* * *

Securing Public-key Cryptography on the Android Platform

Hannah Gommerstadt

Advisors: Stephen Chong and Aslan Askarov

March 29, 2013

A thesis presented by

Hannah Gommerstadt

to

Computer Science and Mathematics

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

Abstract

Every second of every day large amounts of confidential data are encrypted to enable financial transactions, intelligence operations, and countless other tasks. Public-key cryptography underlies the security of all of these systems and is used to prevent an attacker from getting access to private information.

The cryptographic algorithms that secure the majority of these interactions are widely accepted as sufficiently strong to protect users' information. The implementations of these algorithms are usually encapsulated in third-party libraries, which are used by systems to secure data. However, there is little assurance that these systems use public-key cryptography correctly, even if they rely on correctly implemented third-party cryptographic libraries.

This thesis defines what it means for public-key cryptography to be secure and presents a mechanism that provably enforces it. The definition of safe public-key cryptography is based on one developed by Askarov, Hedin, and Sabelfeld (2008) to reason about information flow and private-key cryptography.

We extend our enforcement mechanism to handle language features of the Java programming language, and implement *Cryptflow*, a tool to track how secure information flows through Java programs and ensure correct use of public-key cryptography. This tool analyzes programs that employ cryptography and rejects programs that exhibit common security vulnerabilities found on the Android Platform, while accepting programs that use cryptography securely.

This thesis is a step toward enforcing the secure use of public-key cryptography and guaranteeing that confidential information is encrypted properly.

Acknowledgments

To Steve Chong, for your incredibly patient advising, for helping me debug at absurd hours, for teaching CS252r, and for forcing me to use LaTeX macros. This thesis would not have been possible without your constant support, especially on the implementation portion. I am so incredibly grateful.

To Aslan Askarov, for your fantastic earlier work on cryptographically masked flows, for inspiring this thesis, for teaching me how to prove theorems by structural induction, and for indoctrinating me with the importance of the \mathit macro.

To Greg Morrisett, for agreeing to read this thesis, and for teaching CS51 and CS153, which spurred my love of programming languages. Also, thank you for letting me teach CS51 three times; I never would have realized how much I love computer science otherwise.

To Salil Vadhan, for agreeing to read this thesis; I really hope that it is up to your cryptographic standards.

To the Gommerstadt clan, for your unwavering faith in me, for the continuous stream of gluten-free food leading up to the birthday of this thesis, and for not disowning me these last 8 months. I especially want to thank my mother Irene for copyediting this entire thesis.

To Stefan Muller, for being the computer scientist I aspire to be, for supplying a steady stream of Vizsla puppy pictures during the writing process, and for your helpful comments and edits on this thesis.

To my roommates and my friends, thank you for putting up with this thesis writer for the past 8 months.

Contents

Chapter 1

Introduction

Large systems are constantly manipulating sensitive and non-sensitive data. For example, when a customer accesses the internet to buy a new pair of shoes, she expects her credit card information to be kept secure, while her ratings of a shoe to be public. When a lost tourist uses his mobile phone to find the nearest place to procure caffeine, he might not mind that the model of his phone becomes public information, but would probably prefer his exact location to be kept secret. Every second of every day large amounts of confidential data are encrypted to enable financial transactions, intelligence operations, and countless other tasks.

Public-key cryptography underlies the security of all of these systems and is used to prevent an attacker from getting access to private information. Most developers do not write implementations of public-key encryption and decryption algorithms themselves, but rather, they make use of third-party libraries which encapsulate this functionality. However, there is little assurance that these systems use cryptography correctly, even if they rely on correctly implemented third-party cryptographic libraries. This thesis is concerned with preventing misuses of cryptography in systems that rely on correctly implemented third-party cryptographic libraries.

The problem of potentially insecure use of cryptographic protocols is particularly relevant when we consider the ubiquity of smart phones, which will be used by the majority of the American population within the next three years [22]. These phones have access to a high concentration of private data including one's exact GPS location, text messages, contacts, unique phone identifying number and more. While personal computers are protected by a bevy of measures including firewalls and complex anti-virus systems, mobile phones do not have similarly advanced defense mechanisms. This thesis aims to prevent common cryptographic vulnerabilities on the Android Platform which runs on 48.6% [2] of smart phones in the United States, making it the largest player in the American smart phone market.

```
// The bytes that represent the private key are hard-coded into the file
byte[] priv_key_bytes = { 0xe3, 0x00, ... };

// Turn the bytes into a PrivateKey which will later become part of a KeyPair object
PKCS8EncodedKeySpec key = new PKCS8EncodedKeySpec(priv_key_bytes);
KeyFactory kf = KeyFactory.getInstance("RSA");
PrivateKey priv_key = kf.generatePrivate(key);
```

Listing 1.1: Cryptography Vulnerability: hard-coded private key

Indeed, a recent report [27] found that over 40% of Android applications (out of close to 10,000 analyzed applications) use hard-coded cryptographic keys, which violates good encryption practice. The same report claimed that cryptographic issues cause 44% of security breaches on the Android platform and are the leading cause of security errors on the device. A hard-coded key is essentially public information, so any information encrypted with the compromised key can no longer be considered secure. To prevent the use of hard-coded keys, or other similar vulnerabilities, we need to provide stronger guarantees about the security of encrypted information.

Motivating Examples We present two examples which model the most frequent misuses of cryptography on the Android Platform. The first example shows a hardcoded cryptographic key and the second demonstrates the output of a private key to a public channel. Programs for the Android Platform are written in Java and make use of Java's cryptographic libraries.

In Java, a `KeyPair` class contains both the `PublicKey` needed for encryption and the `PrivateKey` needed for decryption. This class can be generated by the `KeyPairGenerator` class, or created from a `PrivateKey`, and a `PublicKey`, generated by the `KeyFactory` class. Both the `PublicKey`and the `PrivateKey` have a `getEncodedBytes` field that allows one to extract the bytes of the key, and store them in an array of bytes.

In the code snippet of Listing 1.1, a private key is created from a sequence of public bytes in the file. The code snippet exhibits a security vulnerability because the contents of the

```
// Extract the private key from a KeyPair object
KeyPair pair = ...;
PrivateKey priv_key = pair.getPrivate();

// Output the bytes of the private key to System.out and print them to the screen
byte[] priv_key_bytes = priv_key.getEncodedBytes();
System.out.println(Arrays.toString(priv_key_bytes));
```

Listing 1.2: Cryptography Vulnerability: output private key to public channel

private key are visible to anyone with access to the source code of the application. The correct approach would be to store keys in a secure private file, or in a key store, which is a secure database of keys.

In Listing 1.2, the bytes of a private key are converted to a string and output to the console via `System.out.println`. This code exhibits a security vulnerability because the contents of the private key are printed out and have become public information. A key should only be output to a secure private file, or to a key store; it should never be output on a public channel.

To prevent the construction of private keys from insecure data, and the flow of private keys to public channels, as in the examples above, we need to provide stronger guarantees about the security of encrypted information. In the first example, we can prevent a private key from being constructed from a low security array of bytes. In the second example, we can prevent any information about the private key from flowing to a public channel, such as `System.out.println`.

To establish these guarantees, we track the *information flow* through programs which employ cryptographic primitives. Because a program is a series of statements, or commands, we can analyze how secure information interacts with public information at every step of the program. Generally, a program is considered secure if the property of *noninterference* holds. This property stipulates that computations on private data may not have any influence on public data.

For example, an application that displayed a user's password in plaintext on its website would clearly violate noninterference because a private password has been output on a public channel. However, many security vulnerabilities are much more subtle. For example, a social networking application that displays the length of your password next to your name is not secure. This is so because even though your entire password has not been compromised, an attacker now knows how long it is, and with enough tries, could more easily break into your account. This also violates noninterference because information about the private password has leaked to a public channel.

We extend the notion of noninterference to allow safe encryption, decryption and key generation for public-key cryptographic protocols. Our extension is based on the work done by Askarov, Hedin and Sabelfeld [5] on symmetric encryption. Any program that complies with our extended notion of noninterference is defined as secure. Armed with this security condition, this thesis presents a provable enforcement mechanism for our security condition. This mechanism is a type system, which is a set of rules that governs how data can flow through a program, for a small, imperative, Java-like language with encryption, decryption, and key generation.

We prove that this type system guarantees our extended notion of noninterference, which implies that programs written in our language are provably secure. Finally, because Android applications are written in Java and largely make use of Java's cryptography libraries, we use the existing Polyglot [21] framework, which facilitates making extensions to the Java language, to implement *Cryptflow*, an information flow analysis of Java programs which employ cryptographic protocols. This tool analyzes programs that employ cryptography and rejects programs that exhibit common security vulnerabilities, while accepting programs that use cryptography securely. We run *Cryptflow* on programs that mimic common security vulnerabilities present on the Android Platform, and ensure that *Cryptflow* rejects those programs.

The rest of the thesis is structured as follows. We describe related work in Chapter 2.

A simple imperative programming language with primitives for public-key cryptography is presented in Chapter 3. Chapter 4 defines a security condition for public-key cryptography and presents a type system that provably enforces it. We describe Cryptflow, our tool for ensuring correct use of public-key cryptography for Java programs in Chapter 5. We conclude the thesis in Chapter 6.

Chapter 2

Background and Previous Work

This chapter presents the prior work that serves as the foundation of this thesis. We first describe the study of information flow which serves as the basis for the type system for our language in Section 2.1. We then overview the prior work done on using information flow techniques to reason about cryptography in Section 2.2. Finally, we conclude the chapter by type systems as a means for information flow control in Section 2.3.

§2.1 Information Flow

In this thesis we consider a system with two security levels, high and low, which model a situation with high-security and low-security data. This set of security levels has a partial ordering $\sqsubseteq$ which is reflexive, transitive and antisymmetric. Let $\mathcal{L} = \{L, H\}$. The partial ordering defines the set of permitted flows in our system. In this set of security levels, $L \sqsubseteq L$, $L \sqsubseteq H$, $H \sqsubseteq H$ and $H \not\sqsubseteq L$. For example, since $L \sqsubseteq H$ we say that L *flows into* H, which is permitted since low-security data can flow into high-security data. However, because $H \not\sqsubseteq L$, we say that H does not flow to L which indicates that we do not permit high-security data to flow to low-security data.

We can also define the *join* and *meet* operations on any two security levels in $\mathcal{L}$. In our system, the *join* of two levels is their least upper bound and the *meet* of two levels is their greatest lower bound. Denning [8] describes the generalization of a system with two levels of security to one with more levels and more complex relations on the levels.

There are multiple ways high-security data can flow to low-security data within a program.

```
// Assign the value of H to L
L = H;
```

Listing 2.1: Explicit Flow

```
// Assign the value of H to L
if H {
  L = 4;
}
else
  L = -5;
```

Listing 2.2: Implicit Flow

The majority of these flows are classified as either *explicit flows*, or *implicit* flows. An *explicit flow* occurs when some high-security data is assigned to a low-security variable, or when some high-security variable is output on a low-security channel. An *implicit* flow describes a situation where the control flow of the program depends on high security data [9]. Let L represent a variable which contains low-security public data, and let H be a variable which contains high-security secret data. Consider the program in Listing 2.1.

The code in Listing 2.1 is the most simple example of an explicit flow of information. This code is not secure, and violates *noninterference*. Recalling from Chapter 1, *noninterference* [14] is a strong semantic security guarantee that, in essence, requires that secret inputs do not influence public outputs. A semantic security condition is a security condition that is defined in terms of the semantics of the language.

While the previous example illustrates the *direct flow* of information, the *indirect flow* of sensitive information is even harder to reason about. Consider the program in Listing 2.2.

In the code in Listing 2.2, the variable H represents a high-security boolean variable. If H is true, then L is assigned a positive number, and otherwise L is assigned a negative number. While the value of H is not revealed directly, an attacker will observe different program behavior based on the high-security input.

By inundating the system with enough test cases, a clever attacker can figure out under which conditions a positive, or a negative number is produced. Therefore, he has deduced how the control flow of the program depends on the high-security variable and the security of the system has been compromised. Therefore, public outputs are influenced by sensitive inputs and this program also violates *noninterference*.

In a language-based setting, noninterference can be enforced by tracking and controlling the flow of information in a program, using program analyses such as type systems. In the presence of *implicit flows* it is hard to enforce *noninterference*. Most type systems solve this problem by tracking the level of the information stored in program counter which represents the upper bound of the security level at the current point in the program. For example, in the loop in Listing 2.2, the loop depends on a high-security variable, so the program counter would also be marked high. Therefore, an assignment to a low variable would be prohibited.

§2.2 Cryptographic Flows

While it is necessary to reason about information flow through systems that employ cryptographic protocols, the traditional definition of *noninterference* [14], the notion that no computations on private data may influence computations on public data, is insufficient for our analysis. This occurs because the value produced by encrypting a high-security value, the *plaintext*, with a key, frequently called the *ciphertext*, is by definition a low-security public value. It is a low-security public value because it is constructed to conceal sensitive information well enough to be broadcast publically. This low-security ciphertext depends on the high-security plaintext and the key used for encryption. If the private plaintext is varied, then there might possibly be variation in the low-security ciphertext, so noninterference is broken.

To extend the notion of noninterference to allow safe encryption, decryption and key generation for public-key cryptographic protocols, we use a form of noninterference called *possibilistic noninterference*. Possibilistic noninterference is based on the idea that if the ciphertext can possibly be any value, then a change in the high-security plaintext does affect the low-security ciphertext. This occurs because in both situations, the ciphertext is possibly any value, so the attacker is not able to extract new information from a change in the high-security plaintext. We define our notion of possibilistic noninterference with respect to a probabilistic encryption algorithm with nondeterministic encryption.

This extension furthers the work done on cryptographically-masked flows by Askarov et al [5] that presents a security condition and a type system for a language with symmetric encryption. We extend their notion of possibilistic noninterference to support safe public-key encryption. We also offer an implementation that enforces our type system.

Approaches to dealing with information flow in the presence of cryptography tend to extend the notion of noninterference to accommodate information-theoretic artifacts of cryptography. Traditionally, such extensions treat cryptographic primitives as a black-box [10]. Our approach has its roots in Abadi's model [1] for symmetric-key cryptographic protocols. Vaughan and Zdancewic[26] present a language in which security labels are connected to public-key cryptography; however, as argued in [5], this approach is prone to occlusion [24].

An instance of occlusion occurs when all ciphertexts are considered low-equivalent, or indistinguishable to an attacker. Let x and y be two ciphertexts that are indistinguishable to an attacker, and H a high-security boolean variable. Consider the code in Listing 2.3.

In this code we cannot distinguish whether x and y are equivalent. However, their equality, or inequality, leaks information about the secret value H, so a leak has occurred. The main issue with occlusion is that an indistinguishability definition (which is used to simplify the security guarantees for encryption and decryption) may also mask other unintended leaks. Our approach protects against this variety of occlusion.

```
// Encrypt plaintext a with key k
x = encrypt(k,a);
if H {
  // Encrypt plaintext a with key k
  y = encrypt(k,a);
}
else
  y = x;
```

Listing 2.3: Occlusion

Much recent work [19, 18, 12, 11, 17, 25, 13] focuses on the computational probabilistic guarantees of programs that use cryptography. While this thesis aims at showing a simpler possibilistic guarantee, we believe that computational soundness of our enforcement may be established by following the Laud's analysis of cryptographically-masked flows [20]. Another advantage of using a possibilistic condition is composition with possibilistic policies for declassification and key release [6] and security in the presence of dynamic policies [4]. As far as we know, these policies have no probabilistic counterparts.

From the implementation perspective, the work most closely related to ours is the one by Küsters et al [17] that analyzes implementation-level usage of cryptographic primitives in Java-like programs. Our analysis is implemented as a modular extension to the ObjAnal framework [7] which covers the full Java language. As a part of a bigger framework, our implementation benefits from the additional analyses done by other components of the ObjAnal framework, allowing easy extensions to our implementation, such as key release and declassification.

§2.3 Type Systems as Enforcement Mechanisms

The most common method of statically (at compile time) enforcing the secure flow of information is by means of a type system. In a type system used to enforce information flow,

```
// Assign the value of H to L
x = y;
```

Listing 2.4: Insecure Assignment

```
// Return 4 if H is true or false
if H {
  L = 4;
}
else
  L = 4;
```

Listing 2.5: Secure Program Rejected by Type System

the type of a variable usually encodes its basic type, such as whether it is an integer or a boolean, for example, and its security level. For some mapping Γ that maps variables to types, program counter pc, and command c, we write $\Gamma, pc \vdash c$ to indicate that the command c is well-typed with respect to Γ and the pc. The pc represents an upper bound on the security level of the program just before c is executed.

Let x be a low integer that has type int L and y be a high integer that has type int H. The code in Listing 2.4 would be rejected by the type system, because assignment from a low level variable to a high level variable is forbidden.

Consider the program in Listing 2.5. This program leakes no information (the public output is always 4), but because assignment to low-security variables happens when the pc is high (in a loop which depends on high-security data the pc is marked high), this program is rejected by the type system.

The program in Listing 2.5 is rejected by the type system even though it is secure and does not leak information. Some secure programs will inevitably be rejected by the type system, but any insecure program is guaranteed to be rejected. This means that the enforcement mechanism is conservative because any sound type system (a type system that guarantees that a well-typed program will not cause an error) will reject some secure programs. There

are other ways to enforce the secure flow of information besides using a type system, such as runtime approaches. Sabelfeld and Myers [23] survey other approaches for language-based information-flow control.

Chapter 3

Language Based Model

We reason about the security of cryptographic keys using a simply typed imperative language that includes cryptographic primitives and input/output. This section presents the syntax and semantics for that language.

§3.1 Syntax

Our primitives for public-key cryptography include commands for encryption, decryption, and key generation. The full syntax is given in Figure 3.1, where $x \in \mathit{VarName}$ ranges over the set of variable names, $ch \in \mathit{ChanName}$ ranges over the set of channel names, and $n \in \mathbb{Z}$ ranges over the integers.

§3.2 Semantics

This section presents the semantics of the expressions and the commands in the language.

Expressions	e	::=	$n \mid x \mid e_1 \text{ op } e_2 \mid (e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e)$
Commands	c	::=	$\text{skip} \mid x := e \mid c_1; c_2 \mid x := \text{encrypt}(e_1, e_2) \mid x := \text{decrypt}(e_1, e_2) \mid$ $(x, y) := \text{newkeypair} \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \mid$ $\text{in}(x, ch) \mid \text{out}(ch, e)$

Figure 3.1: Syntax

Values in our language are integers, tuples, public encryption keys, private decryption keys, and ciphertexts.

$$\text{Values} \quad v \quad ::= \quad n \mid (v_1, v_2) \mid k_{pub} \mid k_{priv} \mid u$$

Our system is parametrized over a public key encryption scheme $\mathcal{AE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$, where

- $\mathcal{K}$ is a key generation algorithm that takes no input and returns a pair (k_{pub}, k_{priv}) of keys, where k_{pub} is a public key, and k_{priv} is a matching private key. We let Key_{priv} be the set of private keys and Key_{pub} be the set of public keys.
- $\mathcal{E}$ is a nondeterministic encryption algorithm that takes the public key k_{pub} and a plaintext value v and returns a ciphertext u. We write $\mathcal{E}(k_{pub}, v)$ to denote the set of possible ciphertexts that can be obtained by encrypting value v with key k_{pub}.
- $\mathcal{D}$ is a deterministic decryption algorithm that takes the private key k_{priv} and a ciphertext u, and returns the corresponding plaintext value v.

Let M be a variable environment that maps variables to values. The semantics for expressions is defined as a standard big-step relation $\langle M, e\rangle \Downarrow v$. Here, expression e is evaluated in the variable environment M, producing the value v. The full semantics of expressions is given in Figure 3.2.

For the semantics of commands, we use environments E that are composed of a 4-tuple (M, G, I, O). Here, as before, M is a variable environment that maps variables to values, G is a stream of private and public key pairs generated by $\mathcal{K}$, and I and O are environments that map channel names to streams of values. We write $\langle E, c\rangle \Downarrow E'$ to mean that when the command c is executed in the environment E, it produces the environment E'. While most of

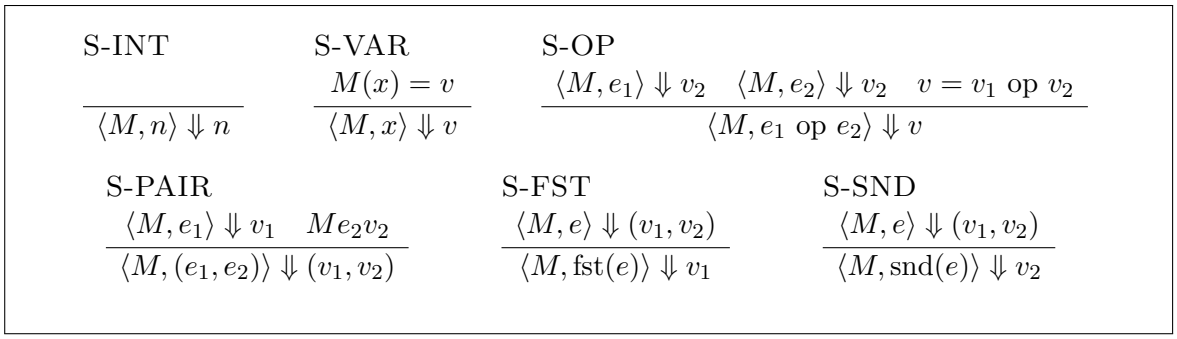

Figure 3.2: Semantics for Expressions

the semantics in this language are standard, the encryption, decryption and key generation commands are non-standard.

Figure 3.3 presents all the semantic rules for commands. The rest of this section presents the interesting rules for encryption, decryption, and key generation.

Rule (S-ENCRYPT) describes the semantics of the encryption command $x := \text{encrypt}(e_1, e_2)$. The first argument, e_1, is a public key and the second argument, e_2, is the value to be encrypted. The resulting ciphertext, $u \in \mathcal{E}(k, v)$, is one of the set of possible encryptions that this specific key and value could create. We note that this command is nondeterministic; it does not have one defined execution, but rather a set of possible executions. The nondeterminism of the encryption command is essential to our notion of noninterference as described in Section 4.3.

Rule (S-DECRYPT) describes the semantics of the decryption command $x := \text{decrypt}(e_1, e_2)$. The first argument, e_1, is a private key and the second argument, e_2, is the ciphertext to be decrypted. The resulting value, $v = \mathcal{D}(k, u)$ is the one possible decryption of the ciphertext u with that key. There is no nondeterminism in this command and it has one defined execution, unlike the encryption command.

Rule (S-NEWKEYPAIR) describes the semantics for the key generation command $(x, y) :=$

newkeypair. The first element of the resulting pair, x, is a public key, and the second element of the resulting pair, y is a private key. This tuple is taken from the key pair stream G.

The semantics of the language is nondeterministic because the encryption command is nondeterministic. We write $[\![\langle E, c\rangle]\!]$ to indicate the the set of possible environments that can be produced by the execution of command c in environment E. That is,

$$[\![\langle E, c\rangle]\!] = \{E' \mid \langle E, c\rangle \Downarrow E'\}$$

We extend the notation to handle a set of input environments. That is, if $\widehat{E}$ is a set of environments, then $[\![\langle \widehat{E}, c\rangle]\!]$ denotes the union of the sets $[\![\langle E, c\rangle]\!]$ for $E \in \widehat{E}$.

We use this notation in the statement of our soundness result in Section 4.4.

$$[\![\langle \widehat{E}, c\rangle]\!] = \bigcup\{[\![\langle E, c\rangle]\!] \mid E \in \widehat{E}\} = \{E' \mid E \in \widehat{E} \wedge \langle E, c\rangle \Downarrow E'\}$$

S-SKIP

$$\frac{}{\langle E, \text{skip}\rangle \Downarrow E}$$

S-SEQ

$$\frac{\langle E, c_1\rangle \Downarrow E' \quad \langle E', c_2\rangle \Downarrow E''}{\langle E, c_1; c_2\rangle \Downarrow E''}$$

S-ASSIGN

$$\frac{\langle M, e\rangle \Downarrow v}{\langle (M, G, I, O), x := e\rangle \Downarrow (M[x \mapsto v], G, I, O)}$$

S-IF1

$$\frac{\langle M, e\rangle \Downarrow v \quad v \neq 0 \quad \langle (M, G, I, O), c_1\rangle \Downarrow E'}{\langle (M, G, I, O), \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \Downarrow E'}$$

S-IF2

$$\frac{\langle M, e\rangle \Downarrow 0 \quad \langle (M, G, I, O), c_2\rangle \Downarrow E'}{\langle (M, G, I, O), \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \Downarrow E'}$$

S-WHILE1

$$\frac{\langle M, e\rangle \Downarrow v \quad v \neq 0 \quad \langle (M, G, I, O), c; \text{while } e \text{ do } c\rangle \Downarrow E'}{\langle (M, G, I, O), \text{while } e \text{ do } c\rangle \Downarrow E'}$$

S-WHILE2

$$\frac{\langle M, e\rangle \Downarrow 0}{\langle (M, G, I, O), \text{while } e \text{ do } c\rangle \Downarrow (M, G, I, O)}$$

S-INPUT

$$\frac{I(ch) = v \cdot vs}{\langle (M, G, I, O), \text{in}(x, ch)\rangle \Downarrow (M[x \mapsto v], G, I[ch \mapsto vs], O)}$$

S-OUTPUT

$$\frac{\langle M, e\rangle \Downarrow v}{\langle (M, G, I, O), \text{out}(ch, e)\rangle \Downarrow (M, G, I, O[ch \mapsto v \cdot O[ch]])}$$

S-ENCRYPT

$$\frac{\langle M, e_1\rangle \Downarrow k \quad \langle M, e_2\rangle \Downarrow v \quad k \in Key_{pub} \quad u \in \mathcal{E}(k, v)}{\langle (M, G, I, O), x := \text{encrypt}(e_1, e_2)\rangle \Downarrow (M[x \mapsto u], G, I, O)}$$

S-DECRYPT

$$\frac{\langle M, e_1\rangle \Downarrow k \quad \langle M, e_2\rangle \Downarrow u \quad k \in Key_{priv} \quad v = D(k, u)}{\langle (M, G, I, O), x := \text{decrypt}(e_1, e_2)\rangle \Downarrow (M[x \mapsto v], G, I, O)}$$

S-NEWKEYPAIR

$$\frac{G = (k_1, k_2) \cdot ks}{\langle (M, G, I, O), (x, y) := \text{newkeypair}\rangle \Downarrow (M[x \mapsto k_1, y \mapsto k_2], ks, I, O)}$$

Figure 3.3: Semantics for Commands

Chapter 4

Security Guarantees

This section presents the security condition for public-key cryptography and the type system that enforces it. Section 4.1 presents the security levels and the intuition behind them. Section 4.2 defines the attacker observation model and Section 4.3 defines our notion of possibilistic noninterference. Section 4.4 defines the type system and Section 4.5 shows how the type system prevents some common vulnerabilities.

§4.1 Security Levels

The system considered in this thesis consists of the following security levels. As explained in Section 2.1, we consider a system that has high-security and low-security data, represented by the levels H and L, respectively. However, we also consider the two types of keys, public keys for encryption and private keys for decryption. Let $\mathcal{L} = \{H, L\}$. The set of security levels we consider are $L \cup \{publickey, privatekey\}$. We define a partial order $\sqsubseteq$ on this set that models the permitted flows. These flows are shown in Figure 4.1. We note that the *meet*, $\sqcap$, and *join*, $\sqcup$, operations are only defined for any two levels in $\mathcal{L}$. For example, $privatekey \sqcup L$ is undefined.

The solid line shows that we allow flows from data marked with the level L to data marked with the level H. More specifically, we define $L \sqsubseteq L$, $L \sqsubseteq H$, $H \sqsubseteq H$ and $H \not\sqsubseteq L$. We do not permit information marked with the security level H to flow to information marked with the level L.

We do not allow public keys to flow into private keys, or other high values. While a public

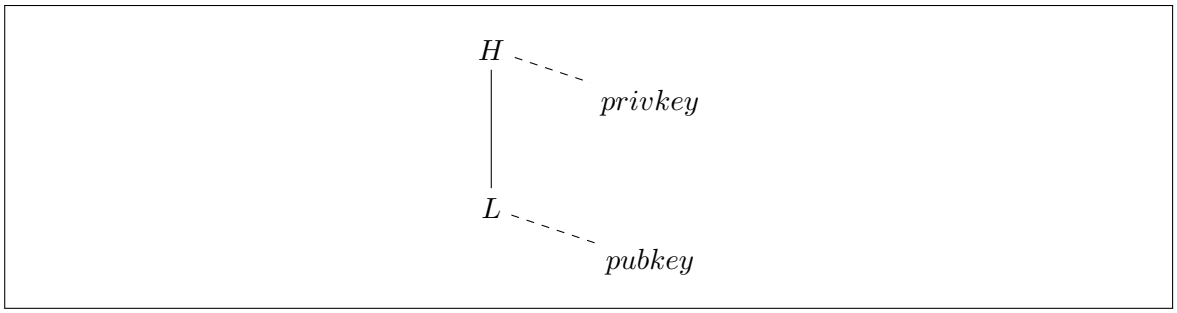

Figure 4.1: Security Levels

key can flow to a low value, as shown with a dotted line, this flow is not particularly useful because public keys are only used for encryption. Our type system provides typed output channels with enough structure to make this flow unnecessary.

We also do not allow private keys to flow into public keys, or low values. While a private key can flow into a high value, as shown with a dotted line, this flow does not have a purpose, since private keys are used only for decryption and our type system provides typed channels to output private keys, if necessary. We do not allow anything to flow into a public key.

§4.2 Low Equivalence

To define noninterference, we first define an attacker observation model. We do this by introducing a low equivalence relation with respect to all the possible types. That is, for each security type τ we define low-equivalence relation $\approx_\tau$ such that if v and v' are values of type τ and $v \approx_\tau v'$, then the attacker is unable to distinguish values v and v'. Inference rules for these equivalence relations are given in Figure 4.2. Intuitively, the attacker is only able to see low-security information.

$$\text{LE-INT-L}\ \frac{}{n \approx_{int\ L} n} \qquad \text{LE-INT-H}\ \frac{}{n_1 \approx_{int\ H} n_2}$$

$$\text{LE-ENC-L}\ \frac{\exists u_i, k_i\ .\ v_i = D(k_i, u_i) \quad i = 1,2 \qquad k_1 \approx_{privkey} k_2 \quad v_1 \approx_{\tau} v_2 \quad u_1 \doteq u_2}{u_1 \approx_{enc\ \tau\ L} u_2} \qquad \text{LE-ENC-H}\ \frac{}{u_1 \approx_{enc\ \tau\ H} u_2}$$

$$\text{LE-PRIVKEY}\ \frac{}{k_1 \approx_{privkey} k_2} \qquad \text{LE-PUBKEY}\ \frac{}{k \approx_{pubkey} k} \qquad \text{LE-PAIR}\ \frac{v_{11} \approx_{\tau_1} v_{21} \quad v_{21} \approx_{\tau_2} v_{22}}{(v_{11}, v_{12}) \approx_{(\tau_1,\tau_2)} (v_{21}, v_{22})}$$

Figure 4.2: Low Equivalence for Values

Because we do not consider all ciphertexts indistinguishable, we must define a low-equivalence relation for ciphertexts. In our language, a ciphertext has type $enc\ \tau\sigma$ where τ is the type of the plaintext and σ is either high or low. Any two high-security ciphertexts are indistinguishable to the attacker. However, any two low-security ciphertexts are not indistinguishable. In order to define the low equivalence relation with respect to the $enc\ \tau\ L$ type we define a helper low equivalence relation $\doteq$. We require that for any choice of plaintext and key there will be exactly one other related ciphertext for any other plaintext and key. We formalize this below.

$$u_1 \in \mathcal{E}(k_1, v_1) \implies \exists u_2\ .\ u_2 \in \mathcal{E}(k_2, v_2) \wedge u_1 \doteq u_2$$

$$\exists u_1, u_2\ .\ u_1 \in \mathcal{E}(k_1, v_1) \wedge u_2 \in \mathcal{E}(k_2, v_2) \wedge u_1 \not\doteq u_2$$

We now briefly discuss how low-equivalence $\doteq$ can be defined for probabilistic encryption algorithms. Many such algorithms, for example, El Gamal, generate a random string r that is used as part of the encryption. The resulting ciphertext u is represented as a tuple $u = (c_1, c_2)$, where c_1 is a function of the random string r and the encryption key k_{pub}, and c_2 is a function of the plaintext to be encrypted v and the random string r, and possibly of the key k_{pub}. In general, we can represent the ciphertext (c_1, c_2) as a tuple $(f(r), \mathsf{ENC}(r, v, k_{pub}))$, where ENC is the implementation of the rest of the encryption algorithm. Using this structure, we can define low-equivalence $\doteq$ by relating equally-sized ciphertexts obtained with the same

$$\text{LE-MEM}\quad \frac{\forall x \in Dom(\Omega) \quad M_1(x) \approx_{\Omega(x)} M_2(x)}{M_1 \approx_\Omega M_2}$$

$$\text{LE-IOENV}\quad \frac{\forall ch \in Dom(\Theta) \quad X_1(ch) \approx_{\Theta(ch)} X_2(ch)}{X_1 \approx_\Theta X_2}\ (X_1, X_2) \in \{(I_1, I_2), (O_1, O_2)\}$$

$$\text{LE-STREAM1}\quad \frac{}{\epsilon \approx_{\tau\ stream} \epsilon}$$

$$\text{LE-STREAM2}\quad \frac{v_1 \approx_\tau v_2 \quad vs_1 \approx_{\tau\ stream} vs_2}{v_1 \cdot vs_1 \approx_{\tau\ stream} v_2 \cdot vs_2}$$

$$\text{LE-ENV}\quad \frac{M \approx_\Omega M' \quad G \approx_{(pubkey, privkey)\ stream} G' \quad (I, O) \approx_\Theta (I', O')}{(M, G, I, O) \approx_{(\Omega,\Theta)} (M', G', I', O')}$$

Figure 4.3: Low Equivalence for Memories, Channels, and Environments

random string:

$$\forall k_{pub}, k'_{pub}, v, v' \ .\ (f(r), \mathsf{ENC}(r, v, k)) \doteq (f(r), \mathsf{ENC}(r, v', k'))$$

For non-deterministic public-key encryption schemes such as RSA, a probabilistic encryption, which would satisfy the above definition, can be obtained using mechanism of Goldwasser and Micali [15].

We extend the low-equivalence relation to memories, channels, streams and environments. Inference rules for these relations are in Figure 4.3. Let Ω be a mapping from variables to security types, and Θ be a mapping from channel names to types. The low-equivalence relations for memories, input and output environments, and streams are defined straightforwardly using low-equivalence on values. Similarly, low-equivalence for environments (M, G, I, O) is defined using the low-equivalence relations on memories M, key pair streams G, and input and output environments I and O.

We lift the low-equivalence relation $\approx_{(\Omega,\Theta)}$ on environments to a low-equivalence relation

$\hat{\approx}_{(\Omega,\Theta)}$ on sets of environments in the standard way. Let $\widehat{E_1}$ and $\widehat{E_2}$ be two sets of environments. We define $\widehat{E_1}\hat{\approx}_{(\Omega,\Theta)}\widehat{E_2}$ as follows.

$$\widehat{E_1}\hat{\approx}_{(\Omega,\Theta)}\widehat{E_2} \iff \forall E_1' \in \widehat{E_1}.\, \exists E_2' \in \widehat{E_2}.\, E_1' \approx_{(\Omega,\Theta)} E_2' \text{ and } \forall E_2' \in \widehat{E_2}.\, \exists E_1' \in \widehat{E_1}.\, E_1' \approx_{(\Omega,\Theta)} E_2'$$

Intuitively, $\widehat{E_1}$ is indistinguishable from $\widehat{E_2}$ to an attacker if for every environment in $\widehat{E_1}$ there is an equivalent environment in $\widehat{E_2}$, and vice versa. This equivalence relation over sets of environments is key to our definition of noninterference.

§4.3 Noninterference

Intuitively, command c satisfies noninterference if for every pair of low-equivalent environments E_1 and E_2, the set of possible behaviors that executing c in these environments is indistinguishable to the attacker. That is, the attacker is unable to determine whether the program started execution in environment E_1 or E_2.

Definition 1 (Noninterference)**.** *Command c is* noninterfering *with respect to Ω and Θ if for all environments E_1 and E_2 such that $E_1 \approx_{(\Omega,\Theta)} E_2$ and $[\![\langle E_1, c\rangle]\!] \neq \emptyset$ and $[\![\langle E_2, c\rangle]\!] \neq \emptyset$ then*

$$[\![\langle E_2, c\rangle]\!] \hat{\approx}_{(\Omega,\Theta)} [\![\langle E_2, c\rangle]\!].$$

Security Levels	σ	::=	$L \mid H$
Basic Types	t	::=	$int \mid enc\ \tau$
Security Types	τ	::=	$t\ \sigma \mid privkey \mid pubkey \mid (\tau_1, \tau_2)$

Figure 4.4: Types

§4.4 The Type System

In this section, we cover the type system of this language, including the typing of expressions, the typing of commands and subtyping.

Figure 4.4 describes the types for our language. Security levels σ represent the restrictions that should apply to the use of data. For simplicity, we only consider two levels of security, L, for public information, and H, for secret information. We assume $L \sqsubseteq H$ and $H \not\sqsubseteq L$.

Basic types t include *int* (the type for integer values) and $enc\ \tau$ (the type for encryptions of a plaintext value of type τ). Security types τ include $t\ \sigma$ for basic type t and security level σ, which represents a value of type t whose use is restricted according to σ. Security types *privkey* and *pubkey* are the types of private keys and public keys respectively. Public keys are used to encrypt data, and private keys are used to decrypt data. Type (τ_1, τ_2) is the type of a pair of a τ_1 value and a τ_2 value.

Expressions

Variable typing context Ω maps variables to security types. We write $\Omega \vdash e : \tau$ if the expression e has type τ in context Ω. Inference rules for the expression typing judgment are given in Figure 4.5, and are standard.

$$\text{T-INT}\ \frac{}{\Omega \vdash n : \mathit{int}\ L} \qquad \text{T-VAR}\ \frac{\Omega(x) = \tau}{\Omega \vdash x : \tau} \qquad \text{T-PAIR}\ \frac{\Omega \vdash e1 : \tau_1 \quad \Omega \vdash e_2 : \tau_2}{\Omega \vdash (e_1, e_2) : (\tau_1, \tau_2)} \qquad \text{T-FST}\ \frac{\Omega \vdash e : (\tau_1, \tau_2)}{\Omega \vdash \mathrm{fst}(e) : \tau_1}$$

$$\text{T-SND}\ \frac{\Omega \vdash e : (\tau_1, \tau_2)}{\Omega \vdash \mathrm{snd}(e) : \tau_2} \qquad \text{T-OP}\ \frac{\Omega \vdash e_1 : \mathit{int}\ \sigma_1 \quad \Omega \vdash e_2 : \mathit{int}\ \sigma_2}{\Omega \vdash e_1\ \mathrm{op}\ e_2 : \mathit{int}\ (\sigma_1 \sqcup \sigma_2)}$$

Figure 4.5: Typing Rules for Expressions

Commands

Before we present the typing for commands, we present some useful functions and notation. We write τ^σ to mean that security type τ is *tainted* with security level σ, that is, the security level of τ is increased to σ. The full definition of the tainting function is presented below.

$$(t\ \sigma)^{\sigma'} = t\ (\sigma \sqcup \sigma')$$

$$(\tau_1, \tau_2)^\sigma = (\tau_1^\sigma, \tau_2^\sigma)$$

$$\mathit{pubkey}^L = \mathit{pubkey}$$

$$\mathit{privkey}^\sigma = \mathit{privkey}$$

Note that we do not allow a public key to be tainted with security level H. A private key, however, may be tainted with either H or L, but $\mathit{privkey}^\sigma = \mathit{privkey}$. We also define the *least* function, which represents the lowest bound on the security level of a type, as follows:

$$\mathit{least}(\mathit{int}\ \sigma) = \sigma$$

$$\mathit{least}(\mathit{enc}\ \tau\ \sigma) = \mathit{least}(\tau) \sqcap \sigma$$

$$\mathit{least}(\tau_1, \tau_2) = \mathit{least}(\tau_1) \sqcap \mathit{least}(\tau_2)$$

Channel typing environments Θ map channel names to security types. If $\Theta(ch) = \tau$, then only values of type τ may be send on channel ch. This structure allows us to safely output

keys on appropriate channels. Context $\Gamma = (\Omega, \Theta)$ is a pair of a variable typing environment Ω and channel typing environment Θ.

We define the judgment $\Gamma, pc \vdash c$ to mean that the command c is well typed with respect to the variable type environment Ω, the channel type environment Θ and the program counter's security level pc. The inference rules for this judgment are given in Figure 4.6. While most of the typing for commands is standard for a security-typed imperative language, the input, output, encryption, decryption and key generation commands are distinct to our language.

Input command $\text{in}(x, ch)$ inputs a value x from a channel ch. We require that the pc flow into the lowest possible level of ch and that the type of the channel be a subtype of the type of x. Intuitively, these requirements mean that in high context we cannot read from a low channel, and that the type of the channel must be compatible with that of x.

Output command $\text{out}(ch, x)$ outputs an expression e as the value x to a channel ch. To type check, τ tainted with the pc must be a subtype of the type of the channel, where τ is the type of e. The intuition for this is that we desire to prevent situations where the $\tau^{pc} = H$ and $\mathit{least}(\Theta(ch)) = L$ leading to the output of high values on low channels.

An encrypt command is of the form $x = \text{encrypt}(e_1, e_2)$. An encryption will only type check if the first argument, e_1 is a public key and the pc flows into the lowest possible level of the type of x. Intuitively, this makes sense because we want to avoid encrypting a low value where $\mathit{least}(\Omega(x)) = L$ in the context where $pc = H$ because in this situation high security information has influenced the low security encryption value.

A decrypt command is of the form $x = \text{decrypt}(e_1, e_2)$. It requires that the first argument e_1 be a private key, and that the second argument, e_2 be an encryption of some type τ with some level σ. A decryption can only occur if tainting τ with lowest upper bound of the pc and σ is a subtype of the type of x. Intuitively, this means that in order for a successful decryption to occur, the level of x must be at least as secure as the type of the encryption

$$\text{T-SKIP}\quad \frac{}{\Gamma, pc \vdash skip}$$

$$\text{T-SEQ}\quad \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\text{T-ASIGN}\quad \frac{\Omega \vdash e : \tau \quad \tau^{pc} <: \Omega(x)}{(\Omega, \Theta), pc \vdash x := e}$$

$$\text{T-IF}\quad \frac{\Omega \vdash e : int\ \sigma \quad (\Omega, \Theta), pc \sqcup \sigma \vdash c_i \quad i = 1, 2}{(\Omega, \Theta), pc \vdash \text{if } e \text{ then } c_1 \text{ else } c_2}$$

$$\text{T-WHILE}\quad \frac{\Omega \vdash int\ \sigma \quad (\Omega, \Theta), pc \sqcup \sigma \vdash c}{(\Omega, \Theta), pc \vdash \text{while } e \text{ do } c}$$

$$\text{T-INPUT}\quad \frac{least(\Theta(ch)) = \sigma \quad pc \sqsubseteq \sigma \quad \Theta(ch) <: \Omega(x)}{(\Omega, \Theta), pc \vdash \text{in}(x, ch)}$$

$$\text{T-OUTPUT}\quad \frac{\Omega \vdash e : \tau \quad \tau^{pc} <: \Theta(ch)}{(\Omega, \Theta), pc \vdash \text{out}(ch, x)}$$

$$\text{T-ENCRYPT}\quad \frac{\Omega \vdash e_1 : pubkey \quad \Omega \vdash e_2 : \tau \quad pc \sqsubseteq least(\Omega(x))}{(\Omega, \Theta), pc \vdash x := \text{encrypt}(e_1, e_2)}$$

$$\text{T-DECRYPT}\quad \frac{\Omega \vdash e_1 : privkey \quad \Omega \vdash e_2 : enc\ \tau\ \sigma \quad \tau^{pc \sqcup \sigma} <: \Omega(x)}{(\Omega, \Theta), pc \vdash x := \text{decrypt}(e_1, e_2)}$$

$$\text{T-NEWKEYPAIR}\quad \frac{\Omega(x) = pubkey \quad \Omega(y) = privkey}{(\Omega, \Theta), L \vdash (x, y) := \text{newkeypair}}$$

Figure 4.6: Typing Rules for Commands

with the *pc* and the level of the encryption σ folded in.

Key-pair generation command $(x, y) = \text{newkeypair}$ only type checks when the *pc* is low, and requires that a tuple is created where the first element is a public key and the second element is a private key. The key generation can only happen in a low context because a public key cannot be generated in a high context.

Subtyping

We say that the type τ_1 is a subtype of the type τ_2, if everywhere a value of type τ_2 is expected, a value of type τ_1 can be used. We write this as $\tau_1 <: \tau_2$. The subtyping relation is defined in Figure 4.7.

$$\text{SUB-INT}\ \frac{\sigma_1 \sqsubseteq \sigma_2}{int\ \sigma_1 <: int\ \sigma_2} \qquad \text{SUB-PAIR}\ \frac{\tau_1 <: \tau_3 \quad \tau_3 <: \tau_4}{(\tau_1, \tau_2) <: (\tau_3, \tau_4)} \qquad \text{SUB-ENC}\ \frac{\tau_1 <: \tau_2 \quad \sigma_1 \sqsubseteq \sigma_2}{enc\ \tau_1\ \sigma_1 <: enc\ \tau_2\ \sigma_2} \qquad \text{SUB-PRIV}\ \frac{}{privkey <: privkey}$$

$$\text{SUB-PUB}\ \frac{}{pubkey <: pubkey}$$

Figure 4.7: Subtyping

Soundness

The type system soundly enforces the security condition of noninterference, and prevents misuse of cryptographic primitives, such as the vulnerabilities discussed in Chapter 1.

Theorem 1 (Type-soundness)**.** *If command* $(\Omega, \Theta), pc \vdash c$*, then* c *is noninterfering with respect to* Ω *and* Θ*.*

We prove this theorem by structural induction on the commands. This theorem relies on two key lemmas, Lemma 1 which proves the noninterference of expressions and Lemma 2 that shows that running a command in a high context renders the resulting environment low equivalent to the initial environment.

Lemma 1. *If* $\Omega \vdash e : \tau$ *and* $M_1 \approx_\Omega M_2$*, then* $M_1(e) \approx_\tau M_2(e)$*.*

This lemma states that if an expression is well typed and if two memories are low equivalent, then the evaluations of that expression in the two memories must be low equivalent with respect to its type. We prove this lemma by structural induction on the type of the expression.

```
// The public integer that is the private key
secret = 25;
// Make a private key out of this integer
priv_key = secret;
```

Listing 4.1: Securing Cryptography Vulnerability #1

Lemma 2. *If* $(\Omega, \Theta), H \vdash c$*, then for all environments* E *and* E' *such that* $E' \in [\![\langle E, c\rangle]\!]$*, we have* $E' \approx_{(\Omega,\Theta)} E$*.*

This lemma states that if a command is run in a high context then the resulting environment is low equivalent to the initial environment. By proving that an environment produced by running a command in a high context is low equivalent to the original environment, we lift this notion to sets of environments and prove our noninterference condition. We prove this lemma by structural induction on the commands.

§4.5 Examples

In this section, we reimplement the motivating examples from Chapter 1 and show that they are rejected by our type system. The example from Listing 1.1 is rewritten in Listing 4.1.

This code is rejected by our type system because the assignment of `secret` to `priv_key` does not type check by rule (T-ASSIGN). Here, the type of `secret` is $int\ L$ and the type of `priv_key` is *privkey*. Regardless of the level of the program counter pc, $(int\ L)^{pc}$ cannot be a subtype of *privkey*. Therefore, this code does not type check and it is correctly rejected.

We reimplement the code from Listing 1.2 in Listing 4.2. This code is rejected by our type

```
// Generate a key pair
keypair = newkeypair;
// Extract the private key from the key pair
priv_key = snd(keypair);
// Output the private key to the low channel ch
out(ch, priv_key);
```

Listing 4.2: Securing Cryptography Vulnerability #2

system because the output of `priv_key` to `ch` does not type check by rule (T-OUTPUT). Here, the type of `priv_key` is *privkey*. Regardless of the level of the program counter *pc*, type $\Theta(ch)^{pc}$ cannot be a subtype of *privkey*. Therefore, this code is rejected and this vulnerability is prevented.

Chapter 5

Implementation

We have extended the ideas of the type system of Section 4.4 to the Java programming language, and instantiated them in the tool Cryptflow. Cryptflow is implemented using the Accrue ObjAnal framework [7] for interprocedural analysis of Java programs. ObjAnal is itself built as a compiler extension to Polyglot [21], an extensible compiler framework for Java. This chapter presents *Cryptflow* in Section 5.1 and presents some simple *Cryptflow* programs in Section 5.2.

§5.1 Cryptflow

The analysis we have implemented in Cryptflow is actually more precise than the type system described in Section 4.4, since it is built on top of a flow-sensitive information-flow analysis. In a flow-sensitive security type system [16], the type of a variable may depend on the program point and is not immutable. We have modified the typing rules presented in Section 4.4 to be flow sensitive.

The flow-sensitive information-flow analysis takes care of tracking information flow through Java language features, including objects, methods, fields, and exceptions. The analysis does not handle reflection or custom class loaders. Cryptflow required approximately 500 lines of Java code to adapt the information-flow analysis to handle the restrictions on public-key cryptography.

Cryptflow visits source files to gather constraints of the form $\sigma \sqsubseteq \sigma'$ and $t = t'$, corresponding to restrictions on the permitted security levels, and types. Once all source files of a program

have been visited, Cryptflow checks whether the set of constraints is solvable, using a standard work-queue algorithm.

Since the partial order of basic types is finite, the constraint solver is guaranteed to terminate, producing either a solution to the constraints, or indicating that no solution is possible. If a solution exists, then the program is secure, and there is a mapping from variables to security levels that does not violate our noninterference condition. If a solution does not exist then there is no mapping from variables to security levels such that noninterference is preserved. The tool either rejects the program, or accepts it. As mentioned previously, because the underlying type system is sound, the tool is conservative and will reject some secure programs.

We have run Cryptflow on several simple Java programs, including programs that use cryptography safely and programs with vulnerabilities such as those of Listings 1.1 and 1.2. Cryptflow correctly rejects the unsafe usages of cryptography, and admits simple programs that use cryptography correctly, such as a program that generates a keypair and correctly performs RSA encryption and decryption.

Our prototype implementation currently relies on the programmer identifying the uses of encryption and decryption. This is done because in the Java Cryptography framework, the method `javax.crypto.Cipher.doFinal` is used to perform both encryption and decryption, based on a flag passed earlier to a the `javax.crypto.Cipher.init` method. We implement this functionality by providing a wrapper around Java's cryptography methods, which separates the encryption and decryption functionalities. *Cryptflow* then makes use of this wrapper when analyzing code that involves encryption and decryption.

```
public static void main(String[] args) throws Exception {
       // Generate the key pair
       KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
       KeyPair key = keyGen.generateKeyPair();
   }
```

Listing 5.1: Safe Key Generation

§5.2 Examples

The code in Listing 5.1 is an example of a very simple program that *Cryptflow* accepts as secure. This program generates a key pair, which consists of a public key and private key. The keys are generated and can later be used for encryption and decryption. *Cryptflow* analyzes the call to `generateKeyPair` and marks the `PrivateKey` field in a `KeyPair` with the security level *privatekey* and marks the `PublicKey` field in a `KeyPair` with the security level *publickey*. *Cryptflow* adds these constraints to its set of generated constraints on the flow of information in this program (constraints are generated by the assignments, the method call, etc). *Cryptflow* then attempts to solve this set of constraints by assigning the variables various security levels while only allowing permitted flows of information. In this situation, because the private key is not output anywhere, *Cryptflow* is able to solve the constraints generated and accepts this program. This program does not violate our security definition, so it is allowed by the type system and accepted by *Cryptflow*.

The code in Listing 5.2 is an example of a program that *Cryptflow* rejects. This code models a situation where a key pair consisting of a public key and a private key is created and then the private key is output to a public channel. The annotation `/* @output "L" */` is used to indicate that `System.out.println` is a low-level channel. Similarly, as in the previous example, *Cryptflow* analyzes the call to `generateKeyPair` and marks the `PrivateKey` field in a `KeyPair` with the security level *privatekey* and marks the `PublicKey` field in a `KeyPair` with the security level *publickey*.

```
public static void main(String[] args) throws Exception {
       // Generate the key pair
       KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
       KeyPair key = keyGen.generateKeyPair();

       // Get the bytes that comprise the private key
       PrivateKey privkey = key.getPrivate();
       byte[] privateKeyBytes = privkey.getEncoded();

       // Print the bytes
       System.out.println(/* @output "L" */("PrivateKey:" + new String(privateKeyBytes)));
   }
```

Listing 5.2: Printing Private Key Bytes

Then, because the `getEncoded()` method is used to extract the bytes of the private key and store them in the byte array `privateKeyBytes`, *Cryptflow* generates constraints that note that a value of security level *privatekey* has moved to a byte array. Finally, *Cryptflow* generates constraints that model that the byte array `privateKeyBytes` is output on the low-channel represented by `System.out.println`. When *Cryptflow* tries to solve these constraints it is not able to, because there is no mapping of variables to security levels that would permit the output of a private key on a low-security channel. This program violates our security definition and does not typecheck. Therefore, it is rejected by *Cryptflow*.

Chapter 6

Conclusion

This thesis presents a possibilistic noninterference condition for public-key cryptography. The security condition defines what it means for public-key cryptography to be secure, and modifies the traditional notion of noninterference to be compatible with the needs of public-key cryptography. This semantic security condition is enforced by a type system which tracks the flow of information through programs that employ cryptographic primitives.

The typing for expressions is completely standard for an imperative language, and most of the typing for the commands is standard with the exception of the commands for input, output, encryption, decryption and key generation. This type system is able to provably prevent the most common cryptographic vulnerabilities present on the Android platform, such as outputs of private keys on public channels and hardcoded cryptographic keys.

Finally, this thesis presents *Cryptflow*, a tool to track the information flow through programs written in Java that employ cryptography. This tool is meant to reject programs with insecure uses of public-key cryptography and accept programs that demonstrate safe uses of public-key cryptography. *Cryptflow* is a static analysis that enforces the main ideas of our type system by rejecting programs that do not comply with our notion of possibilistic noninterference.

Public-key cryptography is essential to a majority of transactions, an increasing number of which are performed on mobile phones due to their increasing popularity. The fact that cryptographic vulnerabilities are a leading cause of security vulnerabilities on the Android Platform suggests that current security guarantees for public-key cryptography are not sufficient. By offering a definition for safe public-key cryptography, a type system that enforces it, and a tool that implements this functionality, this thesis is a step towards providing strong security guarantees for public-key cryptography.

Future Extensions Though *Cryptflow* is successful in rejecting programs which exhibit common security vulnerabilities currently present on the Android Platform, there is more work to be done to enhance its capabilities. First, as mentioned in Section 5.2, *Cryptflow* currently requires minimal annotations by the programmer. This is done because in the Java Cryptography framework, the method `javax.crypto.Cipher.doFinal` is used to perform both encryption and decryption, based on a flag passed earlier to a different method of the object. Future work on this tool will eliminate the need for these programmer annotations by having *Cryptflow* analyze the arguments to the `javax.crypto.Cipher.init` method to determine whether encryption or decryption is being performed.

Further, while *Cryptflow* does reject programs which model the common vulnerabilities which occur on the Android Platform, many more cryptographic vulnerabilities abound. This thesis was only concerned with misuses of public-key cryptography, however, enforcing the correct use of private-key cryptography is also important. A security definition and a type system for private-key cryptography are developed by Askarov, Hedin and Sabelfeld [5]. This type system could be folded into our type system and *Cryptflow* could detect security vulnerabilities in private-key cryptographic protocols, as well as those in public-key cryptographic protocols.

Finally, *Cryptflow* could be extended to detect the appropriate release of information, also called *declassification*, and the release of keys. Askarov and Sabelfeld developed type based enforcement that guarantees security when cryptographic keys are released [3]. The ideas of this type system could be implemented in *Cryptflow*, which would allow *Cryptflow* to detect appropriate and inappropriate releases of information.

Appendix A

Noninterference Proofs

§A.1 Well-Formedness of Types

The well-formedness relation can be found in Figure A.1.

§A.2 Noninterference Proofs

$$\text{WF-INT}\ \frac{}{n : int\ \sigma} \qquad \text{WF-PAIR}\ \frac{v_1 : \tau_1 \quad v_2 : \tau_2}{(v_1, v_2) : (\tau_1, \tau_2)}$$

$$\text{WF-KEY-PRIV}\ \frac{k \in Key_{priv}}{k : privkey} \qquad \text{WF-KEY-PUB}\ \frac{k \in Key_{pub}}{k : pubkey}$$

$$\text{WF-ENC}\ \frac{k : pubkey \quad u \in E(k, v) \quad v : \tau}{u : enc\ \tau\ \sigma} \qquad \text{WF-MEM}\ \frac{\forall x \in dom(\Omega) \quad M(x) : \Omega(x)}{M : \Omega}$$

$$\text{WF-IOENV}\ \frac{\forall ch \in dom(\Theta) \quad X(ch) : \Omega(ch)}{X : \Theta}\ x \in \{I, O\} \qquad \text{WF-STREAM1}\ \frac{}{\epsilon : \tau\ stream} \qquad \text{WF-STREAM2}\ \frac{v : \tau \quad vs : \tau\ stream}{v \cdot vs : \tau\ stream}$$

$$\text{WF-ENV}\ \frac{M : \Omega \quad G : (pubkey, privkey)\ stream \quad I : \Omega \quad O : \Omega}{(M, G, I, O) : (\Omega, \Theta)}$$

Figure A.1: Well Formedness of Types

This theorem states that if command $(\Omega, \Theta), pc \vdash c$, then c is noninterfering with respect to Ω and Θ. We prove this theorem by structural induction on the commands. This theorem relies on two key lemmas, Lemma 1, which proves the noninterference of expressions and Lemma 2, which shows that running a command in a high context renders the resulting environment low equivalent to the initial environment.

Theorem 1 (Type-soundness)**.** *If* $\Omega, \Theta, pc \vdash c$ *and* $\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \hat{E_2}$ *then* $\widehat{[[c]]}\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E_2}$.

Proof. Let $P(c)$ be if $\Omega, \Theta, pc \vdash c$ and $\hat{E_1} \approx_{(\Omega,\Theta)} \hat{E_2}$, then $\widehat{[[c]]}\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E_2}$. We proceed by structural induction on subcommands. If c' is a subcommand of c, then $P(c')$. Let c be given and assume $\Omega, \Theta, pc \vdash c$. Let $\hat{E_1}, \hat{E_2}$ be fixed. Because most of the cases are standard, we present only the cases unique to our type system – the input, output, encrypt, decrypt and newkeypair cases.

1. Case INPUT

   We are given that $\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \hat{E_2}$. By the definition of the $\hat{\approx}_{(\Omega,\Theta)}$ relation we see that $\exists E_1 \in \hat{E_1}$ and $E_2 \in \hat{E_2}$ such that $E_1 \approx_{(\Omega,\Theta)} E_2$. Let $E_1 = (M_1, G_1, I_1, O_1)$ and $E_2 = (M_2, G_2, I_2, O_2)$. Let $E_1' = [[c]]E_1$ and $E_2' = [[c]]E_2$. Let $E_1' = (M_1', G_1', I_1', O_1')$ and (M_2', G_2', I_2', O_2'). Because c is $in(ch, e)$ we see that $G_1' = G_1$ and $O_1' = O_1$. Similarly, we see that $G_2' = G_2$ and $O_2' = O_2$. This means that it suffices to show that $M_1' \approx_\Omega M_2'$ given that $M_1 \approx_\Omega M_2$ and $I_1' \approx_\Omega I_2'$ given that $I_1 \approx_\Theta I_2$.

   By Lemma 2, we see that if $pc = H$ that $E_1 \approx_{(\Omega,\Theta)} E_1'$ and $E_2 \approx_{(\Omega,\Theta)} E_2'$, so we see that $E_1' \approx_{(\Omega,\Theta)} E_2'$ and we are done. We now consider the cases where $pc = L$.

   Given that $\Omega, \Theta, pc \vdash in(x, ch)$, we need to show that $M_1' \approx_\Omega M_2'$. This is equivalent to showing $M_1[x \mapsto v] \approx_\Omega M_2[x \mapsto v]$. By (LE-MEM) and given that $M_1 \approx_\Omega M_2$ it

suffices to show that $M_1(x) \approx_{\Omega(x)} M_2(x)$. By typing, we see that $\Theta(ch) <: \Omega(x)$ where $least(\Theta(ch)) = \sigma$ and $L \sqsubseteq \sigma$. We now consider the two cases of σ.

If $\sigma = H$ then $least(\Theta(ch)) = H$. By subtyping, we see that this implies that $least(\Theta(ch)) \sqsubseteq \Omega(x)$. This implies that $least(\Omega(x)) = H$. By the base cases (LE-INT-H), (LE-ENC-H), and the compound case (LE-PAIR) we see that if $\Omega(x)$ is high then $M_1(x) \approx_{\Omega(x)} M_2(x)$. This means that $M_1' \approx_\Omega M_2'$ and we are done.

We now consider the case if $\sigma = L$. This means that $least(\Theta(ch)) = L$. By subtyping, we see that this implies that $least(\Theta(ch)) \sqsubseteq \Omega(x)$. This implies that $least(\Omega(x)) = L$ or $least(\Omega(x)) = H$. If $least(\Omega(x)) = H$ then by the base cases (LE-INT-H), (LE-ENC-H), and the compound case (LE-PAIR) we see that if $\Omega(x)$ is high then $M_1(x) \approx_{\Omega(x)} M_2(x)$. This means that $M_1' \approx_\Omega M_2'$ and we are done.

If $least(\Omega(x)) = L$, then a low value is input from a channel. We know that $I_1 \approx_\Theta I_2$ so therefore $I_1(ch) \approx_{\Theta(ch)} I_2(ch)$. Because $\Theta(ch)$ is low, from the base cases (LE-INT-L), (LE-ENC-L), and the compound case (LE-PAIR) we can infer that $I_1(ch) = I_2(ch)$. By the semantics, we see that this means that $v_1 = v_2$ where $I_1(ch) = v_1 \cdot vs_1$ and $I_2(ch) = v_2 \cdot vs_2$. By the semantics, we see that $M_1(x) = v_1$ and $M_2(x) = v_2$. This implies that $M_1(x) = M_2(x)$. By definition, this means that $M_1(x) \approx_{\Omega(x)} M_2(x)$. We have now shown that $M_1' \approx_\Omega M_2'$ and we are done.

Now, it remains to show that $I_1' \approx_\Theta I_2'$. This is equivalent to showing $I_1[ch \mapsto vs] \approx_\Theta I_2[ch \mapsto vs]$. By the semantics, we see that prior to the input, $I_1(ch) = v_1 \cdot vs_1$ and $I_2(ch) = v_2 \cdot vs_2$. Because $I_1(ch) \approx_{\Theta(ch)} I_2(ch)$ we know that $v_1 \cdot vs_1 \approx_{\Theta(ch)} v_2 \cdot vs_2$. This means that $vs_1 \approx_{\Theta(ch)} vs_2$ so $I_1[ch \mapsto vs] \approx_\Theta I_2[ch \mapsto vs]$. We have now shown that $I_1' \approx_\Theta I_2'$ and we are done.

We have shown that $M_1' \approx_\Omega M_2'$ and $I_1' \approx_\Theta I_2'$ which implies that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because $E_1 \in \hat{E}_1$ and $E_2 \in \hat{E}_2$ are arbitrary environments that are equivalent under $\approx_{(\Omega,\Theta)}$, we have shown that $\widehat{[[c]]}\hat{E}_1 \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E}_2$.

2. Case OUTPUT

We are given that $\hat{E}_1 \hat{\approx}_{(\Omega,\Theta)} \hat{E}_2$. By the definition of the $\hat{\approx}_{(\Omega,\Theta)}$ relation we see that $\exists E_1 \in \hat{E}_1$ and $E_2 \in \hat{E}_2$ such that $E_1 \approx_{(\Omega,\Theta)} E_2$. Let $E_1 = (M_1, G_1, I_1, O_1)$ and $E_2 = (M_2, G_2, I_2, O_2)$. Let $E_1' = [[c]]E_1$ and $E_2' = [[c]]E_2$. Let $E_1' = (M_1', G_1', I_1', O_1')$ and (M_2', G_2', I_2', O_2'). We want to show that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because c is $out(ch, e)$ we see that $M_1' = M_1$, $G_1' = G_1$ and $I_1' = I_1$. Similarly, we see that $M_2' = M_2$, $G_2' = G_2$ and $I_2' = I_2$. This means that it suffices to show that $O_1' \approx_\Theta O_2'$ given that $O_1 \approx_\Theta O_2$.

By Lemma 2, we see that if $pc = H$ that $E_1 \approx_{(\Omega,\Theta)} E_1'$ and $E_2 \approx_{(\Omega,\Theta)} E_2'$, so we see that $E_1' \approx_{(\Omega,\Theta)} E_2'$ and we are done. We now consider the cases where $pc = L$.

Given that $\Omega, \Theta, pc \vdash out(ch, e)$, we need to show that $O_1' \approx_\Theta O_2'$. This is equivalent to showing $O_1[ch \mapsto v_1 \cdot O_1[ch]] \approx_\Theta O_2[ch \mapsto v_2 \cdot O_2[ch]]$. By (LE-IO) and given that $O_1 \approx_\Theta O_2$ it suffices show that $O_1(ch) \approx_{\Theta(ch)} O_2(ch)$. By typing we see that $\tau^{pc} <: \Theta(ch)$ so we are trying to show that $O_1(ch) \approx_{\tau^{pc}} O_2(ch)$. Because $pc = L$, this is equivalent to showing that $O_1(ch) \approx_\tau O_2(ch)$.

Because $O_1(ch) = v_1 \cdot O_1[ch]$ and $O_2(ch) = v_2 \cdot O_2[ch]$, it suffices to show that $v_1 \approx_\tau v_2$ where $\Omega \vdash e : \tau$. From the semantics we see that $v_1 = M_1(e)$ and $v_2 = M_2(e)$. Due to Lemma 1, because $M_1 \approx_\Omega M_2$, $M_1(e) \approx_\tau M_2(e)$. This implies that $v_1 \approx \tau v_2$ and we are done.

We have shown that $O_1' \approx_\Theta O_2'$ which implies that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because $E_1 \in \hat{E}_1$ and $E_2 \in \hat{E}_2$ are arbitrary environments that are equivalent under $\approx_{(\Omega,\Theta)}$, we have shown that $\widehat{[[c]]}\hat{E}_1 \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E}_2$.

3. Case ENCRYPT

We are given that $\hat{E}_1 \hat{\approx}_{(\Omega,\Theta)} \hat{E}_2$. By the definition of the $\hat{\approx}_{(\Omega,\Theta)}$ relation we see that $\exists E_1 \in \hat{E}_1$ and $E_2 \in \hat{E}_2$ such that $E_1 \approx_{(\Omega,\Theta)} E_2$. Let $E_1 = (M_1, G_1, I_1, O_1)$ and $E_2 = (M_2, G_2, I_2, O_2)$. Let $E_1' = [[c]]E_1$ and $E_2' = [[c]]E_2$. Let $E_1' = (M_1', G_1', I_1', O_1')$ and (M_2', G_2', I_2', O_2'). We want to show that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because c is $x := \text{encrypt}(e_1, e_2)$ we see that $G_1' = G_1$, $I_1' = I_1$ and $O_1' = O_1$. Similarly, we see that $G_2' = G_2$, $I_2' = I_2$ and $O_2' = O_2$. This means that it suffices to show that $M_1' \approx_\Omega M_2'$ given that $M_1 \approx_\Omega M_2$.

By Lemma 2, we see that if $pc = H$ that $E_1 \approx_{(\Omega,\Theta)} E_1'$ and $E_2 \approx_{(\Omega,\Theta)} E_2'$, so we see that $E_1' \approx_{(\Omega,\Theta)} E_2'$ and we are done. We now consider the cases where $pc = L$.

Given that $\Omega, \Theta, pc \vdash x := \text{encrypt}(e_1, e_2)$, we need to show that $M_1' \approx_\Omega M_2'$. This is equivalent to showing $M_1[x \mapsto u_1] \approx_\Omega M_2[x \mapsto u_2]$. By (LE-MEM) and given that $M_1 \approx_\Omega M_2$ it suffices to show that $M_1(x) \approx_{\Omega(x)} M_2(x)$.

We are trying to show that $u_1 \approx_{enc\ \tau\ \sigma} u_2$. By (LE-ENC-H), it is obvious that any two encryptions are equivalent under $\approx_{enc\ \tau\ H}$ so we only consider the case when $\sigma = L$. By (LE-ENC-L), we see that this requires a few conditions. We first define $v_1 = D(k_1, u_1)$ and $v_2 = D(k_2, u_2)$. We know that $k_1 \approx_{privkey} k_2$ because any two private keys are low equivalent. We now need to show that $v_1 \approx_\tau v_2$ where $\Omega \vdash e_2 : \tau$.We know that $u_1 \in E(k', v_1)$ where $k' = M_1(e_1)$ and $v_1 = M_1(e_2)$. Similarly, $u_2 \in E(k'', v_2)$ where $k'' = M_2(e_1)$ and $v_2 = M_2(e_2)$. By Lemma 1, because $M_1 \approx_\Omega M_2$ we see that $M_1(e_2) \approx_\tau M_2(e_2)$ where $\Omega \vdash e_2 : \tau$. Therefore, $v_1 \approx_\tau v_2$.

Finally, we need to show that $u_1 \doteq u_2$. Now we recall the fact that because if $u_1 \in E(k_1, v_1)$ then $\exists u_2' \in E(k_2, v_2)$ such that $u_1 \doteq u_2'$. We cannot claim that $u_2 = u_2'$, but for some n such that $u_2' = M_n'(x)$ this is true. Because the above argument for memory equivalence can be exactly replicated to apply to E_1 and E_n, we see that this means that $E_1' \approx_{(\Omega,\Theta)} E_n'$ where E_n' is an arbitrary environment produced by applying c to E_n which is no different from the initial choice of E_2'.

Because $E_1 \in \hat{E_1}$ and $E_2 \in \hat{E_n}$ are arbitrary environments that are equivalent under $\approx_{(\Omega,\Theta)}$, we have shown that $\widehat{[[c]]}\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E_2}$.

4. Case DECRYPT

We are given that $\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \hat{E_2}$. By the definition of the $\hat{\approx}_{(\Omega,\Theta)}$ relation we see that $\exists E_1 \in \hat{E_1}$ and $E_2 \in \hat{E_2}$ such that $E_1 \approx_{(\Omega,\Theta)} E_2$. Let $E_1 = (M_1, G_1, I_1, O_1)$ and $E_2 = (M_2, G_2, I_2, O_2)$. Let $E_1' = [[c]]E_1$ and $E_2' = [[c]]E_2$. Let $E_1' = (M_1', G_1', I_1', O_1')$ and (M_2', G_2', I_2', O_2'). We want to show that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because c is $x := \text{decrypt}(e_1, e_2)$ we see that $G_1' = G_1$, $I_1' = I_1$ and $O_1' = O_1$. Similarly, we see that $G_2' = G_2$, $I_2' = I_2$ and

$O_2' = O_2$. This means that it suffices to show that $M_1' \approx_\Omega M_2'$ given that $M_1 \approx_\Omega M_2$.

By Lemma 2, we see that if $pc = H$ that $E_1 \approx_{(\Omega,\Theta)} E_1'$ and $E_2 \approx_{(\Omega,\Theta)} E_2'$, so we see that $E_1' \approx_{(\Omega,\Theta)} E_2'$ and we are done. We now consider the cases where $pc = L$.

Given that $\Omega, \Theta, pc \vdash in(x, ch)$, we need to show that $M_1' \approx_\Omega M_2'$. This is equivalent to showing $M_1[x \mapsto v_1] \approx_\Omega M_2[x \mapsto v_2]$. By (LE-MEM) and given that $M_1 \approx_\Omega M_2$ it suffices show that $M_1(x) \approx_{\Omega(x)} M_2(x)$. By typing we see that $\tau^{L \sqcup \sigma} <: \Omega(x)$ where $\Omega \vdash e_2 : enc\ \tau\ \sigma$. We now consider the cases of σ.

If $\sigma = H$ then $\tau^{L \sqcup H} = \tau^H$ and by Lemma 3 $M_1(x) \approx_{\tau^H} M_2(x)$. We know that $M_1(x) : \tau$ and $M_2(x) : \tau$, so by Lemma 4 we see that $M_1(x) \approx_{\Omega(x)} M_2(x)$.

If $\sigma = L$ then $\tau^{L \sqcup L} = \tau^L = \tau$. We now need to show that $v_1 \approx_\tau v_2$. By the semantics, we know that $v_1 = D(k', u_1)$ and $v_2 = D(k'', u_2)$. Because both k' and k'' are private keys we know that $k' \approx_{privkey} k''$. By Lemma 1 we see that because $M_1 \approx M_2$, $M_1(e_2) \approx_{enc\ \tau\ L} M_2(e_2)$ where $\Omega \vdash e_2 : enc\ \tau\ L$. By (LE-ENC-L), we see that $v_1 \approx_\tau v_2$. Given the fact that $\tau <: \Omega(x)$ and by the application of Lemma 4 we see that $v_1 \approx_{\Omega(x)}$ and we are done.

We have shown that $M_1' \approx_\Theta M_2'$ which implies that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because $E_1 \in \hat{E_1}$ and $E_2 \in \hat{E_2}$ are arbitrary environments that are equivalent under $\approx_{(\Omega,\Theta)}$, we have shown that $\widehat{[[c]]}\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E_2}$.

5. Case NEWKEYPAIR

We are given that $\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \hat{E_2}$. By the definition of the $\hat{\approx}_{(\Omega,\Theta)}$ relation we see that $\exists E_1 \in \hat{E_1}$ and $E_2 \in \hat{E_2}$ such that $E_1 \approx_{(\Omega,\Theta)} E_2$. Let $E_1 = (M_1, G_1, I_1, O_1)$ and $E_2 = (M_2, G_2, I_2, O_2)$. Let $E_1' = [[c]]E_1$ and $E_2' = [[c]]E_2$. Let $E_1' = (M_1', G_1', I_1', O_1')$ and (M_2', G_2', I_2', O_2'). Because c is $(x, y) := newkeypair$ we see that $I_1' = I_1$ and $O_1' = O_1$. Similarly, we see that $I_2' = I_2$ and $O_2' = O_2$. This means that it suffices to show that $M_1' \approx_\Omega M_2'$ given that $M_1 \approx_\Omega M_2$ and that $G_1' \approx_\Omega G_2'$ given that $G_1 \approx_{(pubkey,privkey)\ stream)} G_2$.

Given that $\Omega, \Theta, L \vdash (x, y) := newkeypair$, we need to show that $M_1' \approx_\Omega M_2'$. This is equivalent to showing $M_1[x \mapsto k_1, y \mapsto k_2] \approx_\Omega M_2[x \mapsto k_1, y \mapsto k_2]$. By (LE-MEM), and given that $M_1 \approx_\Omega M_2$, it suffices show that $M_1(x) \approx_{\Omega(x)} M_2(x)$ and $M_1(y) \approx_{\Omega(y)} M_2(y)$. From the typing we see that $\Omega(x) = privkey$ and $\Omega(y) = pubkey$. This means that we are trying to show that $M_1(x) \approx_{pubkey} M_2(x)$ and $M_1(y) \approx_{privkey} M_2(y)$. Because $k_1 = M_1(x) = M_2(x)$ we see by (LE-PUBKEY) that $M_1(x) \approx_{pubkey} M_2(x)$. Because $k_1 = M_1(y) = M_2(y)$ and by (LE-PRIVKEY) we see that any two private keys are equivalent under $\approx_{privkey}$, we know that $M_1(y) \approx_{privkey} M_2(y)$. We have now shown that $M_1' \approx M_2'$.

We now show that $G_1' \approx_{(pubkey,privkey)\ stream} G_2'$. Let $G_1 = (k_{11}, k_{21}) \cdot ks_1$ and $G_2 = (k_{12}, k_{22}) \cdot ks_2$. By the semantics of the language, we see that $G_1' = ks_1$ and $G_2' = ks_2$. If $G_1 \approx_{(pubkey,privkey)\ stream} G_2$, then by (LE-STREAM2), we see that $(k_{11}, k_{21}) \approx_{(pubkey,privkey)} (k_{12}, k_{22})$ and $ks_1 \approx_{(pubkey,privkey)\ stream} ks_2$. We have now shown that $G_1' \approx_{(pubkey,privkey)\ stream} G_2'$.

We have shown that $M_1' \approx_\Omega M_2'$ and $G_1' \approx_{(pubkey,privkey)\ stream} G_2'$ which implies that $E_1' \approx_{(\Omega,\Theta)} E_2'$. Because $E_1 \in \hat{E_1}$ and $E_2 \in \hat{E_2}$ are arbitrary environments that are low equivalent, we have shown that $\widehat{[[c]]}\hat{E_1} \hat{\approx}_{(\Omega,\Theta)} \widehat{[[c]]}\hat{E_2}$.

□

This lemma states that if an expression is well typed and if two memories are low equivalent, then the evaluations of that expression in the two memories must be low equivalent with respect to its type. We prove this lemma by structural induction on the type of the expression.

Lemma 1. *If* $\Omega \vdash e : \tau \wedge M_1 \approx_\Omega M_2$ *then* $M_1(e) \approx_\tau M_2(e)$.

Proof. Let the given M_1, M_2 be such that $M_1 \approx_\Omega M_2$. We proceed by induction on the

derivation of $\Omega \vdash e : \tau$. Let $P(\Omega \vdash e : \tau) =$ if $\Omega \vdash e : \tau$ and $M_1 \approx_\Omega M_2$ then$M_1(e) \approx_\tau M_2(e)$. The inductive hypothesis is if $\Omega' \vdash e' : \tau'$ is a subderivation of the derivation for $\Omega \vdash e : \tau$ then $P(\Omega' \vdash e' : \tau')$. Because the expressions in this language are completely standard, the complete proof is omitted. □

This lemma states that if a command is run in a high context, then the resulting environment is low equivalent to the initial environment. By proving that an environment produced by running a command in a high context is low equivalent to the original environment, we lift this notion to sets of environments and prove our noninterference condition. We prove this lemma by structural induction on the commands.

Lemma 2. *If* $\Omega, \Theta, H \vdash c$ *then* $\forall E, E' \;.\; E' \in [[c]]E \;.\; E' \approx_{(\Omega,\Theta)} E$.

Proof. Let $P(c)$ be if $\Omega, \Theta, H \vdash c$, then $\forall E, E' \;.\; E' \in [[c]]E \;.\; E' \approx_{(\Omega,\Theta)} E$. We proceed by structural induction on subcommands. If c' is a subcommand of c, then $P(c')$. Let c be given and assume $\Omega, \Theta, H \vdash c$. Let E be fixed. Because most of the cases are standard, we present only the cases unique to our type system – the input, output, encrypt, decrypt and newkeypair cases.

1. Case INPUT

   Let $c = in(x, ch)$. Here, let $E = (M, G, I, O)$ and $[[c]](M, G, I, O) = \{(M[x \mapsto v], G, I[ch \mapsto vs], O)\}$ where $I(ch) = v \cdot vs$. Let $E' = (M[x \mapsto v], G, I[ch \mapsto vs], O)$. We want to show that $E \approx_{(\Omega,\Theta)} E'$. Because G and O are unchanged, it suffices to show that $M \approx_\Omega M[x \mapsto v]$ and $I \approx_\Theta I[ch \mapsto vs]$. Let $M' = M[x \mapsto v]$ and $I' = I[ch \mapsto vs]$. We first show that $M \approx_\Omega M'$ and then we show that $I \approx_\Theta I$'.

   We know that for all variables y, such that $y \neq x$, $M(y) = M'(y)$. By (LE-MEM), for $M \approx_\Omega M'$ to hold, we need to show that $\forall x \in Dom(\Omega)$, $M(x) \approx_{\Omega(x)} M'(x)$. Assume

$Dom(\Omega) = \{x\} \uplus V$ where V is the set of all channel names with the exception of x. We have $\forall y \in V$, $M(y) = M'(y)$ which implies that $M(y) \approx_\tau M'(y)$ where $\tau = \Omega(y)$. Therefore, in order to show that $\forall x \in Dom(\Omega)$, $M(x) \approx_{\Omega(x)} M'(x)$ we only need to show that $M(x) \approx_{\Omega(x)} M'(x)$.

Let $v_1 = M(x)$ and $v_2 = M'(x)$. By the typing, we see that $\Theta(ch)^H <: \Omega(x)$. This means that for $\tau = \Theta(ch)$ we are trying to show that $v_1 \approx_{\tau^H} v_2$. We know that $v_1 : \tau$ and $v_2 : \tau$. By Lemma 3, we see this is the case and we have shown that $M(x) \approx_{\Omega(x)} M'(x)$ and we are done with the first part of the proof.

We know that for all channel names ch', such that $ch' \neq ch$, $I(ch') = I'(ch')$. By (LE-IOENV), for $I \approx_\Theta I'$ to hold, we need to show that $\forall ch \in Dom(\Theta)$, $I(ch) \approx_{\Theta(ch)} I'(ch)$. Assume $Dom(\Theta) = \{ch\} \uplus Ch$ where Ch is the set of all channel names with the exception of ch. We have $\forall ch' \in V$, $I(ch') = I'(ch')$ which implies that $I(ch') \approx_\tau I'(ch')$ where $\tau = \Theta(ch')$. Therefore, in order to show that $\forall ch \in Dom(\Theta)$, $I \approx_\Theta I'$ we only need to show that $I(ch) \approx_{\Theta(ch)} I'(ch)$.

Let $v_1 = I(ch)$ and $v_2 = I'(ch)$. By the typing, we see that $\Theta(ch) <: \Omega(x)$ where $least(\Theta(ch)) = \sigma$ and $pc \sqsubseteq \sigma$. We know that $pc = H$ so $\sigma = H$. Therefore, $least(\Theta(ch)) = H$. We now perform an induction on the derivation of the *least* function by considering the cases of $\Theta(ch)$.

If $\Theta(ch) = int\ \sigma$ then $\sigma = H$. In this case we want to show that $v_1 \approx_{int\ H} v_2$. By (LE-INT-H), we see that any two integers are high equivalent, so we know that $v_1 \approx_{\Theta(ch)} v_2$. Consequently $v_1 \approx_{\Omega(x)} v_2$ and we are done. If $\Theta(ch) = enc\ \tau\ \sigma$ then $least(\tau) \sqcap \sigma = H$. This means that $least(\tau) = H$ and $\sigma = H$. In this case we want to show that $v_1 \approx_{enc\ \tau\ H} v_2$. By (LE-ENC-H), we see that any two ciphertexts are high equivalent, so we know that $v_1 \approx_{\Theta(ch)} v_2$. Consequently $v_1 \approx_{\Omega(x)} v_2$ and we are done.

We now consider the inductive case where $\Theta(ch) = (\tau_1, \tau_2)$. In this case, $least(\tau_1, \tau_2) = least(\tau_1) \sqcap least(\tau_2)$. This means that $least(\tau_1) = H$ and $least(\tau_2) = H$. In this case we want to show that $v_1 \approx_{(\tau_1,\tau_2)} v_2$ where $v_1 = (v_{11}, v_{21})$ and $v_2 = (v_{21}, v_{22})$. By induction

we know that if $least(\tau_1) = H$ and $least(\tau_2) = H$ then $v_{11} \approx_{\tau_1} v_{21}$ and $v_{21} \approx_{\tau_2} v_{22}$. By (LE-PAIR) this means that $v_1 \approx_{(\tau_1,\tau_2)} v_2$. Consequently $v_1 \approx_{\Omega(x)} v_2$ and we are done.

2. Case OUTPUT

   Let $c = out(ch, e)$. Here, let $E = (M, G, I, O)$ and $[[c]](M, G, I, O) = \{(M, G, I, O[ch \mapsto v \cdot O[ch]])\}$ where $v = M(e)$. Let $E' = (M, G, I, O[ch \mapsto v \cdot O[ch])$. We want to show that $E \approx_{(\Omega,\Theta)} E'$. Because G and I are unchanged, it suffices to show that $O \approx_\Theta O[ch \mapsto v \cdot O[ch]]$. Let $O' = O[ch \mapsto v \cdot O(ch)]$. We first show that $M \approx_\Omega M'$ and then we show that $O \approx_\Theta O$'.

   We know that for all channel names ch', such that $ch' \neq ch$, $O(ch') = O'(ch')$. By (LE-IOENV), for $O \approx_\Theta O'$ to hold, we need to show that $\forall ch \in Dom(\Theta)$, $O(ch) \approx_{\Theta(ch)} O'(ch)$. Assume $Dom(\Theta) = \{ch\} \uplus Ch$ where Ch is the set of channel names with the exception of ch. We have $\forall ch' \in V, O(ch') = O'(ch')$ which implies that $O(ch') \approx_\tau O'(ch')$ where $\tau = \Theta(ch')$. Therefore, in order to show that $\forall ch \in Dom(\Theta)$, $O(ch) \approx_{\Theta(ch)} O'(ch)$ we only need to show that $O(ch) \approx_{\Theta(ch)} O'(ch)$.

   Let $v_1 = O(ch)$ and $v_2 = O'(ch)$. By the typing, we see that $\tau^H <: \Omega(x)$ where $\Omega \vdash e : \tau$. We are trying to show that $v_1 \approx_{\tau^H} v_2$. By Lemma 3, we see this is the case and we have shown that $O(ch) \approx_{\Theta(ch)} O'(ch)$ and we are done with the proof.

3. Case ENCRYPT

   Let $c = x := \text{encrypt}(e_1, e_2)$. Here, let $E = (M, G, I, O)$ and $[[c]](M, G, I, O) = \{(M[x \mapsto u], G, I, O)\}$ where $M(e_1) = k$, $k \in Key_{pub}$, $M(e_2) = v$ and $u \in E(k, v)$. Let $E' = (M[x \mapsto v], G, I, O)$. We want to show that $E \approx_{(\Omega,\Theta)} E'$. Because G, I, O are unchanged, it suffices to show that $M \approx_\Omega M[x \mapsto v]$. Let $M' = M[x \mapsto v]$. We now show that $M \approx_\Omega M'$.

   We know that for all variables y, such that $y \neq x$, $M(y) = M'(y)$. By (LE-MEM), for $M \approx_\Omega M'$ to hold, we need to show that $\forall x \in Dom(\Omega)$, $M(x) \approx_{\Omega(x)} M'(x)$. Assume $Dom(\Omega) = \{x\} \uplus V$ where V is the set of all variables with the exception of x. We have $\forall y \in V, M(y) = M'(y)$ which implies that $M(y) \approx_\tau M'(y)$ where $\tau = \Omega(y)$. Therefore,

in order to show that $\forall x \in Dom(\Omega)$, $M(x) \approx_{\Omega(x)} M'(x)$ we only need to show that $M(x) \approx_{\Omega(x)} M'(x)$.

Let $v_1 = M(x)$ and $v_2 = M'(x)$. Due to the judgment $\Omega \vdash e_2 : \tau$ we know that $\tau^H <: \Omega(x)$. This means that we are trying to show that $v_1 \approx_{\tau^H} v_2$. We know that $v_1 : \tau$ and $v_2 : \tau$. By Lemma 3, we see this is the case and we have shown that $M(x) \approx_{\Omega(x)} M'(x)$ and we are done.

4. Case DECRYPT

   Let $c = x := \text{decrypt}(e_1, e_2)$. Here, let $E = (M, G, I, O)$ and $[[c]](M, G, I, O) = \{(M[x \mapsto v], G, I, O)\}$ where $M(e_1) = k$, $k \in Key_{priv}$, $M(e_2) = u$ and $v = D(k, y)$. Let $E' = (M[x \mapsto v], G, I, O)$. We want to show that $E \approx_{(\Omega,\Theta)} E'$. Because G, I, O are unchanged, it suffices to show that $M \approx_\Omega M[x \mapsto v]$. Let $M' = M[x \mapsto v]$. We now show that $M \approx_\Omega M'$.

   We know that for all variables y, such that $y \neq x$, $M(y) = M'(y)$. By (LE-MEM), for $M \approx_\Omega M'$ to hold, we need to show that $\forall x \in Dom(\Omega)$, $M(x) \approx_{\Omega(x)} M'(x)$. Assume $Dom(\Omega) = \{x\} \uplus V$ where V is the set of all variables with the exception of x. We have $\forall y \in V$, $M(y) = M'(y)$ which implies that $M(y) \approx_\tau M'(y)$ where $\tau = \Omega(y)$. Therefore, in order to show that $\forall x \in Dom(\Omega)$, $M(x) \approx_{\Omega(x)} M'(x)$ we only need to show that $M(x) \approx_{\Omega(x)} M'(x)$.

   By the typing, we see that $\tau^{pc \sqcup \sigma} <: \Omega(x)$ where $\Omega \vdash e_2 : enc\ \tau\ \sigma$. Because $pc = H$, $pc \sqcup \sigma = H$. Therefore, we need to show that $M(x) \approx_{\tau^H} M'(x)$. Let $v_1 = M(x)$ and $v_2 = M'(x)$. This means that we are trying to show that $v_1 \approx_{\tau^H} v_2$. We know that $v_1 : \tau$ and $v_2 : \tau$. By Lemma 3, we see this is the case and we have shown that $M(x) \approx_{\Omega(x)} M'(x)$ and we are done.

5. Case NEWKEYPAIR

   Let $c = (x, y) := newkeypair$. By typing, we see that $(\Omega, \Theta), L \vdash (x, y) := \text{newkeypair}$ and because $pc = H$, this case is impossible in this context.

□

Lemma 1 and Lemma 2 rely on two lemmas.

Lemma 3. *Given,* v_1, v_2, τ *such that* $v_1 : \tau$*, and* $v_2 : \tau$ *and* τ^H *is defined then* $v_1 \approx_{\tau^H} v_2$*.*

Proof. We proceed by induction on the structure of v_1 and v_2. □

Lemma 4. *Given,* v_1, v_2, τ *such that* $v_1 : \tau$*,* $v_2 : \tau$*,* $\tau <: \tau'$ *and* $v_1 \approx_\tau v_2$*, then* $v_1 \approx_{\tau'} v_2$*.*

Proof. We proceed by induction on the structure of v_1 and v_2. □

Bibliography

[1] Martín Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, September 1999.

[2] Chloe Albanesius. Almost half of u.s. smartphones running android, March 2012.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 207–221, 2007.

[4] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, Piscataway, NJ, USA, June 2012. IEEE Press.

[5] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3):82–101, July 2008.

[6] Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221. IEEE Computer Society, 2007.

[7] Stephen Chong, Andrew Johnson, Scott Moore, and Owen Arden. Accrue ObjAnal, 2013. `http://people.seas.harvard.edu/~chong/accrue.html`.

[8] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[9] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[10] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[11] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 341–350, New York, NY, USA, 2011. ACM.

[12] Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 351–360, New York, NY, USA, 2011. ACM.

[13] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 323–335, New York, NY, USA, 2008. ACM.

[14] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

[15] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[16] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 79–90, New York, NY, USA, January 2006. ACM Press.

[17] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of java-like programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th.* IEEE Computer Society, June 2012.

[18] Peeter Laud. Handling encryption in analyses for secure information flow. In *Proceedings of the 12th European Symposium on Programming*, pages 159–173. Springer, 2001.

[19] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 77–91, London, UK, UK, 2001. Springer-Verlag.

[20] Peeter Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 337–348, New York, NY, USA, 2008. ACM.

[21] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *12th International Conference on Compiler Construction*, pages 138–152, 2003.

[22] Don Reisinger. In-stat: Majority in u.s. to have smartphones, August 2011.

[23] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[24] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, June 2005.

[25] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *Proceedings of the Fourth ACM Workshop on Formal Methods in Security*, pages 33–44, New York, NY, USA, 2006. ACM.

[26] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 192–206, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Veracode. State of software security report: The intractable problem of insecure software, December 2011.