto Checker forces developers to take account of the Android Keystore system to use cryptography in Android applications correctly.

5.1.3 CRYPTOAPI-BENCH Case Study

The benchmark CRYPTOAPI-BENCH [2] 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[1] 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

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

to indicate the specifications. With the 79 added annotations, the Crypto Checker can handle the field-sensitive cases. For path-sensitive cases, the Crypto Checker issues nine cryptographic misuses. One example is shown below:

```
method2(2); // 2 is passed as the value of choice

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, we believe that this method is unsafe and should not be trusted. Developers could 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 positives. 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.

5.2 Case Studies with the Network Request Checker

For the Network Request Checker, we performed a simple case study with 6 real-world Java and Android applications to illustrate the functionality of the Network Request Checker. In total, the Network Request Checker found 17 potential network requests in 5 out of 6 Java and Android applications. For each of the applications, we provided three stub files: `jdk8.astub` for Java native Network Request APIs, `httpclient4.astub` for Apache HTTPClient Network Request APIs, and `springframework.astub` for Spring Framework Network Request APIs. The experimental result is shown in Table 5.4. After manual inspection, we confirm that the Network Request Checker finds all the target APIs' usage.

Applications	NCNB LOC	Manual Annotations	Total Network Requests	C1	C2	C3
MITREid Connect	45k	0	2	0	1	1
PFLockScreen-Android	38k	0	0	0	0	0
wigle-wifi-wardriving	35k	0	2	2	0	0
termux-api	6k	0	1	1	0	0
elasticsearch-analysis-ik	4k	0	1	0	1	0
Eclipse Lyo Server	3k	0	11	11	0	0
Totals	131k	0	17	14	2	1

Table 5.4: Case study statistics for Java and Android 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 (Java native Network Request APIs), C2 (Apache HTTPClient Network Request APIs), C3 (Spring Framework Network Request APIs) are the three categories of network request APIs.

Chapter 6

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 forbidden algorithms and providers, this will be our main focus. As the Network Request Checker is a type system extension, we will not discuss the Network Request Checker's related work in this chapter. However, the Property File Handler's related work is discussed because the Crypto Checker leverages the Property File Handler to enhance Java Properties types.

6.1 Algorithm Checking

The AWS Crypto Policy Compliance Checker (AWS Checker for short) is a type checker built on the Checker Framework [17], which checks if there are any usages of weak cipher algorithms in Java applications. Part of our work, weak algorithms 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 [39] 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 5.1.3, 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: it does not believe the outside world of the method or class which is currently being checked, which may find more potential vulnerabilities, such as exposures of public methods (Section 5.1.1.2).

Error Prone [22] is a famous 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 algorithms and providers' rules easily: 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. Moreover, the Crypto Checker supplies developers with a type system, which can improve the code style, program understanding, and documentation.

There are also some other static analysis tools supporting cryptographic algorithm checking, such as Coverity [15], SonarSource [40], SpotBugs [41], and LGTM [33]. 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.

6.2 Provider Checking

We are not aware of any other tools that support provider checking. Correspondingly, there are no such tools supplying security rules for the `AndroidKeyStore` provider.

6.3 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. However, developers must ensure that the compile-time configuration files match the runtime configuration files. Otherwise, the Property File Handler cannot work as expected.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

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. To the best of our knowledge, this is the first open-source tool that checks cryptographic providers, analyses property files, and enforces security rules for the Android Keystore system. 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.

We also present two type system extensions, the Property File Handler and the Network Request Checker, to enhance the Crypto Checker and other type systems' performance and help developers avoid potential information leakage. The Property File Handler tries to load applications' property files at compile time and then performs path-sensitive type refinement to gain more valuable compile-time constant values. The Network Request Checker uses declaration annotations to check all the usage of network APIs. The Network Request Checker can also be seen as a stand-alone type system depending on developers' needs.

7.2 Future Work

In this thesis, we present our work on Java cryptographic APIs misuse. Clearly, the work in this domain is not finished yet. The following ideas can be considered as future work:

- Check more cryptographic APIs misuses. Currently, the Crypto Checker focuses on the weak or unsupported cryptographic algorithms and providers. More different security rules can be checked in the future, e.g., non-random initialization vector for CBC encryption, HTTP, constant salts for PBE, constant encryption keys, and constant seeds for the secure random number generator.

- Infer Crypto Checker annotations automatically. To reduce the developers' burden of adding annotation manually, whole-program type inference should be investigated as solution [18, 47].

- Adapt the Network Request Checker to also support allow/deny lists like the Crypto Checker. The current Network Request Checker reports all the possible network requests, which will produce a lot of information. Allow/deny lists can help developers find the potential malicious behaviors faster.

- To handle more complicated cryptographic APIs misuse and indirect information leakage, object capability can be a comprehensive and general solution: immutability, readonly, and allowing to copy and pass the reference but not allow to dereference. This idea can be started based on PICO, the ownership and immutability framework for typechecking and inference [43].

References

[1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *Cybersecurity Development (SecDev)*, pages 49–61. IEEE, 2019.

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

[3] An implementation of the Git version control system in pure Java. URL: https://github.com/eclipse/jgit.

[4] Android Keystore Provider. URL: https://developer.android.com/training/articles/keystore#SupportedAlgorithms.

[5] Android Keystore System. URL: https://developer.android.com/training/articles/keystore#HardwareSecurityModule.

[6] Apache Commons Crypto. URL: https://github.com/apache/commons-crypto.

[7] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *Automated Software Engineering (ASE)*, November 2015.

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

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

[10] C.Z. Chen and W. Dietl. Don't miss the end: Preventing unsafe end-of-file comparisons. In *NASA Formal Methods*, 2018.

[11] Class KeyGenerator. URL: https://docs.oracle.com/javase/8/docs/api/javax/crypto/KeyGenerator.html.

[12] Constant Value Checker. URL: https://checkerframework.org/manual/#constant-value-checker.

[13] Constant Value Checker Qualifier Hierarchy. URL: https://checkerframework.org/manual/#fig-value-hierarchy.

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

[15] Coverity Static Application Security Testing (SAST). URL: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.

[16] CWE-327: Use of a Broken or Risky Cryptographic Algorithm. URL: https://cwe.mitre.org/data/definitions/327.html.

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

[18] W. Dietl, M. D. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2011.

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

[20] ElementType (Java Platform SE 8). URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html.

[21] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Computer and Communications Security (CCS)*, November 2014.

[22] Error Prone Bug Pattern: InsecureCryptoUsage. URL: https://errorprone.info/bugpattern/InsecureCryptoUsage.

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

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

[25] Issue: Cryptographic API misuse detected. URL: https://github.com/a466350665/smart/issues/47.

[26] Issue: ECB Mode is Insecure. URL: https://github.com/mogol/flutter_secure_storage/issues/60.

[27] Java Cryptography Architecture (JCA) Reference Guide. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

[28] Java™ Cryptography Architecture Standard Algorithm Name Documentation. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html.

[29] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D Ernst. Lightweight verification of array indexing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 3–14, 2018.

[30] Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. Continuous compliance. In *Automated Software Engineering (ASE)*, Melbourne, Australia, September 2020.

[31] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[32] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[33] LGTM: Continuous security analysis. URL: https://lgtm.com/.

[34] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[35] MSC61-J. Do not use insecure or weak cryptographic algorithms. URL: https://wiki.sei.cmu.edu/confluence/display/java/MSC61-J.+Do+not+use+insecure+or+weak+cryptographic+algorithms.

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

[37] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto API rules from code changes. In *Programming Language Design and Implementation (PLDI)*, pages 450–464, 2018.

[38] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

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

[40] SonarSource builds world-class products for Code Quality & Security. URL: https://www.sonarsource.com/.

[41] SpotBugs: Find bugs in Java Programs. URL: https://spotbugs.github.io/.

[42] Stack Overflow: Java - Default RSA padding in SUN JCE/Oracle JCE. URL: https://stackoverflow.com/questions/21066902/default-rsa-padding-in-sun-jce-oracle-jce.

[43] Mier Ta. Context sensitive typechecking and inference: Ownership and immutability, 2018. URL: http://hdl.handle.net/10012/13185.

[44] Type Annotations (JSR 308). URL: https://jcp.org/en/jsr/detail?id=308.

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

[46] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38 – 94, 1994.

[47] Tongtong Xiang, Jeff Y Luo, and Werner Dietl. Precise inference of expressive units of measurement types. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.