A Lightweight Type System with Uniqueness and Typestates for the Java Cryptography API

by

Shuchen Liu

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, 2023

© Shuchen Liu 2023

Author's Declaration

I hereby declare that I am the sole author of this 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.

Abstract

Java cryptographic APIs facilitate building secure applications, but not all developers have strong cryptographic knowledge to use these APIs correctly. Several studies have shown that misuses of those cryptographic APIs may cause significant security vulnerabilities, compromising the integrity of applications and exposing sensitive data. Hence, it is an important problem to design methodologies and techniques, which can guide developers in building secure applications with minimum effort, and that are accessible to non-experts in cryptography.

In this thesis, we present a methodology that reasons about the correct usage of Java cryptographic APIs with types, specifically targeting to cryptographic applications. Our type system combines aliasing control and the abstraction of object states into typestates, allowing users to express a set of user-defined disciplines on the use of cryptographic APIs and invariants on variable usage. More specifically, we employ the typestate automaton to depict typestates within our type system, and we control aliases by applying the principle of uniqueness to sensitive data.

We mainly focus on the usage of initialization vectors. An initialization vector is a binary vector used as the input to initialize the state for the encryption of a plaintext block sequence. Randomization and uniqueness are crucial to an initialization vector. Failing to maintain a unique initialization vector for encryption can compromise confidentiality. Encrypting the same plaintext with the same initialization vector always yields the same ciphertext, thereby simplifying the attacker's task of guessing the cipher pattern.

To address this problem practically, we implement our approach as a pluggable type system on top of the EISOP Checker Framework. To minimize the cryptographic expertise required by application developers looking to incorporate secure computing concepts into their software, our approach allows cryptographic experts to plug in the protocols into the system. In this setting, developers merely need to provide minimal annotations on sensitive data—requiring little cryptographic knowledge.

We also evaluated our work by performing experiments over one benchmark and 7 real-world Java projects from Github. We found that 6 out 7 projects have security issues. In summary, we found 12 misuses in initialization vectors.

Acknowledgements

I would like to express my profound gratitude to my supervisor, Professor Werner Dietl. His support throughout my Master's program has been invaluable, not only making this thesis possible but also guiding me step by step through this research field. I am profoundly grateful for his willingness to provide me an oasis, a space to explore, learn, and grow in my research.

Next, my sincere thanks go to Professor Arie Gurfinkel, whose kind instructions significantly impacted my work. I am also immensely thankful to Professor Mahesh Tripunitara for agreeing to be my committee member for my thesis and providing me with insightful feedback.

Last but not least, I want to express my heartfelt appreciation to Professor Yuyan Bao, who co-supervised my work, as well as to my peers: Haifeng, Zhiping, Piyush, and Aosen. It has been an absolute pleasure working alongside you all, and the experience has been all the richer for your contributions.

Dedication

This thesis is dedicated to my husband, Xusheng Chen. Encountering him during these two years was akin to a dream coming true. I am profoundly grateful for his companionship and affection. In ways that words can scarcely capture, he has infused my life with joy and support.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

1 Introduction 1

2 Background 4

2.1 Java Cryptographic API Standard 4

2.1.1 Initialization Vector (IV) and Class IvParameterSpec 4

2.1.2 Class SecureRandom 5

2.2 The EISOP Checker Framework 6

2.3 Aliasing 7

2.4 Typestates 7

3 Uniqueness and Typestate System 9
3.1 Informal Introduction . . . 9
3.1.1 Tracking Uniqueness with Type Qualifiers . . . 9
3.2 Type Qualifiers and Type Hierarchy . . . 11
3.3 Program Verification via Typestate . . . 12
3.4 Formalization . . . 12
3.4.1 Syntax . . . 12
3.4.2 Auxiliary Functions . . . 14
3.4.3 Well-formedness Definitions . . . 15
3.4.4 Static Typing . . . 17

4 Implementation 21
4.1 Type Refinement . . . 21
4.2 Default Type Annotation . . . 21
4.3 Typestates . . . 22
4.3.1 Typestate Automaton . . . 22
4.3.2 Typestate Validation . . . 23

5 Case Study 24
5.1 Findings in CRYPTOAPI-BENCH . . . 25
5.2 Findings in Real-World Projects . . . 26
5.2.1 Using Predictable IVs . . . 26
5.2.2 Reused IVs . . . 27
5.3 False Positives . . . 27

6 Related Work 30
6.1 Aliasing Control . . . 30
6.2 Typestate Systems . . . 31
6.3 Vulnerability Detection Tools . . . 32
6.4 Static Verification Systems . . . 33

7 Conclusions **34**

References **35**

List of Figures

2.1 The automaton of a file object in Java. 7

3.1 IvParameterSpec initialization protocol specified by state machine. 10

3.2 Type hierarchy. 11

3.3 Abstract syntax. 13

3.4 Auxiliary functions extracted from a given class table. 15

3.5 Type rules for expressions. 17

3.6 Type rules for statements (part 1). 18

3.7 Type rules for statements (part 2). 19

4.1 Typestate automaton in the EISOP Checker Framework. 22

4.2 Typestates validation in the EISOP Checker Framework. 23

List of Tables

5.1 Benchmark test cases. 25
5.2 Our evaluation on practical projects from Github. 28

Chapter 1

Introduction

Java Cryptography Architecture (JCA) [67] is a cryptographic framework provided by the standard Java platform. It offers cryptographic application programming interfaces (APIs) to build secure applications. The design of those cryptographic APIs intends to abstract away the implementation details, so that developers without cryptographic expertise can build cryptographic applications with the security guarantees provided by the APIs. However, several studies showed that developers may misuse cryptographic APIs due to the deficiency in understanding of security API usage [4, 57, 72], complex API designs [4, 63, 72], the lack of cybersecurity training [57, 72], and insecure/misleading suggestions in Stack Overflow [5, 57, 72]. Misuses of cryptographic APIs may lead to runtime exceptions or introduce significant security vulnerabilities, e.g., exposed secret keys and passwords, predictable random numbers, use of insecure cryptographic APIs and vulnerable certificate verification [29, 30, 34, 57, 72, 73].

Taking the building of an Android app as an example, the Android Keystore system [1] allows users to store sensitive data (e.g., keys and passwords) in the secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)) of the Android device to make it more difficult to extract from the device. In this scenario, users can store and retrieve sensitive information by invoking cryptographic operations encapsulated by those APIs. Misusing those cryptographic APIs, e.g., invoking cryptographic methods in the wrong order, may still leak the sensitive data [76]. Thus, the problem of developing cryptographic applications in realistic scenarios with promising security guarantees needs to be solved.

To address the problem, recently, there have been continuous efforts in detecting cryptographic misuse in Java, e.g., statically detection tools (CryptoLint [29], CRYSL [52],

FixDroid [64], MalloDroi [30], CryptoGuard [72]) and dynamic code screening tools (SMV-Hunter [78], and AndroSSL [33]). However, we note that these approaches leave remaining challenges that weaken the promising security guarantees provided by the APIs.

Challenge 1: Lack of Security Guarantees. Static program analysis [29] examines and detects vulnerability without executing the code. Existing approaches can detect the misuse of common flaws and provide assistance to developers with writing and maintaining secure code. Those approaches can only mitigate the problem, but there are no soundness guarantees. Dynamic program analysis requires one to execute a program and spend (usually) manual efforts to trigger and detect specific misuse symptoms at runtime. Thus, they may be limited in their coverage and are not scalable.

Challenge 2: Lack of Usability. Verification-based tools, e.g., the Microsoft Crypto Verification Kit [14] and Murφ [58], perform verification on models written in a verification language. To apply those approach to verify software written in industrial programming language, like Java, those approaches require a compilation pipeline that translates source and target languages back and forth. In addition, they require heavyweight annotations written by experts in formal methods.

Challenge 3: Lack of Modularity. Java is widely used in building cryptographic applications and systems, and is an object-oriented (OO) programming language. Thus, Java applications are generally extensible. Without a modular methodology, the security guarantees that a type-checked (verified) program provides may be weakened once it is extended. However, OO techniques make modular reasoning difficult, due to non-localized features such as subtyping, inheritance and dynamic dispatch, references (or pointers), callbacks, and globally accessible (static) fields and methods.

Challenge 4: Lack of Detection of Reused Initialization Vectors (IV). The existing detection tools we mentioned, which are capable of identifying specific vulnerabilities such as constant nonce and improper cipher mode, fall short when it comes to identifying the reuse of initialization vectors—especially those stored in fields. However, the initialization vectors should be used at most once [2]. Any violation of this rule can result in identical ciphertext for the same plaintext, thus leading to potential security vulnerabilities, as pointed out in [76].

With the goal of designing a methodology and type system for modular reasoning about cryptographic applications with lightweight annotation overhead, we present a type system

that supports typestate-style reasoning and controls aliasing through uniqueness. Typestate, as a refinement of the concept of type [80], can express a set of user-defined disciplines on the uses of cryptographic APIs and invariants on variables, and can facilitate developers' exploration into the space of secure application and validate the correct uses with limited effort. Generally, sound typestate verification requires precise information about aliasing. Globally controlling aliasing may be over-restricted, e.g., disallowing common programming patterns. Thus, our approach combines aliasing control and abstractions of object states into a typestate system. It only restricts aliasing on local variables that refer to cryptographic objects, which involve secure-critical operations that require a strict order on method calls. Based on our empirical study in Chapter 5, our approach fits well with the abstractions that cryptographic APIs provide.

To minimize the cryptographic expertise required by application developers looking to incorporate secure computing concepts into their software, our framework allows cryptographic experts to plug in the protocols into the system.

In summary, this thesis makes the following contributions:

- We design a modular verification methodology that requires lightweight annotations and disentangles the constraints of programming models from the cryptographic APIs usage specifications. Following our approach, developers can write secure applications by following the protocols provided by cryptographic experts.
- We formalize a typestate system that combines the aliasing information and program states abstraction into typestate automaton (Chapter 3). It allows users to write secure programs with flexible programming patterns without weakening the promising security guarantees.
- We implemented the system in the EISOP Checker Framework [3] which is an evolution of the Checker Framework [24, 69] as an extension (Chapter 4). The implementation enriches the type refinement algorithm with typestate automaton.
- We performed case studies on 7 practical projects and each project is actively maintained. We also found 12 vulnerabilities of initialization vectors and filed issues to some of those projects (Chapter 5).

The rest of our thesis is structured as follows. Chapter 2 provides essential background information to enhance understanding of our work. Chapter 3 introduces our formal type system. Chapter 4 explains key concepts integral to our implementation. In Chapter 5, we present case studies and evaluations of our approach. Chapter 6 discusses the related work, followed by Chapter 7, where we offer conclusions and outline future work.

Chapter 2

Background

This chapter provides background information to facilitate a better understanding of our work. Section 2.1 presents the Java Cryptographic API and discusses the potential challenges that developers might encounter when building secure applications. Section 2.2 introduces the EISOP Checker Framework, forming the basis of our implementation. Section 2.3 discusses the issues of aliasing and techniques for controlling aliasing. Section 2.4 explores verification based on typestates. The last two sections highlight the key techniques employed in our work.

2.1 Java Cryptographic API Standard

Java security libraries (JCA and JSSE) provide a set of APIs for building secure applications, e.g.,including cryptography, public key infrastructure, authentication, secure communication, and access control [67]. In our work, we focus on the classes of SecureRandom which is used to generate random criteria and IvParameterSpec which can specify the initialization vector (IV). The subsequent content discusses the standards and the scenarios where misuse might occur.

2.1.1 Initialization Vector (IV) and Class IvParameterSpec

In cryptography, an IV acts as input to a cryptographic primitive being used to provide the initial state. An IV is used to add entropy to ciphertexts of encryption. Thus, it must have enough randomness and be properly generated. In Java Cryptographic applications, a byte array often serves as the IV to create an IvParameterSpec object.

To make sure an IV is unpredictable and unique, the following lists the recommended way to use an IV:

1. An IV should be initialized from an object of SecureRandom (Section 2.1.2).
2. Once an IV is initialized, its elements should not be modified or copied.
3. An IV can be used to initialize **at most one** IvParameterSpec object.

Thus, we propose the following scenarios to enforce correct use of IvParameterSpec.

1. An object of IvParameterSpec can be used for encryption at most once.
2. An object of IvParameterSpec may be used for encryption and decryption. Therefore, the precondition of the constructor is `true`, which implies the parameter iv could be tracked or untracked.
3. Before encryption, the type of an object is tracked. After encryption, the type of an object is untracked, which prevents the same object being used for another encryption.
4. SecureRandom.nextBytes() must be invoked on the byte array before encryption.
5. Tracked objects should not be aliased.

Listing 2.1 shows the improper usage which violates the recommended standards. Specifically, in line 6, the initialized IV is aliased and the same array is passed into the IvParamSpec again.

Listing 2.1: The improper usage of IvParameterSpec.

```
1      byte [] bytesIV = new [16];
2      SecureRandom secureRandom = new SecureRandom();
3      secureRandom.nextBytes(bytesIV);
4      IvParameterSpec ivSpec = new IvParameterSpec(bytesIV);
5      byte [] newBytesIv;
6      newBytesIv = bytesIV;
7      IvParameterSpec ivSpec2 = new IvParameterSpec(newBytesIv);
```

2.1.2 Class SecureRandom

This class provides a cryptographically strong random number generator (RNG). SecureRandom must yield random output which is hard to predict. Therefore, any seed material passed to a SecureRandom object must be unpredictable. In our scenario, we must use a unpredictable seed generated by SecureRandom.nextBytes() to initialize the initialization vector.

2.2 The EISOP Checker Framework

The EISOP Checker Framework [3] is an evolution of the Checker Framework [24, 69]. The EISOP Checker Framework is a pluggable types system, which supports adding pluggable type systems to the Java language in a backward-compatible way. In general, existing built-in static type systems are already able to detect and prevent errors before we run the programs. Unfortunately, the standard type system cannot detect and prevent enough errors in practice. For example, NullPointerException frequently appears in practice and is challenging to eliminate entirely by the existing built-in type system. As a result, programmers often debug their code manually, even though they can do this automatically.

The EISOP Checker Framework enriches the standard type system by allowing both type system designers and developers to define their new type qualifiers, type hierarchies, type rules, etc. Thus, they can build and run their type systems as a plugin to the standard Java compiler without extra efforts and interruptions to the existing components.

Furthermore, the EISOP Checker Framework is expressive, which is a key factor in its real-world application. The EISOP Checker Framework is easy to learn; for example, users such as developers do not need to deeply understand the concepts of the type system, they just need to annotate their program properly.

To implement a user-defined type system using the EISOP Checker Framework, we typically need the following steps.

- Defining Type Qualifiers and Type Hierarchy. Type qualifiers can represent types in the type system. In the EISOP Checker Framework, type qualifiers can be written in Java annotations and the syntax of type qualifiers is simple: the type annotations should be put before the basic type, for example, `@Nullable Object o`.

- Designing a Control Flow Graph Algorithm. The EISOP Checker Framework already enforces the basic control flow graph algorithms, and users can define their algorithms based on these. By overriding the transfer function, users can apply their own algorithms.

- Writing type rules. Once the type qualifiers and type hierarchy have been defined, users can then write their type rules. As mentioned above, the EISOP Checker Framework also involves the basic type rules for the users, e.g., subtyping rule. Thus, users only need to write their specific rules.

It is worth mentioning that the EISOP Checker Framework has a relatively light way to implement typestates by introducing pre- and post-conditions. Figure 4.1 shows how to

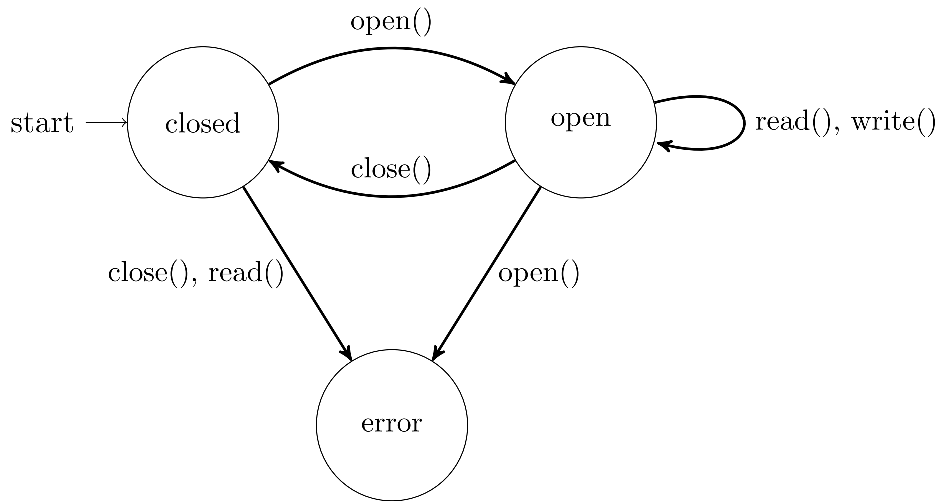

Figure 2.1: The automaton of a file object in Java.

define post-conditions in the EISOP Checker Framework. We will discuss how to represent typestate automaton later in Section 4.3.

2.3 Aliasing

Aliasing has been the key technical challenge in Java program verification. It allows different variable names to refer to the same resource. To make static verification sound, the verifier must be aware of the alias relationship by which the program changes the object's state of interest. Modern verification techniques either rely on advanced verification logics (e.g., separation logic [66, 75] and region logic [8, 9, 10]), whose semantics provide for the modification of memory locations, or expressive type systems to control aliasing (see Section 6.1 for detailed discussion). Applying those approaches to solve our problem requires experts with a deep understanding of the verification methodology and significant human efforts.

2.4 Typestates

In computer programming, typestates can be defined as specific behaviors that expressions are allowed to perform. For example, in Java, we can perform read operations on a File object only if this object is in the "open" state. Conversely, we cannot perform read and write operations on a "closed" file. In essence, type states indicate which operations, such as methods, can be called on the objects.

The concept of type states has been formally defined by [80, 47], where the typestate can be represented as a finite type state automaton $A = (\Sigma, S, s_0, \delta, e)$. Σ represents the operations that can be performed on this type; S is a set of all the type states of this type, and s_0 is the initial state. δ determines the state transitions. Lastly, e represents the error states or non-accepting states.

Figure 2.1 shows how typestates work by providing a brief example. In this example, the operations include "read()", "write()", "open()" and "closed()", and the state set S consists of closed, open, and error. Any invalid operation will lead to an error state.

Implementing type states in a type system can be challenging: developers usually need an additional protocol file that specifies all states and related transitioning operations. In our work, we introduce a new approach to implementing type states in the type system by leveraging the pre/post-conditions feature of the EISOP Checker Framework. We will discuss this implementation later in Section 4.3.

Chapter 3

Uniqueness and Typestate System

This section provides a detailed explanation of our typestate system and presents the formal type rules. Section 3.1 offers an informal introduction to our design by our examples; Section 3.2 introduces our type qualifiers and type hierarchy; Section 3.3 provides an overview of the typestate automaton for our case. Section 3.4 presents a formalization of our type system, including the syntax and type rules.

3.1 Informal Introduction

To make sure the security guarantees provided by Java cryptographic APIs are not weakened via incorrect usage, a developer must obey the rules for properly calling an object's method, e.g., calling them in an allowed order, obeying the preconditions on the methods' argument, and data being used properly, for instance, used at most once.

Taking the IV as an example, as explained in Section 2.1.1, an IV needs to be initialized from an object of SecureRandom, cannot be modified before being used to initialize an IvParameterSpec object, and can not be used for this purpose anymore. This protocol is illustrated in Figure 3.1 as a simple abstract state machine combined with aliasing information.

3.1.1 Tracking Uniqueness with Type Qualifiers

As discussed in Section 2.3, it is challenging to enforce such a protocol without controlling aliasing. We introduce a type qualifier @Unique to indicate that there is only one variable that can refer to its referent. For example, in the following code snippet, variable `bytesIV` is annotated as @Unique at its declaring site.

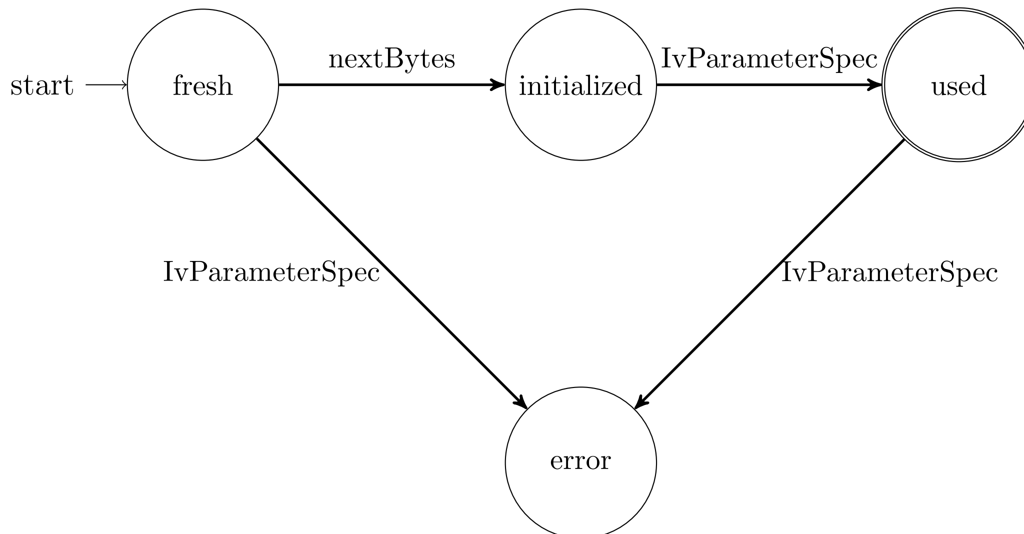

Figure 3.1: IvParameterSpec initialization protocol specified by state machine.

```
1     byte @Unique [] bytesIV = new [16];
2     SecureRandom secureRandom = new SecureRandom();
3     secureRandom.nextBytes(bytesIV);
4     IvParameterSpec ivSpec = new IvParameterSpec(bytesIV); //: ok
```

This uniqueness property is enforced up until it is used (line 4), and is automatically inferred based on the protocol (documented in a separate file (see Section 4.3)).

Assignment. An assignment will disable the use of `bytesIV` as shown in the followings:

```
1     byte @Unique [] bytesIV = new [16];
2     SecureRandom secureRandom = new SecureRandom();
3     secureRandom.nextBytes(bytesIV);
4     byte [] bytesIV2 = bytesIV;
5     IvParameterSpec ivSpec = new IvParameterSpec(bytesIV);  //: wrong
6     IvParameterSpec ivSpec = new IvParameterSpec(bytesIV2); //: ok
```

Here, the type inference assigns the unique qualifier to `byteIV2`. Now `bytesIV` cannot be used for the initialization (at line 5). One can use `bytesIV2` for initialization (at line 6).

Functional Abstraction. Methods are annotated with pre- and post-conditions. At the caller site, the type system checks the actual parameter against the pre-condition, assumes the type annotation in the post-condition after the method call. If a unique variable is mentioned in the pre-condition, and is used (e.g., via assignment) in the function body, then the client cannot use it anymore.

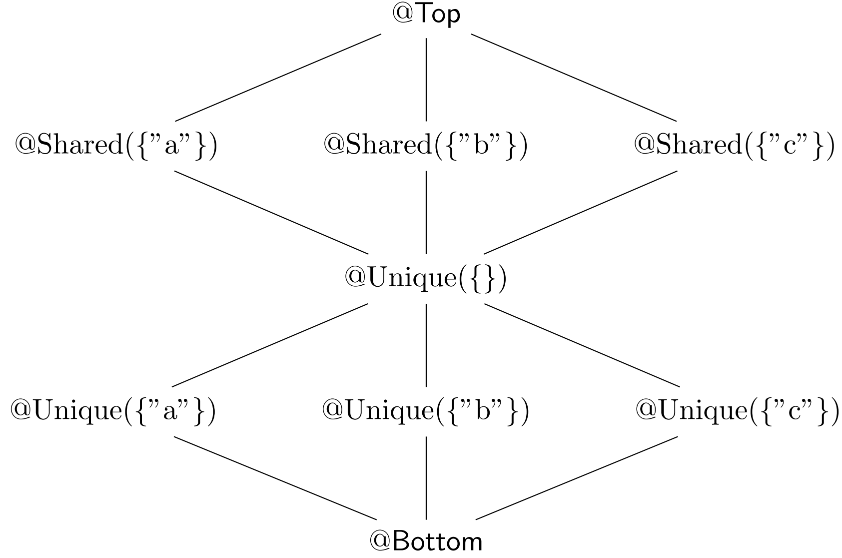

Figure 3.2: Type hierarchy.

To check the method body, we assume the type of formal parameters, and check the method body. At the end of the method body, we check the type of variables against its post-condition.

3.2 Type Qualifiers and Type Hierarchy

We have four type qualifiers: @Unique {}, @Shared {}, @Top and @Bottom, which represent our uniqueness property, denoted by the symbol ρ, in the subsequent Section 3.4.1. Moreover, we associate our type qualifiers with curly braces {} to hold related typestates.

Figure 3.2 depicts our type hierarchy. @Top and @Bottom are straightforward in the hierarchy. @Shared is the super type of @Unique, indicating variables associated with this type can be used repeatedly. Moreover, a @Shared type can hold all typestates, enabling the recording of all states a variable undergoes. Therefore, regardless of what typestate the @Unique holds, @Shared always remains its supertype. @Unique qualifiers with different typestates, e.g., "a", "b" and "c" do not have relationships with each other. This is because we aim to constrain the typestate transfer in between the uniques, i.e., the typestate can be only changed by the operations in the defined automaton, rather than by assignments.

In our implementation, our default type qualifier is @Top, which is @Shared{} in our implementation, we will discuss it in Section 4.2.

3.3 Program Verification via Typestate

Formally, typestate can be represented as a finite type state automaton A = (Γ, S, S_0, δ, e). In our work, we perform the typestate verification on the initizalization vectors, i.e., initialization vectors have their typestates at certain points and the typestate will change according to the invocation of the methods called on them. Γ denotes the operations in the automaton. In this paper, some methods which are called on the security nonce are operations, e.g., the SecureRandom.nextbytes and IvParameterSpec. The set of typestate **S** can be defined by users, as well as the initial state S_0. δ determines the state transitions, i.e., from a pre-state to a post-state. When an initialization with the invalid state is being called on a specific operation, the automaton will reach an error state e. In our work, we prevent the misuses of initialization vectors by verifying the automaton does not enter an error state, otherwise a type error will be reported.

Figure 3.1 illustrates the typestate automaton in our work. The set of states S consists of "{}", "initialized" and "used". When an byte array for an initialization vector is created, an initial state "{}" is attached to it automatically. To generate the initialization vector, this array must be randomized by calling the RNG method SecureRandom.nextByte() on it, and its state will be initialized afterwards. Once the array is passed in IvParameterSpec to generate the IV, the associated state becomes used, indicating this variable cannot be used again. Any byte array without the state initialized will lead to the error state when calling IvParameterSpec on it.

3.4 Formalization

This section presents our formal type system. Section 3.4.1 introduces the syntax of our type system. To streamline the presentation, Section 3.4.2 introduces several auxiliary functions. Section 3.4.3 shows the well-formed definitions. Section 3.4.4 presents the type rules for our expressions and statements respectively.

3.4.1 Syntax

The syntax of a program is defined in Figure 3.3. We assume all variable names are globally unique. The notation $\overline{x}$ means zero or more occurrences of x, and the notation $[x]$ means that x is optional. We use P to denote a program; C and D to denote a class and its superclass, K to denote a constructor; F denotes field declarations and f denotes field name respectively. The uppercase M denotes the method declaration while lowercase m is a method's name. The notations x and y range over variable names; e means expressions;

$$
\begin{array}{rcl}
P & ::= & \overline{\mathit{ClassDecl}} \\
\text{ClassDecl} & ::= & \text{class } C \text{ extends } D\{\ \overline{F}\ K\ \overline{M}\ \} \\
F & ::= & f:\ QT; \\
K & ::= & C\ (\overline{g:AT_g};\ \overline{f:AT_f})\ \ :\text{ret}:QT_C:\ \{\ \text{super}(\overline{g});\ \overline{\text{this}.f := f};\ \text{ret} := \mathit{this}\ \} \\
M & ::= & \text{def } m(\text{this}:AT_c,\ \overline{x:AT_x})\ \ :\text{ret}:QT_r:\ \{S\} \\
e & ::= & x \mid c \mid x.f \mid e_1 \oplus e_2 \\
S & ::= & \text{skip} \mid x := y \mid x := y.f \mid x.f := y \mid x := x_1.m(\overline{x_2}) \\
 & \mid & \text{var } x:QT := e \ \textsf{in}\ S \mid \text{var } x:QT := \text{new } \rho\ C(\overline{y})\ \textsf{in}\ S \mid \textsf{if}\ e\ \text{then}\ S_1\ \textsf{then}\ S_2 \\
 & \mid & \textsf{while}\ e\ \textsf{do}\ S \mid S_1;\ S_2 \\
T & ::= & \text{Int} \mid \text{C} \\
\rho & ::= & \textsf{unique}\ \text{TS} \mid \textsf{shared}\ \text{TS}_a \mid \top \mid \bot \\
\text{TS} & ::= & \text{TS}_u \mid \text{TS}_a \\
c & \in & \mathbb{Z} \\
QT & ::= & \text{Int} \mid \rho\ \text{C} \\
AT & ::= & \text{Int} \mid \rho \rightarrow \rho'\ \text{C} \\
\text{TS}_u & ::= & \text{unique typestate identifier} \\
\text{TS}_a & ::= & \text{accumulation typestate identifier} \\
\oplus & ::= & + \mid - \\
\Gamma & ::= & \varnothing \mid \Gamma, x:QT \mid \Gamma, x.f:QT \\
\Phi & ::= & \varnothing \mid \Phi, \overline{p:\text{TS}_{pre}} \xrightarrow{\text{(C,m)}} \overline{\text{TS}_{pst}}
\end{array}
$$

Figure 3.3: Abstract syntax.

S ranges over statements; T ranges over pre-types; ρ combines aliasing information with typestate; c ranges over constants; QT denotes qualified types. AT ranges over arrow type; TS ranges over typestates. We use Γ to denote a type environment and Φ to denote a typestate table.

A program P consists of a sequence of class declarations. A class declaration has an identifier C, a super class identifier D, a sequence of field declarations $\overline{F}$, a single constructor K and a sequence of method declarations $\overline{M}$. A field declaration is a field identifier f associated with its qualified type QT. A constructor declaration K shows how to initialize an object of class C. It takes $\overline{g}$ and $\overline{f}$ with their qualified types as its parameters, and has a return type for ret. In the body of the constructor,the superclass constructor is called to initialize the fields declared in the superclass, followed by the assignments initializing its own fields, and finally this variable is assigned to the return variable ret.

A method declaration M consists of its identifier m, its parameters, its return type, and its body. The parameters include its receiver this and the formal method parameters

$\overline{x}$ with their types, and the method body contains a set of statements S.

The syntax of the expression e and statement S are straightforward. The expression e contains a variable and a constant, field access and binary operations which are denoted by the symbol $\oplus$. The statement S consists of skip, assignment, field read, field update, method invocation, local variable block, object initialization, if and while statements, and sequential statements.

Our pre-types T are integers and the class names. Our type system tracks qualified types $\texttt{QT}$ that annotates a class name with a qualifier ρ to track aliasing (i.e., unique and shared) and an object's state, TS. Our qualifier ρ also includes $\top$ and $\bot$, which were discussed in Section 3.2. Our typestate TS is either TS_u or TS_a, indicating the general typestate and accumulation typestate respectively. An arrow type, $\texttt{AT}$, is used to annotate constructor and method declarations, and is in the form of $(\rho \rightarrow \rho' \ C)$. It can express objects state changes from ρ in the pre-state to ρ' in the post-state. It serves as an abbreviation of $\rho \ C \rightarrow \rho' \ C$.

A type environment Γ maps from variables or field read expressions to their qualified types. The typestate table Φ specifies the protocls that are expressed via typestates transitions for object constructor and methods. We write $\Phi(C, m, \overline{\text{TS}_{pre}}) = \overline{\text{TS}_{pst}}$ to mean retrieving the typestate in the post-state from the typestate table Φ for a given class C and a method m.

For convenience, we use the notation $\bot$ to represent the uniqueness type associated with the class type C, denoted as $\bot \ C$.

3.4.2 Auxiliary Functions

To streamline the presentation, Figure 3.4 defines several auxiliary functions.

TSLOOKUP gets the typestate from $\rho \ C$ if ρ is unique TS or shared TS. The field type lookup FIELDTYPE returns the fields with their types of a class C. METHODTYPE LOOKUP returns the parameter types and the return type for a given method m in a class C. CONSTRUCTORTYPE returns the parameters types and the return type of a constructor of a class C. Note we use the symbol $\uplus$ to denote the disjoint operation in its premises.

In addition, for the sake of convenience, we also have some straightforward auxiliary functions.

The notation $QT_r[\texttt{PT}(QT_r) \mapsto \texttt{PT}(QT_z)]$ substitutes the pre-type of a qualified type with a pre-type, e.g., $(\rho \ C)[C \mapsto \texttt{PT}(\rho' \ D)] = \rho \ D$.

$$\frac{\rho = \textsf{unique}\ \mathrm{TS} \vee \textsf{shared}\ \mathrm{TS}}{\mathrm{TSLookup}(\rho\ C)\ =\ \mathrm{TS}}\ (\textsc{typestate lookup})$$

$$\frac{\textsf{class}\ C\ \textsf{extends}\ D\ \{\ K\ \overline{F}\ \overline{M}\}}{\mathrm{fieldType}(C)\ =\ \overline{f : QT}}\ (\textsc{field type lookup})$$

$$\frac{\textsf{class}\ C\ \textsf{extends}\ D\ \{\ K\ \overline{F}\ \overline{M}\ \} \qquad \mathrm{def}\ m(\mathrm{this} : AT_c,\ \overline{x : AT_x})\ : \mathrm{ret} : QT_r :\ \{S\} \in \overline{M}}{\mathrm{methodType(m, C)} = (AT_c, \overline{AT_x}) \rightarrow QT_r}\ (\textsc{methodtype lookup})$$

$$\frac{\begin{array}{c}\textsf{class}\ C\ \textsf{extends}\ D\ \{\ K\ \overline{F}\ \overline{M}\ \} \\ K ::= C\ (\overline{g : AT_g};\ \overline{f : AT_f})\ : \mathrm{ret} : QT_C\ : \{\ \mathrm{super}(\overline{g});\ \overline{\mathrm{this}.f := f};\ \mathrm{ret} := \mathrm{this}\} \\ \overline{p : AT_p} ::= \overline{g : AT_g}\ \uplus\ \overline{f : AT_f}\end{array}}{\mathrm{constructorType(C)} = \overline{p : AT_p} \rightarrow QT_c}\ (\textsc{constructortype lookup})$$

$$\begin{array}{rcl} \mathrm{dom(Int)} & = & \mathrm{Int} \\ \mathrm{domTS}(\rho \rightarrow \rho'\ C) & = & TS_{pre} \\ \mathrm{dom}(\rho \rightarrow \rho'\ C) & = & \rho\ C \\ \mathrm{codom(Int)} & = & \mathrm{Int} \\ \mathrm{codom}(\rho \rightarrow \rho'\ C) & = & \rho'\ C \\ \mathrm{codomTS}(\rho \rightarrow \rho'\ C) & = & TS_{pst} \end{array} \qquad \begin{array}{rcl} PT & : & (AT + QT) \rightarrow T \\ \mathtt{PT}(\mathrm{Int}) & = & \mathrm{Int} \\ \mathtt{PT}(\rho\ C) & = & C \\ \mathtt{PT}(\rho \rightarrow \rho'\ C) & = & C \\ \mathrm{TS}(\rho\ C) & = & \mathrm{TS} \end{array}$$

Figure 3.4: Auxiliary functions extracted from a given class table.

3.4.3 Well-formedness Definitions

The notation $\vdash_{\mathrm{wf}}$ is a judgment for well-formed definitions. We assume there is a class table extracted from program P.

The class C is well-formed if it is in the class table. QT is well-formed if it is Int. If QT is $\rho\ C$, it is well-formed if the class type C is well-formed, and its associated typestate TS is in the typestate table Φ.

$$\frac{C \in \text{Class Table}}{\vdash_{\mathrm{wf}} C}\ (\textsc{wf-c}) \qquad \frac{QT = \rho\ C \qquad \vdash_{\mathrm{wf}} C \qquad \mathrm{TSLookup}(\rho\ C) = \mathrm{TS} \quad \mathrm{TS} \in \Phi}{\vdash_{\mathrm{wf}} QT}\ (\textsc{wf-qt-}\rho C) \qquad \frac{QT = \mathrm{Int}}{\vdash_{\mathrm{wf}} QT}\ (\textsc{wf-qt-int})$$

The type environment Γ is well-formed if it can map the variables in it to their types QT and QT is well-formed, and if a field $x.f$ is in Γ, its receiver x is also in Γ and has the

type $\rho\ C$, and the field belongs to the fields of C.

$$\frac{\begin{array}{l}(\textsc{wf-}\Gamma\ )\\ x \in \Gamma \Rightarrow \Gamma(x) = QT \wedge\ \vdash_{\text{wf}} QT \qquad x.f \in \Gamma \Rightarrow x \in \Gamma \wedge \Gamma(x) = \rho\ C \wedge f \in \text{fieldType}(C)\end{array}}{\vdash_{\text{wf}} \Gamma}$$

The field declaration is well-formed if the filed f is in the fields of its class C and has the well-formed type QT where QT is TS_a shared.

$$\frac{\begin{array}{l}(\textsc{wf-f})\\ f \in \text{fieldType}(C) \wedge \text{fieldType}(C)(f) = QT \wedge QT = \text{TS}_a\ \mathsf{shared} \qquad \vdash_{\text{wf}} QT\end{array}}{C \vdash_{\text{wf}} f : \text{shared TS}_a;}$$

Rule WF-CLASS shows that a class declaration is well-formed if its constructor K, field declarations $\overline{F}$ and method declarations $\overline{M}$ are well-formed. The constructor K is well-formed if the domain of the parameters $\overline{AT_p}$ and return type are not $\perp$. Other constraints are similar to Featherweight Java [43].

$$\frac{\begin{array}{c}(\textsc{wf-class})\\ K ::= C\ (\overline{g : AT_g};\ \overline{f : AT_f})\ : \text{ret} : QT_C :\ \{\ \text{super}(\overline{g});\ \overline{\text{this}.f := f};\ \text{ret} := \mathit{this}\ \}\\ \text{constructorType}(C) = \overline{p : AT_p} \rightarrow QT_C\\ \text{this} : C, \text{ret} : QT_c, \overline{p : \text{dom}(AT_p)} \vdash \{\ \text{super}(\overline{g});\ \overline{\text{this}.f := f};\ \text{ret} := \mathit{this}\ \} \dashv \Gamma'\\ \overline{\text{dom}(AT_p)} \neq \perp \qquad QT_C \neq \perp \qquad \Gamma' \vdash \overline{p :\ \text{codom}(AT_p)}\\ \Gamma' \vdash \text{ret} : QT_C \qquad \text{constructorType}(D) = \overline{g : AT_D} \rightarrow QT_D \qquad \overline{\text{dom}(AT_g) <: \text{dom}(AT_D)}\\ \overline{F} = \overline{f : QT_f} \qquad C \vdash_{\text{wf}} \overline{f : QT_f} \qquad \overline{\text{dom}(AT_f) <: QT_f} \qquad C \vdash_{\text{wf}} \overline{M}\end{array}}{\vdash_{\text{wf}} \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ K\ \overline{F}\ \overline{M}\ \}}$$

Rule WF-M indicates that a method declaration is well-formed if the domain and codomain of its receiver type are not $\perp$, and the domain of the parameters' types $\overline{AT_x}$ and the return type QT_r are not $\perp$ as well. Other constrains are similar to Featherweight Java [43]. Moreover, the declared post-typestates of the receiver type and parameter types should match the states defined in the typestate table Φ.

$\boxed{\Gamma \vdash e : \rho\ T}$

$$\text{(T-C)}\quad \frac{}{\Gamma \vdash c : \text{Int}}\qquad \text{(T-VAR-C)}\quad \frac{\Gamma(x) = \rho\ C \qquad \rho \neq \bot}{\Gamma \vdash x : \rho\ C}\qquad \text{(T-VAR-INT)}\quad \frac{\Gamma(x) = \text{Int}}{\Gamma \vdash x : \text{Int}}$$

$$\text{(T-F-1)}\quad \frac{\Gamma \not\vdash x.f\ \ \Gamma \vdash x : \rho_x\ C_x \qquad (f : \mathsf{shared}\ \text{TS}_f\ C) \in \mathsf{fieldType}(C_x)}{\Gamma \vdash x.f : \mathsf{shared}\ \text{TS}_f\ C}$$

$$\text{(T-F-2)}\quad \frac{\Gamma \vdash x : \rho_x\ C_x \qquad \Gamma(x.f) = \mathsf{shared} \qquad \text{TS}_f\ C}{\Gamma \vdash x.f : \mathsf{shared}\ \text{TS}_f\ C}\qquad \text{(T-OP)}\quad \frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash\ e_2 : \text{Int}}{\Gamma \vdash e_1 \oplus e_2 : \text{Int}}$$

Figure 3.5: Type rules for expressions.

$$\text{(WF-M)}\quad \frac{\begin{array}{c} \text{this} : \text{dom}(AT_c), \text{ret} : QT_r, \overline{x : \text{dom}(AT_x)} \vdash S \dashv \Gamma' \\ PT(AT_c) = C \qquad \text{dom}(AT_c) \neq \bot \qquad \text{codom}(AT_c) \neq \bot \qquad \overline{\text{dom}(AT_x) \neq \bot} \\ QT_r \neq \bot \qquad \Gamma' \vdash \text{this} : \text{codom}(AT_c) \qquad \Gamma' \vdash \overline{x :\ \text{codom}(AT_x)} \qquad \Gamma' \vdash QT_r \\ \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ K\ \overline{F}\ \overline{M}\ \} \qquad \mathsf{methodType}(D, m) = (AT_D, \overline{AT_{Dx}}) \rightarrow\ QT_{Dr} \\ \overline{AT_{Dx}} = \overline{AT_x} \qquad QT_{Dr} = QT_r \qquad \text{codomTS}(AT_c, \overline{AT_x}) = \Phi(C, m, \text{domTS}(AT_c, \overline{AT_x})) \end{array}}{C\ \ \vdash_{\text{wf}} \text{def}\ m(\text{this} : AT_c,\ \overline{x : AT_x})\ \ : \text{ret} : QT_r :\ \ \{S\}}$$

3.4.4 Static Typing

Figure 3.5 shows our type rules of expressions. Note all expression rules are under the type environment Γ. A constant c only has the type Int. The type of a variable x is either Int or $\rho\ C$ if x has the type $\rho\ C$ in Γ and ρ is not $\bot$. A field $x.f$ always has type $\mathsf{shared}\ \text{TS}_f\ C$ if it is already in Γ, or its receiver x is in Γ and we can retrive its type from the class table indexed by the static type of x. Finally, an Int expression e_1 operating with another Int expression e_2 produces the Int type as well.

Figure 3.6 and Figure 3.7 illustrate the type rules of our statements. $\Gamma \vdash S\ \dashv \Gamma'$ means the type environment Γ becomes Γ' after the execution of the statement S.

The rules T-SKIP, T-DEREF-SHARED, T-WHILE, T-SEQ and T-VAR-S are standard and straightforward, thus, we omit the definitions of them here.

$$\boxed{\Gamma \vdash S \dashv \Gamma'}$$

(T-SKIP)

$$\Gamma \vdash \mathsf{skip} \dashv \Gamma$$

(T-ASSIGN-SHARED)

$$\frac{\Gamma \vdash x : \rho_x\ C_x \qquad \Gamma \vdash y : \mathsf{shared}\ \mathrm{TS}_y\ C_y \qquad \vdash C_y <: C_x}{\Gamma \vdash x := y \dashv (\Gamma - x.*)[x \mapsto \mathsf{shared}\ \mathrm{TS}_y\ C_x] + +[x.* \mapsto \Gamma(y.*)]}$$

(T-ASSIGN-U)

$$\frac{\Gamma \vdash x : \rho_x\ C_x \qquad \Gamma \vdash y : \mathsf{unique}\ \mathrm{TS}_y\ C_y \qquad \vdash C_y <: C_x}{\Gamma \vdash x := y \dashv (((\Gamma - x.*)[x : \mathsf{unique}\ \mathrm{TS}_y\ C_x] + +[x.* \mapsto \Gamma(y.*)]) - y.*)[y : \bot]}$$

(T-DEREF-SHARED)

$$\frac{\Gamma \vdash x : \rho_x\ C_x \qquad \Gamma \vdash y.f : \mathsf{shared}\ \mathrm{TS}_f\ C_f \qquad \vdash C_f <: C_x}{\Gamma \vdash x := \mathsf{y}.f \dashv (\Gamma - x.*)[x : \mathsf{shared}\ \mathrm{TS}_f\ C_x]}$$

(T-UPD-SHARED)

$$\frac{\Gamma \vdash x.f : \mathsf{shared}\ \mathrm{TS}_f\ C_f \qquad \Gamma(x) = \mathsf{shared} \qquad \Gamma \vdash y : \mathsf{shared}\ \mathrm{TS}_y\ C_y \qquad \vdash C_y <: C_f}{\Gamma \vdash x.f := y \dashv \Gamma[x.f : \mathsf{shared}\ \mathrm{TS}_y\ C_f]}$$

(T-UPD-U)

$$\frac{\Gamma \vdash x.f : \mathsf{shared}\ \mathrm{TS}_f\ C_f \qquad \Gamma(x) = \mathsf{shared} \qquad \Gamma \vdash y : \mathsf{unique}\ \mathrm{TS}_y\ C_y \qquad \vdash C_y <: C_f}{\Gamma \vdash x.f := y \dashv (\Gamma[x.f \mapsto \mathsf{shared}\ \mathrm{TS}_y\ C_f] - y.*)[y : \bot]}$$

Figure 3.6: Type rules for statements (part 1).

There are two rules for assignments: T-ASSIGN-SHARED and T-ASSIGN-U. Rule T-ASSIGN-SHARED is used when the qualifier of right-hand-side is shared. It assigns the right-hand side y to the left-hand-side x, where the pre-type of y is a subtype of x. After the assignment, x becomes shared TS_y. Rule T-ASSIGN-U is used when the qualifier of right-hand-side is unique. In the case of the rhs y has type unique TS_y C_y, y's type becomes $\bot$ after the assignment, and its original type and state will be transferred to the left-hand-side.

The rules for field update (T-UPD-SHARED and T-UPD-U) are similar to those for assignments. Their rhs y transfers its typestate TS_y to the lhs $x.y$ if the class type C_y is a subtype of C_x. Similar to rule T-ASSIGN-U, rule T-UPD-U deprives the use of y with qualifier $\bot$, as the data referred by y loses its uniqueness property.

Rule T-METH specifies state transitions of arguments, which is similar to linear logic (see discussion in Section 4.3). In the pre-state, the typestates of arguments $\overline{y}$ can not be

$\boxed{\Gamma \vdash S \dashv \Gamma'}$

$$\text{(T-METH)}\quad \frac{\begin{array}{c}\Gamma \vdash z : QT_z \qquad \Gamma \vdash x : QT_x \qquad \Gamma \vdash \overline{y : QT_y} \\ \mathsf{methodType}(C_x, m) = (AT_c, \overline{AT_p}) \to QT_r \qquad \vdash \overline{QT_y <: \mathrm{dom}(AT_p)} \qquad \vdash QT_r <: QT_z \\ \Gamma' = (\Gamma - z.*)[x : \mathrm{codom}(AT_c)][\overline{y : \mathrm{codom}(AT_p)}][z : QT_r[\mathtt{PT}(QT_r) \mapsto \mathtt{PT}(QT_z)] \\ Y = \{y_i \in \overline{y} \mid \Gamma'(y_i) = \bot\} \qquad \Gamma'' = \Gamma' - \bigcup_{y_i \in Y} y_i.*\end{array}}{\Gamma \vdash z := x.m(\overline{y}) \dashv \Gamma''}$$

$$\text{(T-IF)}\quad \frac{\Gamma \vdash e : \ \mathsf{Int} \qquad \Gamma \vdash S_1 \dashv \Gamma_1 \qquad \Gamma_1 \vdash S_2 \dashv \Gamma_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ S_1 \mathsf{else}\ S_2 \dashv \Gamma_1 \bowtie \Gamma_2}$$

$$\text{(T-WHILE)}\quad \frac{\Gamma \vdash e : \ \mathsf{Int} \qquad \Gamma \vdash S \dashv \Gamma'}{\Gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ S \dashv \Gamma'}$$

$$\text{(T-SEQ)}\quad \frac{\Gamma \vdash S_1 \dashv \Gamma_1 \qquad \Gamma_1 \vdash S_2 \dashv \Gamma'}{\Gamma \vdash S_1\ S_2 \dashv \Gamma'}$$

$$\text{(T-VAR-S)}\quad \frac{\Gamma \vdash e : QT_e \qquad QT_e <: QT_x \qquad \Gamma, x : QT_e \vdash S \dashv \Gamma'}{\Gamma \vdash \mathsf{var}\ x : QT_x := e\ \mathsf{in}\ S \dashv \Gamma' - x - x.*}$$

$$\text{(T-NEW)}\quad \frac{\begin{array}{c}\Gamma \vdash \overline{y : QT_y} \qquad \mathsf{constructorType}(C) = \overline{p : AT_p} \to QT_c \qquad \vdash QT_c <: \rho\ C \\ \vdash \rho\ C <: QT_x \qquad \vdash \overline{\mathrm{dom}(QT_y) <: \mathrm{dom}(AT_p)} \qquad \Gamma' = \Gamma[\overline{y : \mathrm{codom}(AT_p)}][x \mapsto \rho\ C_x] \\ C_x = PT(QT_x) \qquad Y = \{y_i \in \overline{y} \mid \Gamma'(y_i) = \bot\} \qquad \Gamma'' = \Gamma' - \bigcup_{y_i \in Y} y_i.* \qquad \Gamma'' \vdash S \dashv \Gamma'''\end{array}}{\Gamma \vdash \text{var}\ x : QT_x := \text{new}\ \rho\ C(\overline{y})\ \text{in}\ S \dashv \Gamma''' - x.* - x}$$

Figure 3.7: Type rules for statements (part 2).

$\bot$ indicating they are allowed to access the data. This is enforced as an invariant in our system, i.e., T-VAR-C. The type rule retrieves the method's type ($(AT_c, \overline{AT_p}) \to QT_r$) with respect to the static type QT_x of its receiver x in the class table via the auxiliary function `methodType`. Then we type-check the arguments $\overline{y}$'s type against the method's declared type $\overline{\mathrm{dom}(AT_p)}$. In the post-state, we assume the typestates of arguments against the method's declared type, i.e., Γ is updated with $\overline{y : \mathrm{codom}(AT_p)}$. Similarly to the receiver's typestate, as we treat the receiver as an explicit parameter. The lhs's typestate is updated with respect to the method's return type's typestate, i.e., given a return value's typestate QT_r, we substitute the pre-type of a qualified $(\rho\ C)[C \mapsto \mathtt{PT}(\rho'\ D)] = \rho\ D$, which is defined by the notation $QT_r[\mathtt{PT}(QT_r) \mapsto \mathtt{PT}(QT_z)]$. Finally, an argument will be removed from the environment if it is $\bot$ after the invocation. In this way, the invariant of our type system is

maintained.

The resulting typing environment, Γ, of an if-statement is defined by the operator $\bowtie$ on the two branch's typing environments, Γ_1 and Γ_2, written as $\Gamma_1 \bowtie \Gamma_2$. The domain of Γ is the intersection of the domains of Γ_1 and Γ_2 because we do not consider local variables declared in either branch. For each element, x, in Γ, x's type is the least upper bound of $\Gamma_1(x)$ and $\Gamma_2(x)$ if both of their types are not $\perp$, otherwise, it is their greatest upper bound. The $\bowtie$ operator is formally defined as below:

$$\Gamma_1 \bowtie \Gamma_2 = \begin{cases} (\Gamma_1(x) \sqcup \Gamma_2(x)), (\Gamma_1 - x) \bowtie (\Gamma_2 - x) & \text{if } \Gamma_1(x) \neq \perp \wedge \Gamma_2(x) \neq \perp \\ (\Gamma_1(x) \sqcap \Gamma_2(x)), (\Gamma_1 - x) \bowtie (\Gamma_2 - x) & \text{if } \Gamma_1(x) = \perp \ \vee \ \Gamma_2(x) = \perp \\ (\Gamma_1 - x) \bowtie \Gamma_2 & x \in \Gamma_1 \wedge x \notin \Gamma_2 \\ \Gamma_1 \bowtie (\Gamma_2 - x) & x \in \Gamma_2 \wedge x \notin \Gamma_1 \\ \varnothing & \Gamma_1 = \varnothing \wedge \Gamma_2 = \varnothing \end{cases}$$

where $\Gamma(x)$ means that variable x is defined in Γ.

Rule T-NEW illustrates how to initialize an object. In pre-state, the arguments $\overline{y}$ have type QT_y and QT_y cannot be $\perp$. We retrieve the constructor type as $\overline{p : AT_p} \rightarrow \ QT_c$ denoting the parameters $\overline{p}$ with the return type QT_c. Then we type-check the arguments $\overline{y}$'s types, the receiver type x's type and the new object declaration type $\rho\ C$ against the constructor declared type.

In the post-state, the arguments $\overline{y}$'s type are updated to $\overline{\text{codom}(AT)}$ and the receiver x's type becomes $\rho\ C_x$, where C_x is extracted from QT_x. Similar to the rule of T-METH, an argument will be removed from the environment if it is $\perp$ after the initialization.

Chapter 4

Implementation

This section details the implementation of our work.[1] As a stand-alone pluggable type system built on top of the EISOP Checker Framework, our implementation comprises 697 lines of code spread across five classes. We introduce the type refinement in Section 4.1, the default type in Section 4.2 and typestates in Section 4.3.

4.1 Type Refinement

In our checker, we adapted the type rules to align with our specific requirements, leveraging the inherent flexibility of the EISOP Checker Framework. To accommodate our type rules, we made two significant changes in our implementation, specifically concerning assignments and if-conditions. In EISOP Checker Framework, only the type of left-hand side (lhs) of an assignment will be updated, and the least-upper-bound (lub) is always used for the variables the if-conditions. Our checker introduces some adjustments. According to our type rules (see Section 3.4.4), the right-hand side (rhs)'s type will become $\bot$ if it is initially unique, and both the least upper bound and greatest lower bound are applicable to the if-statement, depending on the types of the variables.

4.2 Default Type Annotation

We designated $\top$, i.e., shared($\{\}$), as our default type qualifier because other types, i.e., unique and $\bot$ are overly sensitive. unique limits a variable to be used at most once, and variables with type $\bot$ are inaccessible.

[1] The code is available on https://github.com/vehiloco/linear-checker.

```
1  class SecureRandom {
2      @EnsureUnique(
3              value = "#1",
4              states = {"initialized"})
5      void nextBytes(byte @Unique({}) [] bytes);
6  }
7
8  public class IvParameterSpec {
9      @EnsureUnique(
10             value = "#1",
11             states = {"used"})
12     public IvParameterSpec(byte @Unique({"initialized"}) [] iv);
13 }
```

Figure 4.1: Typestate automaton in the EISOP Checker Framework.

Therefore, setting such default type can significantly reduce the manual effort required for program annotation, as we only need to annotate the sensitive data, like initialization vectors.

4.3 Typestates

4.3.1 Typestate Automaton

We employed the post-condition feature of the EISOP Checker Framework to represent the typestate automaton. Figure 4.1 shows an implementation example. The annotation @EnsureUnique denotes the post-condition upon method invocation or constructor initialization. In our example, we set a post-condition on the method SecureRandom.nextBytes and the constructor IvParameterSpec. In the post-condition block value = "#1" indicates that the post-condition applies to the first parameter and states = ("initialized") represents the post-state of this parameter as "initialized" after the invocation. Lastly, the annotations before the declarations of parameters show the valid pre-states.

Such an implementation can well represent the typestate automaton $A = (\Sigma, S, s_0, \delta, e)$ well. All the methods with post-condition specifications compose Σ; the set of states S includes all the states specified by the states keyword in @EnsureUnique block. In our example, we have two explicit states, *initialized* and *used*, and an implicit initial typestate s_0 $\{\}$; δ is represented by the post-condition specification @EnsureUnique. Finally, if the passed argument violates the specification, the state transitions into an error state e, resulting in a type error.

```
---
  states:
    - "initialized"
    - "used"
  operations:
    IvParameterSpec(byte[]):
        package: "javax.crypto.spec.IvParameterSpec"
        position: "#1"
        res: "void"
        before: "initialized"
        after: "used"
    nextBytes(byte[]):
        package: "java.security.SecureRandom"
        position: "#1"
        res: "void"
        before: ""
        after: "initialized"
```

Figure 4.2: Typestates validation in the EISOP Checker Framework.

4.3.2 Typestate Validation

To verify the validity of all states during type checking, we introduced a yaml file storing the correct automaton information which can be defined by the users. Each time we conduct the type-checking, we utilize this stored information to prevent any invalid typestate and method usage, such as typo errors.

Figure 4.2 shows the constrain file we integrated. In this file, we define all valid states and operations. When an operation is invoked, we first validate the signature, i.e., we ensure the signature precisely matches what we stored in the yaml. Then, we check all the states are valid by going through the states in the yaml file. Finally, we verify whether the transaction is valid by examining whether the pre- and post-states correspond to the before and after in the file.

Chapter 5

Case Study

This chapter shows our experiments and evaluations of our implementation. In Section 5.1, we evaluate the benchmark named CryptoAPI-Bench. Our experiments on real-world projects are detailed in Section 5.2. Finally, Section 5.3 discusses the false positives found during our experiments.

Our experiments were performed on some test cases generated by CryptoAPI-Bench [6], as well as some practical projects on Github.

We divided our evaluation into two parts, i.e., performing experiments on some artificial benchmark test cases generated by CryptoAPI-Bench [6] and type-checking projects on Github in the real world. CryptoAPI-Bench summarized 16 different kinds of threat models, and we mainly focused on the cases of misuse of IvparameterSpec. Running those cases ensured that our implementation was satisfying on some simple and basic usages. On the other hand, to make our evaluation more convincing, we ran our experiments on some projects on Github and obtained some more sophisticated findings. Table 5.2 shows 7 projects chosen for our experiment. All projects were actively maintained (the last merged commit was in 6 months) and widely used (more than 1,000 stars). In summary, we found 12 vulnerabilities of initialization vectors over 7 projects with only 5 manual annotations.

To reduce the manual effort for annotating source code, we carried out our experiment over two distinct rounds. In the first round we type-checked source code without manual annotations. From the output of the first step, we annotated the source files at necessary places, e.g., annotated `@Unique` on the IV parameters for the sake of encryption, and ran the second round.

Based on the results of our evaluation, we categorized the misuse of IV into two groups:

Test Cases	Number	True Positives	False Positives	False Negatives
Basic Case	2	2	0	0
Two-Interproc.	1	1	0	0
Three-Interproc.	1	2	0	0
Field Sensitive	1	1	0	0
Combined Case	1	1	0	0
Path Sensitive	1	0	1	0
Misc	2	2	0	0
Multiple Class	1	1	0	0
Total	10	9	1	0

Table 5.1: Benchmark test cases.

1. Using Predictable IVs

   Some projects use constant values or values generated by Pseudo Random Number Generator (PRNG) for IVs in the decryption mode. However, IVs should be unpredictable. Using a predictable IV may lead to a risk of exposure of the pattern of the ciphertext to attackers.

2. Re-using IVs

   Similar to using the predictable IVs, reusing the IVs also compromises the confidentiality of the encryption as same IVs always produce same ciphertext with the same plaintext and secret keys. However, this case can be tricky in the coding level as some projects store the IV in the fields when initializing an object and use this iv to encrypt messages. Although some of IVs are unpredictable, they are still being reused or have the risks of being reused since it is hard to limit a field for being used more than once.

As shown in table 5.2, 9 out of the 9 insecure uses belonged to Group 1, and only three to Group 2, representing a 75%-25% distribution in favor of Group 1.

In the rest of this chapter, we discuss our findings in Section 5.1 and Section 5.2 respectively.

5.1 Findings in CRYPTOAPI-BENCH

To ensure the accuracy of our implementation, we first evaluated our implementation by type-checking the test cases in CryptoAPI-Bench [6]. To minimize duplication of efforts, we

selected the test cases related to initialization vectors. Table 5.1 illustrates our evaluation of the initialization vector-related test cases chosen from CRYPTOAPI-BENCH. In summary, we conducted our type-checking of 10 test cases distributing in different types, e.g., basic case, interprocedure, field sensitive, etc., and all true positives were successfully identified. However, one false positive was encountered in a path-sensitive case due to our data flow analysis lacking a path-sensitive mechanism.

5.2 Findings in Real-World Projects

Table 5.2 provides a comprehensive summary of our investigation into insecure usage of Initialization Vectors (IVs) across seven real-world projects. Among these projects, we identified 12 instances of insecure IV usage. We annotated a total of five instances manually. lx-music-mobile and Apache pdfbox were the only two projects which required manual annotations, with two and one, respectively. The other projects did not require any manual annotations. Despite these findings, it's essential to note that there were nine false positives. These inaccuracies were distributed across all projects. Further discussion regarding the false positives and potential solutions will be provided inSection 5.3.

5.2.1 Using Predictable IVs

We found that 4 projects used predictable IVs. In the following content of this subsection, we discuss one representative example from the project OpenPDF.

Listing 5.1 illustrates the incorrect use of IV from the project OpenPDF. A predictable IV is generated in line 7 for encryption purpose. In line 7, an IV is initialized with a new empty array, and is passed into a function to initialize the cipher in CBC mode. Consequently, every time this method is invoked, a predictable IV is generated for the sake of encryption, simplifying the task for an attacker aiming to discern the encryption pattern.

Listing 5.1: An example of using a predictable IV.

```
1     void computeUAndUeAlg8(byte[] userPassword) throws
         GeneralSecurityException {
2         final Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
3         byte[] userSalts = IVGenerator.getIV(16);
4         userKey = new byte[48];
5         cipher.init(Cipher.ENCRYPT_MODE,
6                 new SecretKeySpec(hashAlg2B, "AES"),
7                 new IvParameterSpec(new byte[16]));
8         ueKey = cipher.update(key, 0, keySize);
9     }
```

5.2.2 Reused IVs

Listing 5.2 provides the code snippet from WxJava, where an encrypt method is implemented in CBC mode for outgoing messages. However, an aesKey is used to generate the IV as illustrated in line 17, leading to every message being encrypted with an identical IV.

Listing 5.2: An example of a reused IV.

```
1   public String encrypt(String randomStr, String plainText) {
2     ByteGroup byteCollector = new ByteGroup();
3     byte[] randomStringBytes = randomStr.getBytes(CHARSET);
4     byte[] plainTextBytes = plainText.getBytes(CHARSET);
5     byte[] bytesOfSizeInNetworkOrder = number2BytesInNetworkOrder(
        plainTextBytes.length);
6     byte[] appIdBytes = this.appidOrCorpid.getBytes(CHARSET);
7
8     //...
9     byte[] padBytes = PKCS7Encoder.encode(byteCollector.size());
10    byteCollector.addBytes(padBytes);
11
12    byte[] unencrypted = byteCollector.toBytes();
13
14    try {
15      Cipher cipher = Cipher.getInstance("AES/CBC/NoPadding");
16      SecretKeySpec keySpec = new SecretKeySpec(this.aesKey, "AES");
17      IvParameterSpec iv = new IvParameterSpec(this.aesKey, 0, 16);
18      cipher.init(Cipher.ENCRYPT_MODE, keySpec, iv);
19
20      byte[] encrypted = cipher.doFinal(unencrypted);
21
22      return BASE64.encodeToString(encrypted);
23    } catch (Exception e) {
24      throw new WxRuntimeException(e);
25    }
26  }
```

5.3 False Positives

We have 9 false positives in our evaluation. Listing 5.3 shows the common false positive which was found in all projects in table 5.2.

Listing 5.3: A false positive.

```
1     cipher.init(encryptMode ? Cipher.ENCRYPT_MODE : Cipher.
        DECRYPT_MODE, key, new IvParameterSpec(iv));
```

Project	LOC	Annotations	Unsafe Usage	G1	G2	False Positives
OpenPDF	76,668	0	4	4	0	3
WxJava	129,738	0	1	0	1	1
im-server	88,284	0	1	0	1	1
lx-music-mobile	2,488	2	1	1	0	1
eladmin	12,125	0	2	1	1	0
Apache pdfbox	169,485	2	3	3	0	1
Apache commons-compress	73,719	1	0	0	0	2
Total	552,507	5	12	9	3	9

Table 5.2: Our evaluation on practical projects from Github.

This line of code initializes a cipher according to the cipher mode. In the decryption mode, the iv is not required as unique anymore as we need to use the same IV which is used for the encryption to decrypt the message. Unfortunately, currently we do not have the mechanism to distinguish them as we only focus on the invocation of IvParameterSpec, i.e., an error will be reported once a used IV is passed into the cipher builder thought it is for the decryption purpose. To address this issue, we recommend that instead of initializing IvParameterSpec in the decryption methods, store the IV to some places after the encryption, and use this IV for decryption. Such solution can prevent calling IvParameterSpec in a decryption body.

Another kind of false positive is illustrated in listing 5.4. In this piece of code, the constructor takes an array for iv as its parameter and stores it into the fields, subsequently uses this array to initialize the IV parameter. Such behavior is harmless because the IV is only used once for encryption later. However, in our type system, the rhs iv becomes $\perp$ after the assignment in line 3 and an error is reported in line 10. To align this with our type system, we suggest initializing the IV before storing it in the field.

Listing 5.4: Another false positive.

```
1     public AES256Options(final char[] password, final byte[] salt,
         final byte[] iv, final int numCyclesPower) {
2         this.salt = salt;
3         this.iv = iv;
4         this.numCyclesPower = numCyclesPower;
5         final byte[] aesKeyBytes = AES256SHA256Decoder.sha256Password(
             password, numCyclesPower, salt);
6         final SecretKey aesKey = newSecretKeySpec(aesKeyBytes);
7
8         try {
9             cipher = Cipher.getInstance(TRANSFORMATION);
10            cipher.init(Cipher.ENCRYPT_MODE, aesKey, new
                 IvParameterSpec(iv));
11        } catch (final GeneralSecurityException
             generalSecurityException) {
12            throw new IllegalStateException(
13                "Encryption error (do you have the JCE Unlimited
                     Strength Jurisdiction Policy Files installed?)",
14                generalSecurityException
15            );
16        }
17    }
```

Chapter 6

Related Work

Our system owes much to the rich history of related language designs. In particular, it is inspired from two important lines of research: aliasing control (Section 6.1) and typestate systems (Section 6.2). We also broadly discuss other approaches in detecting vulnerabilities, i.e., static and dynamic analysis (Section 6.3) and verification techniques (Section 6.4).

6.1 Aliasing Control

Aliasing has been the central challenge in tracking the state of resources, e.g., files, sockets and dynamic allocated objects. It allows different variable names to refer to the same resource. Several systems have extended conventional type systems to tackle this problem. Linear types [82] are based on the linear logic [35, 36], and ensures values of a linear type must be used exactly once (cannot be duplicated or removed). Its linearity provides powerful reasoning capabilities and safety guarantees, such as safe in-place updates of memory locations, and correct use of external resources. Linear type systems can be used to track aliasing by prohibiting the aliasing of linear resources entirely, which may seem overly restricted. All practical linear type systems (e.g., Linear Haskell [13]) provide both linear and nonlinear types. Our system uses the `unique` qualifier to disallow aliasing. Unlike linear types, we don't demand exact use-once discipline, which is similar to affine types [81] that ensures every variable is used at most once, and uniqueness types [12, 21, 22] that ensure there is no more than one reference pointing to the resource.

Ownership type systems [65, 19] provide a mechanism to control aliasing and the access to objects. They impose a notion of encapsulation by preventing objects from being accessed outside their enclosing encapsulation boundaries, e.g., ownership-as-dominator [65]

that dictates accessing references from different contexts has to go through owners [25]. More flexible models do not restrict aliasing, but dictate that modification of objects has to pass through the object's owner, i.e., ownership-as-modifiers [27, 28] More recently, the Rust programming language [48, 44] adopts a strong ownership-based type system that globally enforces a mutable reference is unique, and a read-only reference may be shared. Ownership systems fail to enforce the principle of use-once, which is our desired outcome.

Haller and Odersky's work [38] can enforce the property of separate uniqueness for concerned variables and fields using capabilities and borrowing. While adapting their system is possible, we choose to use a lightweight system by only tracking uniqueness for local variables that involve secure sensitive data computations. The idea of disabling further use via the $\perp$ qualifier is inspired by the kill effects in reachability types [11]. It only limits the constraints of programming patterns up to a certain programming point, e.g., `byteIV` does not have to be unique once it has been used for the initialization `IvParameterSpec`.

Lanzinger et al.'s work [53] translates the type uses, which the EISOP checker framework cannot verify for correctness, into proof obligations within the Key verification tool [7], effectively reducing the false positive of a type system. Adapting this work to reduce the false positives in our type system is an interesting future work.

6.2 Typestate Systems

Typestate [80] allows one to track the set of permitted operations on an object in a given context. In general, sound typestate verification relies heavily on precise aliasing information [32]. All the existing type systems are not ready for checking Java Cryptographic API usage.

Behavioral types [42] and Mungo [50] do not allow aliasing on objects with typestates. They not only lack support for common program patterns, e.g., collections with iterators, but also do not have the ability to disable certain future uses. Fugue [23] verifies precise state transitions of NotAliased objects, and requires that a MayBeAliased object's typestate is an invariant. Our system can also perform a special case of typestate analysis when an object is shared, which is inspired by accumulation analysis [47, 46, 45].

More flexible systems control the modification of objects, instead of aliasing. [15] uses access permissions that combine typestates and object aliasing information for typestates checking. Their work uses permissions to control the modification of objects. For example, a unique read/write permission can coexist with an arbitrary number of read-only permissions to the same object. One can have unique permission when an object is created. The system is suitable for verifying general Java libraries, but does not have a mechanism to

enforce uniqueness like ours. Their methods specifications are based on linear logic, where arguments' access permissions are consumed at the call site. Our type system is designed to control the use (including read and write) of data, which is desired to guarantee the correct usage of certain Cryptographic APIs.

JATYC [59] allows one to specify and verify objects' protocols through typestates, based on the EISOP Checker Framework. The system controls the modification of objects via assertion language incorporating fractional permissions[16, 17], which ensures only one reference can be used to write into a memory location. It uses Z3 to solve the constraints generated from assertions.

6.3 Vulnerability Detection Tools

Efforts to detect cryptographic APIs have been made, including static analysis (e.g., CryptoGuard [72], CrySL [52], FixDroid [64], CryptoLint [29], Mallodroid [30]) and dynamic analysis (e.g., SMV-Hunter [78] and AndroSSL [33]). However, they cannot satisfy developers' expectations [85].

Static Analysis. CrytoGuard [72] applies inter-procedural data flow analysis with heuristics (to reduce false positives). CryptoLint [29] disassembles a raw Android binary and uses static program slicing to identify flows between cryptographic keys, initialization vectors, and similar cryptographic material and cryptographic operations. Mallodroid [30]) performs static analysis on decompiled Android apps to detect vulnerability against Man-in-the-Middle (MITM) attacks due to SSL misuses (e.g., invalid SSL certificates and no-default trust managers). FixDroid [64] leverages the static analysis techniques supported by the IntelliJ IDEA to detect obvious security mistakes, e.g., correct use of host names. Those approaches are not modular and lack soundness guarantees.

CrySL [52] performs static analysis on the specifications written in the CrySL specification language. With its specification language, cryptography experts can define CrySL rule set for the Java Cryptography Architecture, e.g., permitted method call orders, which serves a similar goal to our typestate transition rules (see Section 4.3). CrySL translates those specifications into a static analysis to find bugs; our system verifies an implementation satisfying the specification, which has potential soundness guarantees.

In summary, existing static approaches detect the existence of APIs misues in an application, but do not verify that no vulnerability exists.

Dynamic Analysis. AndroSSL [33] is a dynamic testing platform for detecting and analyzing MITM attacks. The approach requires one to manually pre-order the sequence of

actions involved in initiating a secure connection and replays them in AndroSSL experiments. SMV-Hunter [78] is a system to detect MITM vulnerability for SSL through the combination of static and dynamic analysis. Its static analysis can detect if the app overrides the X509TrustManager or HostNameVerifier interfaces, which may introduce vulnerabilities. Its dynamic analysis is guided by the information extracted from static analysis, and performs automatic UI exploration while attempting MITM attacks.

Those approaches can be used to detect vulnerabilities for existing apps where their source code may not be available. Our system provides guidance to developers to build secure applications.

6.4 Static Verification Systems

Progress has been made in automated verification of security protocols. They require a compilation pipeline that performs translation on a target language and a verification language. For example, Bhargava et al.'s work [14] verifies cryptographic implementations for TLS based on an existing tool chain that compiles F♯ code to process models in an applied pi-calculus and the state-of-the-art verifier ProVerif that analyzes models automatically. Protzenko et al.'s work [71] verifies cryptographic Web applications against specifications written in the verification-oriented programming language F∗.

Mitchell et al.'s work [58] verifies Needham-Schroeder protocol using the Murϕ verification system, where one has to code the model against the specification in the Murϕ language.

None of these tools uses an industrial-level programming language (e.g., Java, C♯, and Scala), and can be used by developers who do not have expertise in formal methods.

The Crypto Checker [84] statically detects the use of forbidden Java unsafe cryptographic algorithms and providers. It is also implemented as a plug-in of The EISOP Checker Framework [69], which is complementary to our work.

Chapter 7

Conclusions

We present a lightweight type system that combines aliasing control and typestate reasoning. It can specify and verify protocols for the correct usage of Java cryptographic APIs, so that the promising security guarantees are not weakened. We also implement it as a pluggable type system based on the EISOP Checker Framework to detect the misuse of initialization vectors in Java Cryptographic APIs. Compared with existing techniques, our approach has limited annotation overhead and can provide stronger security guarantees. Our type system is modular and is accessible to developers with little cryptographic expertise to build extensible secure applications.

We performed empirical case studies on benchmarks generated by CRYPTOAPI-BENCH [6] and open-source projects on Github. In summary, our experiment reaches the precision of 90% over the related test cases in CRYPTOAPI-BENCH, and we performed type-checking on 7 projects, finding 12 vulnerabilities.

Based on our empirical experiments, we are confident that our system is sound. In the future, we plan to seek a theoretical foundation for our system by formalizing it in the Coq theorem prover and proving the type soundness. In addition, we plan to improve the precision of our system by adapting a path-sensitive type-checker algorithm, so that false positives can be reduced. Also, employing advanced type inference algorithms (e.g., [83, 41, 40, 26]) to reduce manual annotations presents an interesting direction for future research. Finally, we are going to extend the system to verify the usage of more cryptographic APIs.

References

[1] Android keystore system. https://developer.android.com/training/articles/keystore.

[2] CWE-323: Reusing a nonce, key pair in encryption. https://cwe.mitre.org/data/definitions/323.html.

[3] Enforcing, Inferring, and Synthesizing Optional Properties (EISOP) Framework. https://eisop.github.io/.

[4] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *IEEE Symposium on Security and Privacy*, pages 154–171. IEEE Computer Society, 2017.

[5] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Symposium on Security and Privacy*, pages 289–305. IEEE Computer Society, 2016.

[6] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. Cryptoapi-bench: A comprehensive benchmark on java cryptographic API misuses. In *SecDev*, pages 49–61. IEEE, 2019.

[7] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[8] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Local reasoning for global invariants, part I: region logic. *J. ACM*, 60(3):18:1–18:56, 2013.

[9] Yuyan Bao, Gary T. Leavens, and Gidon Ernst. Conditional effects in fine-grained region logic. In *FTfJP@ECOOP*, pages 5:1–5:6. ACM, 2015.

[10] Yuyan Bao, Gary T. Leavens, and Gidon Ernst. Unifying separation logic and region logic to allow interoperability. *Formal Aspects of Computing*, 30:381–441, 2018.

[11] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021.

[12] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Math. Struct. Comput. Sci.*, 6(6):579–612, 1996.

[13] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018.

[14] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for TLS. In *CCS*, pages 459–468. ACM, 2008.

[15] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320. ACM, 2007.

[16] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.

[17] John Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[18] Dave Clarke, James Noble, and Tobias Wrigstad, editors. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*. Springer, 2013.

[19] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.

[20] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003.

[21] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing redefined. In *IFL*, volume 4449 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2006.

[22] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007.

[23] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[24] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. Building and using pluggable type-checkers. In *ICSE*, pages 681–690. ACM, 2011.

[25] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Separating ownership topology and encapsulation with generic universe types. *ACM Trans. Program. Lang. Syst.*, 33(6):20:1–20:62, 2011.

[26] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for generic universe types. In *ECOOP*, volume 6813 of *Lecture Notes in Computer Science*, pages 333–357. Springer, 2011.

[27] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *J. Object Technol.*, 4(8):5–32, 2005.

[28] Werner Dietl and Peter Müller. Object ownership in program verification. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 289–318. Springer, 2013.

[29] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS*, pages 73–84. ACM, 2013.

[30] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why eve and mallory love android: an analysis of android SSL (in)security. In *CCS*, pages 50–61. ACM, 2012.

[31] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24. ACM, 2002.

[32] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, 2008.

[33] François Gagnon, Marc-Antoine Ferland, Marc-Antoine Fortier, Simon Desloges, Jonathan Ouellet, and Catherine Boileau. Androssl: A platform to test android applications connection security. In *FPS*, volume 9482 of *Lecture Notes in Computer Science*, pages 294–302. Springer, 2015.

[34] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *CCS*, pages 38–49. ACM, 2012.

[35] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[36] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

[37] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[38] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer, 2010.

[39] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285. ACM, 1991.

[40] Wei Huang, Werner Dietl, Ana L. Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *ECOOP*, volume 7313 of *Lecture Notes in Computer Science*, pages 181–206. Springer, 2012.

[41] Wei Huang, Ana L. Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: checking and inference of reference immutability and method purity. In *OOPSLA*, pages 879–896. ACM, 2012.

[42] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[43] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[44] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018.

[45] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. Verifying object construction. In *ICSE*, pages 1447–1458. ACM, 2020.

[46] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *ESEC/SIGSOFT FSE*, pages 181–192. ACM, 2021.

[47] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Accumulation analysis (artifact). *Dagstuhl Artifacts Ser.*, 8(2):22:1–22:3, 2022.

[48] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019.

[49] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 29–42, New York, NY, USA, 1999. Association for Computing Machinery.

[50] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In *PPDP*, pages 146–159. ACM, 2016.

[51] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: supporting developers in using cryptography. In *ASE*, pages 931–936. IEEE Computer Society, 2017.

[52] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Trans. Software Eng.*, 47(11):2382–2400, 2021.

[53] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and precision by combining expressive type systems and deductive verification. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[54] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *APSys*, pages 7:1–7:7. ACM, 2014.

[55] Ian Mackie. Lilac: a functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.

[56] Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In *ESOP*, volume 13240 of *Lecture Notes in Computer Science*, pages 346–375. Springer, 2022.

[57] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo A. Arango-Argoty. Secure coding practices in java: challenges and vulnerabilities. In *ICSE*, pages 372–383. ACM, 2018.

[58] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using mur-phi. In *IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society, 1997.

[59] João Mota, Marco Giunti, and António Ravara. Java typestate checker. In *COORDINATION*, volume 12717 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2021.

[60] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

[61] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478. ACM, 2007.

[62] Alan Mycroft and Janina Voigt. Notions of aliasing and ownership. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 59–83. Springer, 2013.

[63] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. "jumping through hoops": Why do java developers struggle with cryptography apis? In *Software Engineering*, volume P-267 of *LNI*, page 57. GI, 2017.

[64] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writingsecure code. In *CCS*, pages 1065–1077. ACM, 2017.

[65] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.

[66] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142, pages 1–19, Berlin, 2001.

[67] Oracle. Java Cryptography Architecture (JCA) Reference Guide. https://docs.oracle.com/en/java/javase/20/security/java-cryptography-architecture-jca-reference-guide.html.

[68] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. ISSTA '08, page 201–212, New York, NY, USA, 2008. Association for Computing Machinery.

[69] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *ISSTA*, pages 201–212. ACM, 2008.

[70] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[71] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *IEEE Symposium on Security and Privacy*, pages 1256–1274. IEEE, 2019.

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

[73] Sazzadur Rahaman and Danfeng Yao. Program analysis of cryptographic implementations for security. In *SecDev*, pages 61–68. IEEE Computer Society, 2017.

[74] Venkatesh-Prasad Ranganath and Joydeep Mitra. Are free android app security analysis tools effective in detecting known vulnerabilities? *Empir. Softw. Eng.*, 25(1):178–219, 2020.

[75] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, 2002. IEEE Computer Society Press.

[76] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's trustzone keymaster design. Cryptology ePrint Archive, Paper 2022/208, 2022. https://eprint.iacr.org/2022/208.

[77] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 358–379. Springer, 1993.

[78] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps. In *NDSS*. The Internet Society, 2014.

[79] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer, 2013.

[80] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[81] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, pages 447–458. ACM, 2011.

[82] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, page 561. North-Holland, 1990.

[83] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. Precise inference of expressive units of measurement types. *Proc. ACM Program. Lang.*, 4(OOPSLA):142:1–142:28, 2020.

[84] Weitian Xing, Yuanhui Cheng, and Werner Dietl. Ensuring correct cryptographic algorithm and provider usage at compile time. FTfJP 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. Automatic detection of java cryptographic API misuses: Are we there yet? *IEEE Trans. Software Eng.*, 49(1):288–303, 2023.