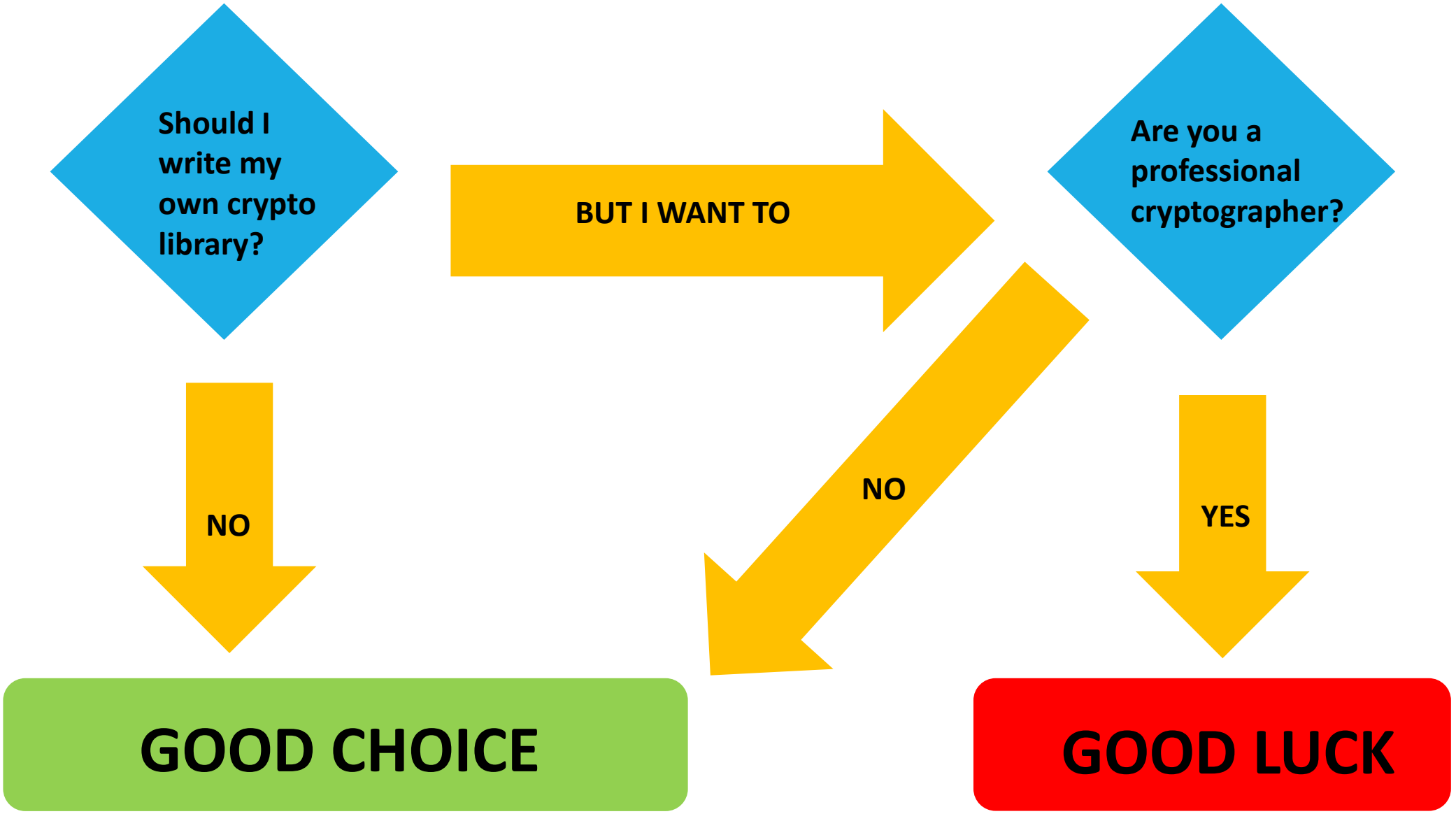
Fighting Cryptographic Misuse with Types

HANNAH (ANNA) GOMMERSTADT

CARNEGIE MELLON UNIVERSITY


Should I write my own crypto library?
BUT I WANT TO
Are you a professional cryptographer?
NO
NO
YES
GOOD CHOICE
GOOD LUCK

Reality

It turns out we are really bad at using cryptographic libraries too.

Case Study: Android Platform

The Android Platform occupies a major share of the American smart phone market

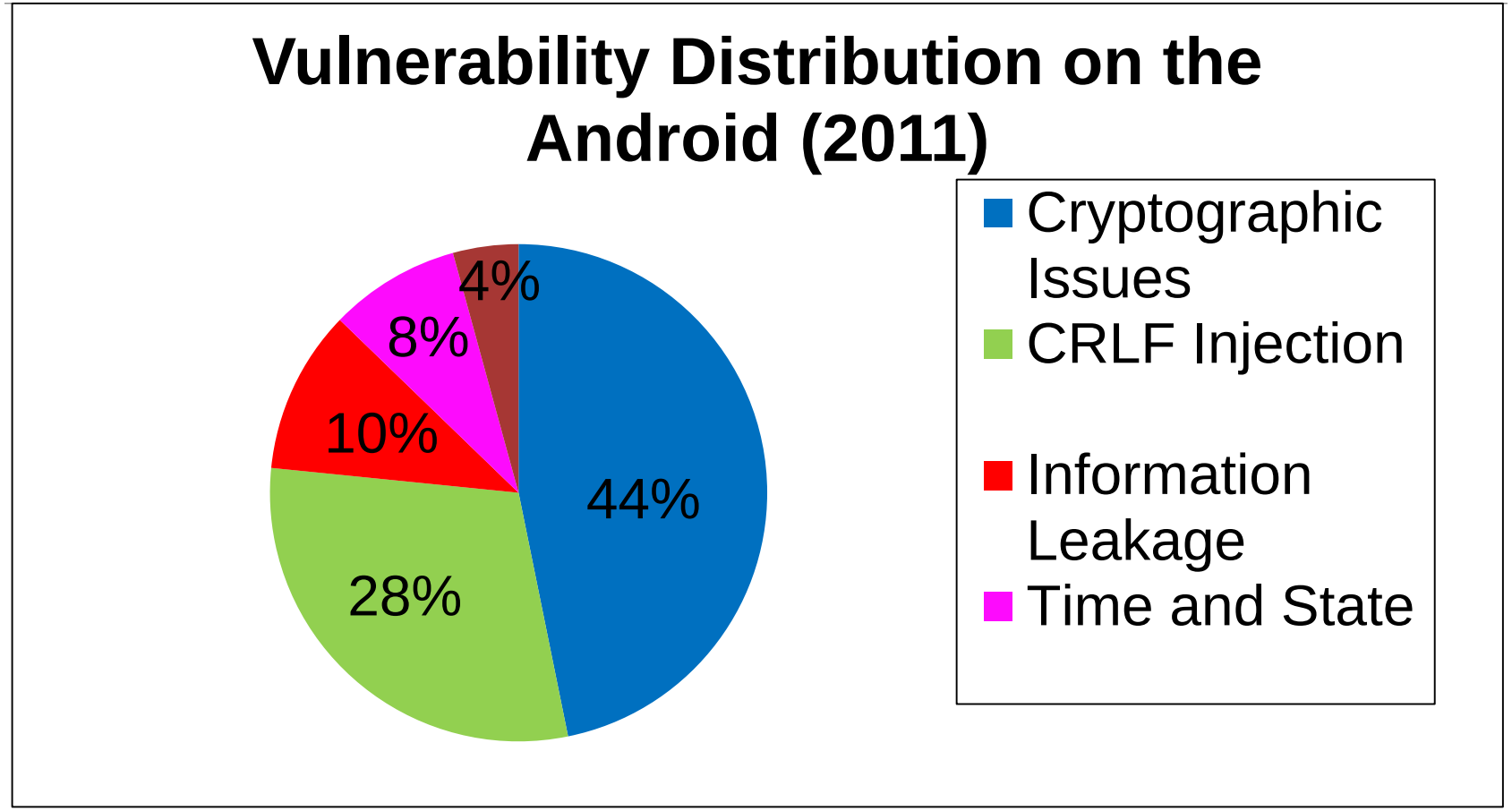
The majority of applications on Google Play make use of cryptographic functions

Android applications are trivial to decompile back to Java and therefore we can run many existing analyses on them

Terrible Fact

Out of 10,000 analyzed Android applications, 40% were found to have a hard coded cryptographic key (2011).

Terrible Chart



It Does Not Get Better

10,327 out of 11,748 applications (88%) in the Google Play marketplace, that use cryptographic APIs make at least one mistake (2013).

What Kinds of Bugs?

Hardcoded cryptographic keys

Outputs of secure keys to public channels

Non-random initial vectors

Weakly chosen keys

Reused initial vectors


EVERYTHING
IS TERRIBLE
memegenerator.net

Motivation

It is necessary to guarantee and enforce the security of public-key cryptographic protocols, especially on the Android platform

Assumptions

We assume that mainstream 3rd party libraries provide **correct** implementations of common algorithms (RSA, etc)

We are concerned with misuses of **correctly implemented** cryptographic protocols

This work focuses on the security of **public-key** cryptographic protocols

Theoretical Approach

Using ideas from **information flow** we define semantic security condition for public key cryptography (possibilistic noninterference)

We develop an enforcement mechanism (type system) and show that it provably enforces the security condition.

Information Flow

var **low** = **high**

explicit flow

```
if (high  == 7)
        var low = 5
else
        var low = -7
```

implicit flow

Noninterference

High-security information cannot observably (to an attacker) influence low-security information

Standard noninterference does not work for public-key cryptography because the low-security ciphertext is influenced by the high-security plaintext

Possibilistic Noninterference

A modified notion of traditional noninterference

We want the ciphertext to possibly be **any** value so that a change in the high-security plaintext does not affect the low-security ciphertext

Based on work by Askarov et al (2008) for private-key cryptography

The Language

Using a simple imperative language with encryption, decryption and key generation commands

Most of the semantics and typing are standard, except for encryption, decryption, input/output channels and key generation

Encryption Semantics

Encryption is nondeterministic

Encrypting a plaintext with a specific key can generate a set of possible ciphertexts.

This nondeterminism is essential for our notion of possibilistic noninterference

Decryption Semantics

Decryption is deterministic

A ciphertext and a key have one possible decryption

Examples: El Gamal, nondeterministic variant of RSA

Channels

We provide input and output channels with a lot of structure

Can output public and private keys on dedicated channels without risking the output of private keys on public channels

Key Generation

Key generation only occurs in a low context, so that the public key cannot be influenced by a high context

Secure key generation will be expanded to also include secure key storage

Proofs

Proved that type system is sound

Proved that a well typed program adheres to possibilistic noninterference

Next Steps

Encoding a secure key store into the type system

Using nonces to formalize initialization vectors

Encodings of integrity policies

Cryptflow Framework

Can analyze snippets of code that misuse cryptography and identify simple vulnerabilities

Built on top of the Polyglot and Objanal frameworks

Performs a flow sensitive information flow analysis of Java code

Code Example

```
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)));
    }
```

More Details

Full thesis (with proofs!) on my website:
http://anyag.net/papers/thesis.pdf