RSA Conference 2015
San Francisco | April 20-24 | Moscone Center

SESSION ID: MBS-W04

Analysis of SSL and Crypto Vulnerabilities in Android Applications

Adrian Mettler
Staff Software Development Engineer
FireEye, Inc.

Yulong Zhang
Senior Software Research Engineer
FireEye, Inc.

Outline

- Introduction
- Part I: Transport Security
  - How SSL/TLS works, and how it can fail to work if used incorrectly
  - Usage in Android and possible mistakes
  - Attack demo and vulnerability statistics
- Part II: Data Encryption
  - Primitives and security properties
  - Android capabilities and mistakes
  - Attack demo and vulnerability statistics
- Summary and Recommendations

Introduction

We identified common Android cryptographic security vulnerabilities

- Options that can “fix” SSL/TLS problems but break security
- Insecure usage of cryptographic operations such as random number generation and block ciphers

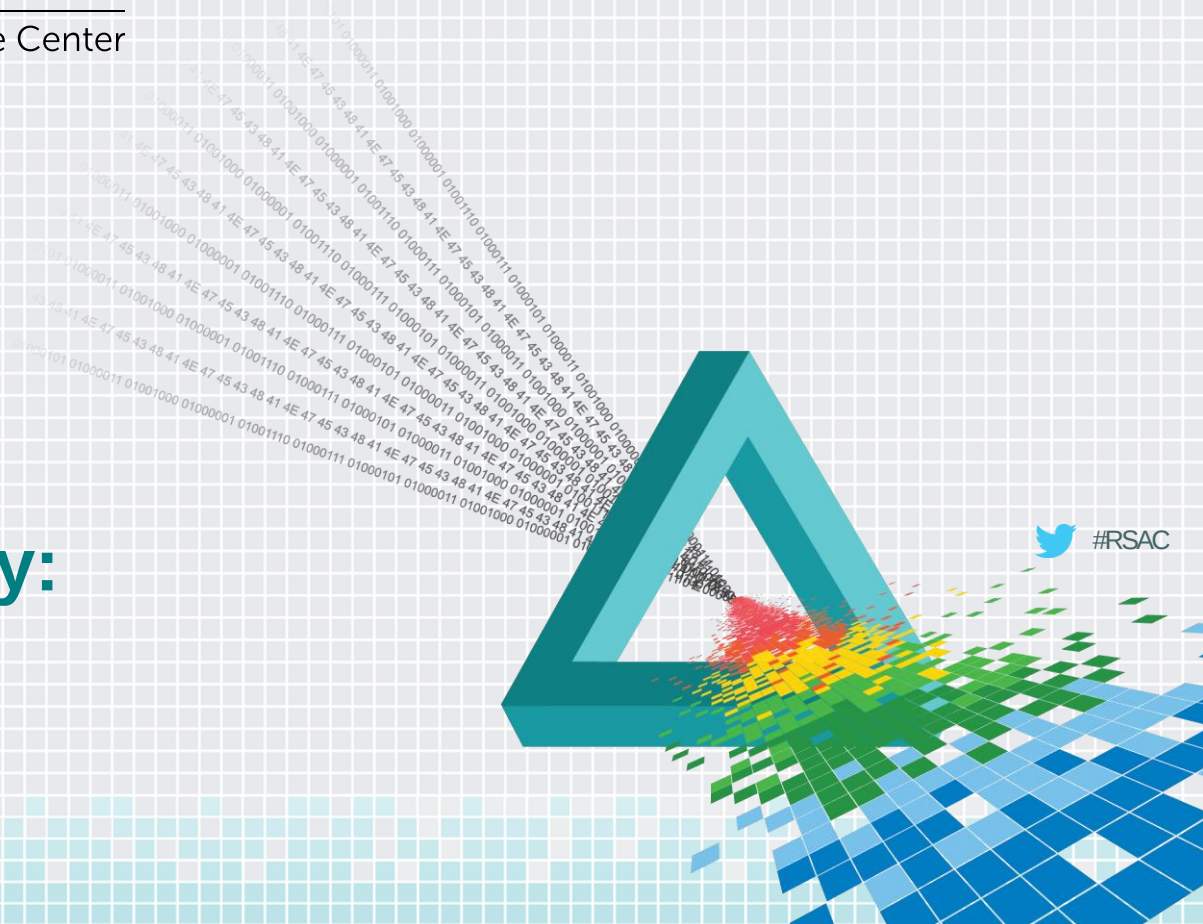
And then analyzed a large data set of common applications

- All free applications in Google Play with over a million downloads
- The total number of such applications as of Apr. 8, 2015 is 11,113

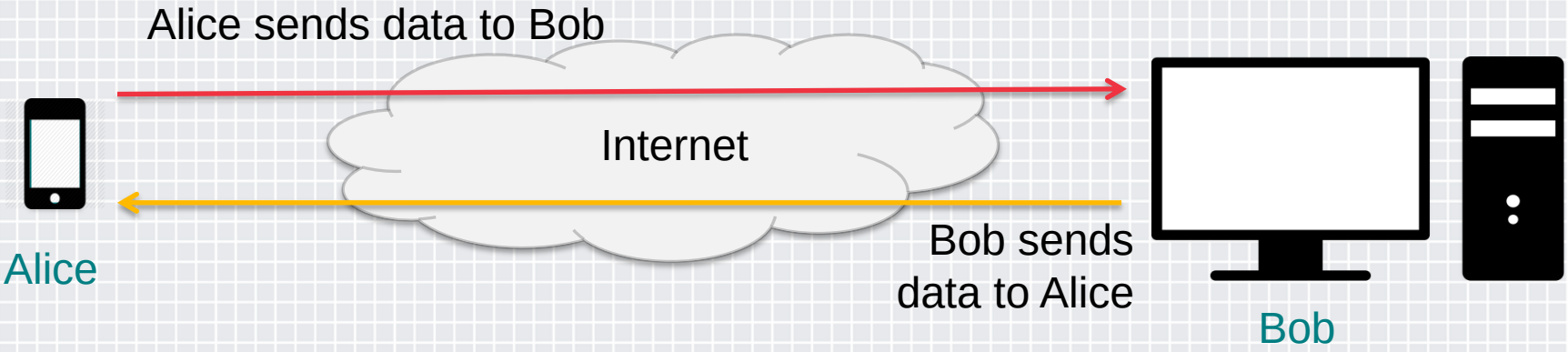
RSAConference2015

San Francisco | April 20-24 | Moscone Center

Transport Security: SSL/TLS

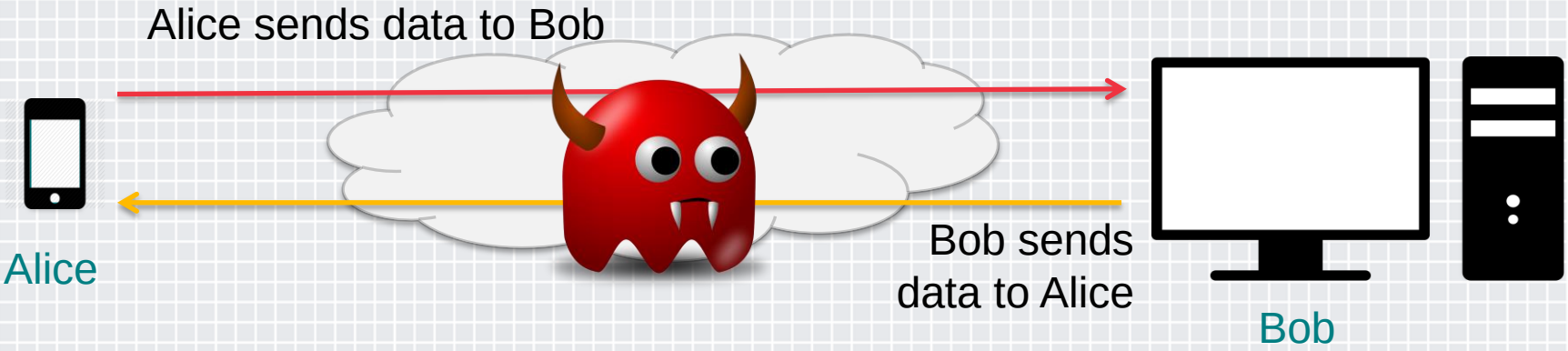


Part I. Securing communications channels



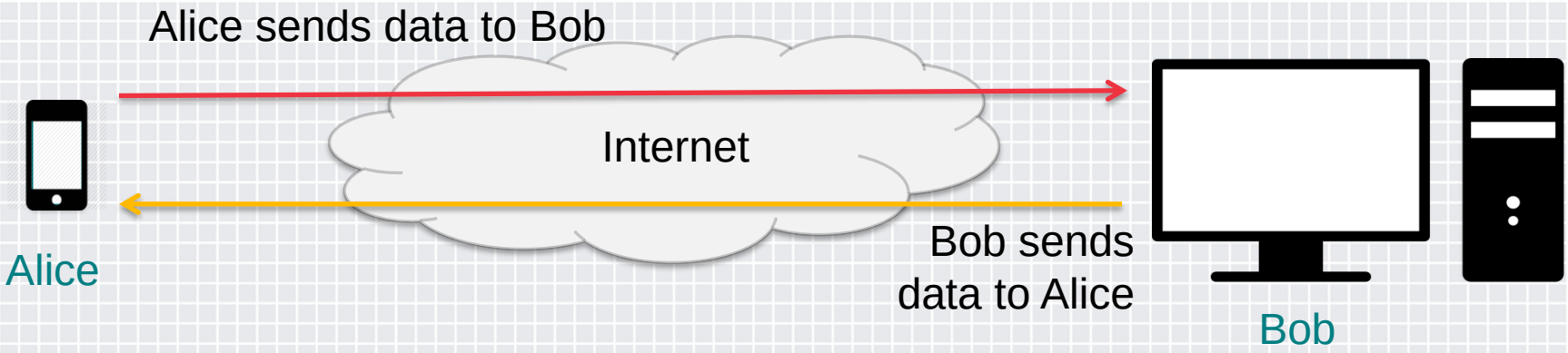
Apps need to securely communicate with their backend servers. This can include sensitive user data. What can go wrong?

Part I. Securing communications channels



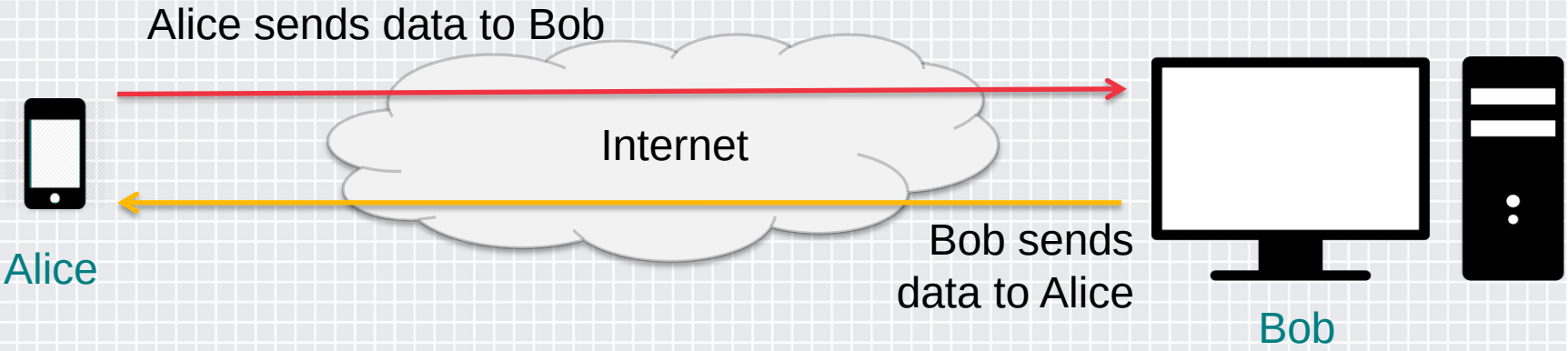
If the data being sent is not encrypted, anyone can read it.
And they can also modify it by acting as a “Man in the Middle”, intercepting and replacing traffic as they see fit.

Part I. The SSL/TLS protocols



Encryption can ensure that only the endpoints can read or write the data transferred over the connection. So let's do that.

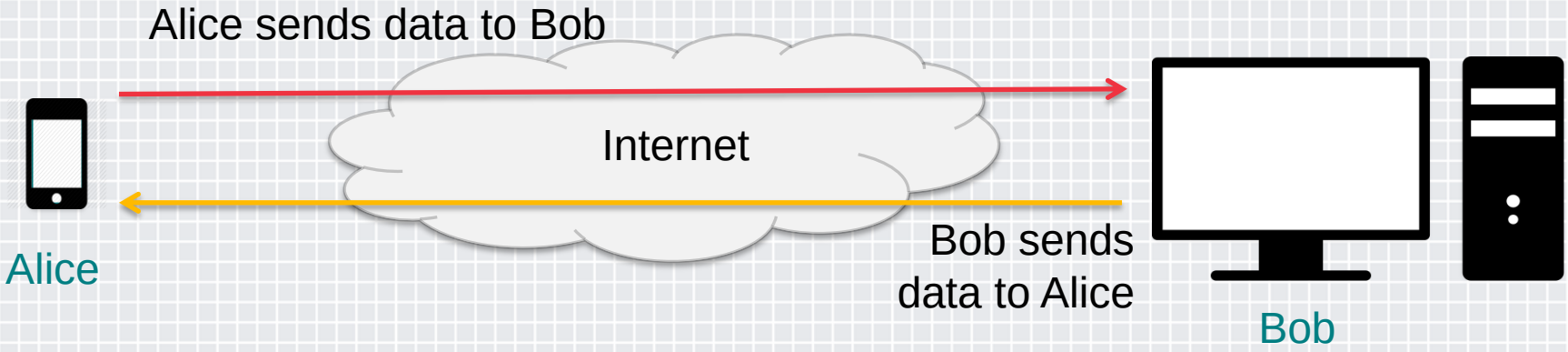
Part I. The SSL/TLS protocols



But how does Alice know that she is talking to the real Bob?

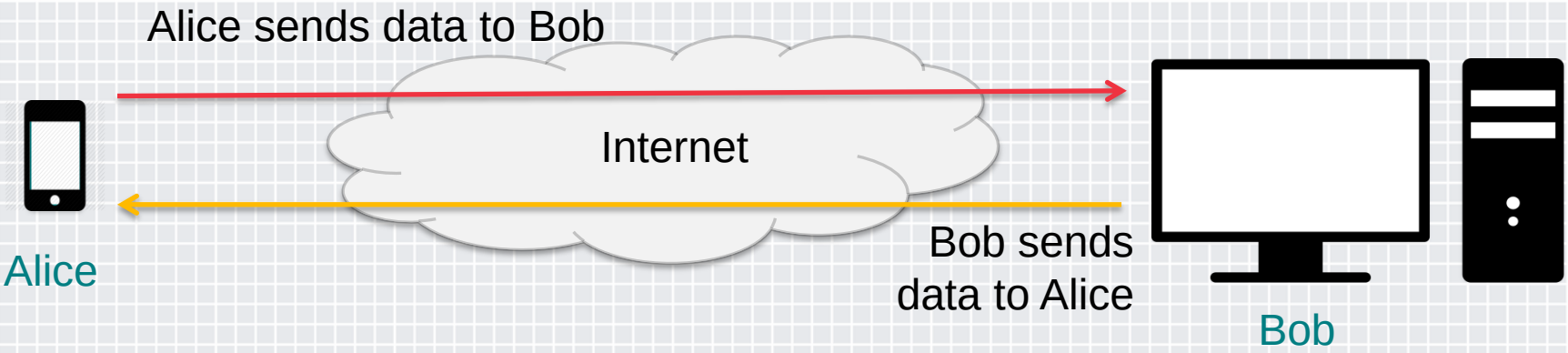
This is a job for PKI.

Part I. The Public-Key Infrastructure



But how does Alice know that she is talking to the real Bob?
Because he has the right public key.
But how does she know Bob's public key?
A trusted third party (TTP) vouches for his identity.

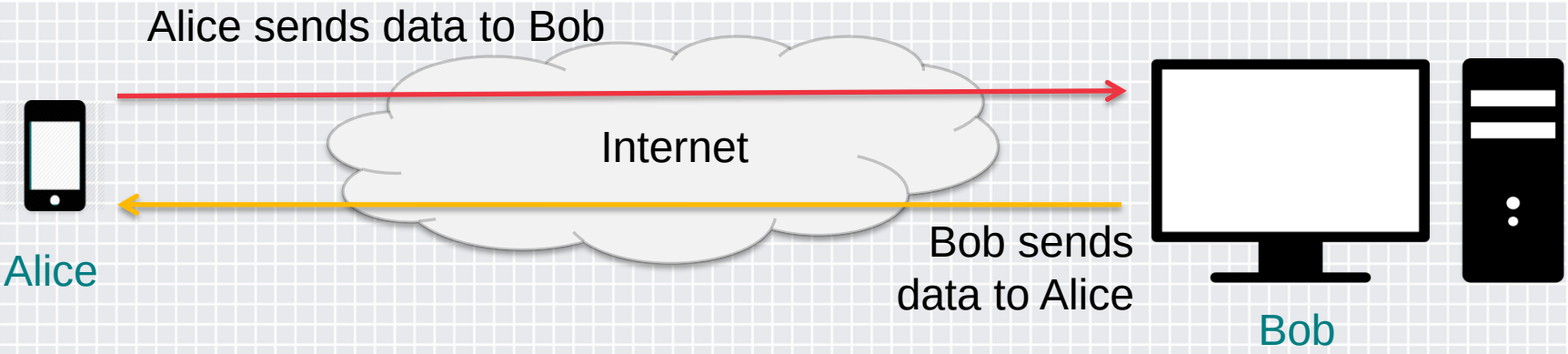
Part I. The Public-Key Infrastructure



Identity verification using a TTP:

- Bob creates an unsigned certificate that specifies his public key
- After verifying Bob's identity, TTP signs the certificate using its private key
- When Alice connects, Bob sends the signed certificate to prove his identity.
- Alice verifies the TTP's signature and that Bob is using the specified key

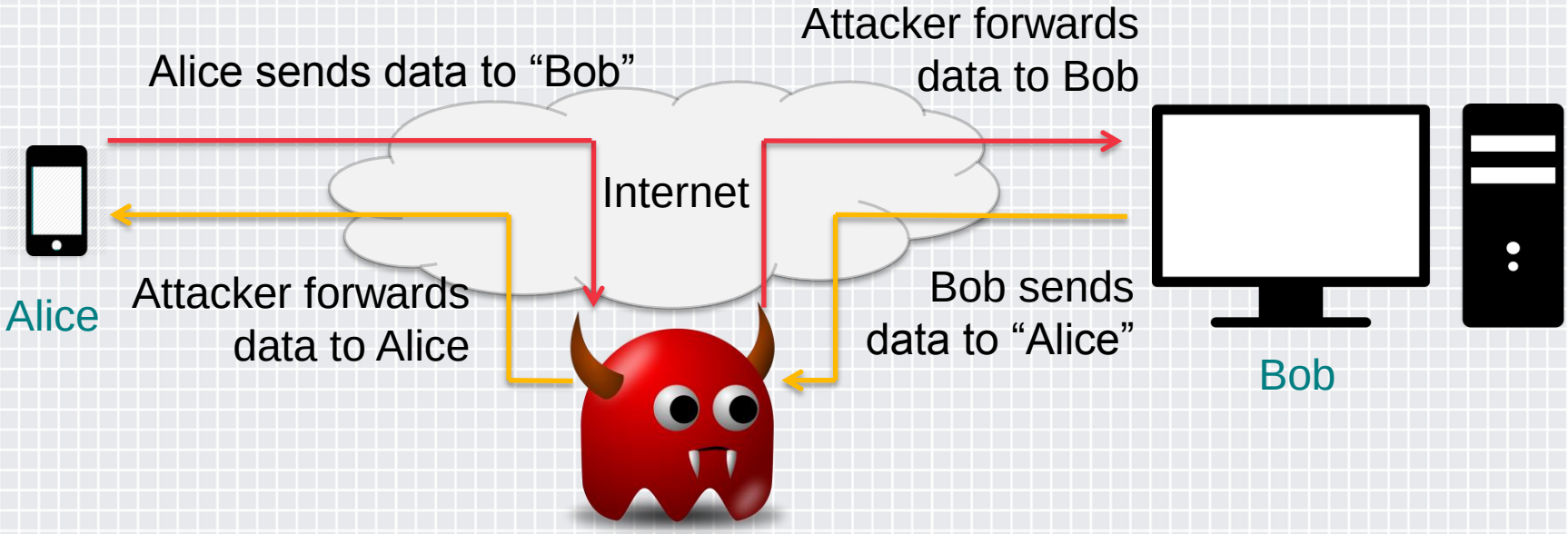
Part I. The Public-Key Infrastructure



Note that Alice must:

- Check that the presented certificate is in fact for Bob, and not Charlie
- Validate that the certificate is properly signed by a TTP

Part I. The Public-Key Infrastructure



A “Man-in-the-Middle” Attacker able to impersonate Bob again gets full access to the communication

Part I. The Public-Key Infrastructure. MITM

Demo…

Part I. Android (Java) capabilities

Trust management for SSL/TLS

```
javax.net.ssl.SSLContext
   .init(KeyManager[], TrustManager[], SecureRandom)
```

The TrustManager parameter allows a custom class to be used for verifying the server's certificate. This has some legitimate uses, e.g. for "pinning" a specific trusted certificate.

Problem: Defining a custom TrustManager that is too accepting (e.g., takes any certificate) to work around SSL errors. This allows an attacker to use any certificate (even self-signed) to perform a man-in-the-middle attack.

Solution: Try to avoid overriding this option. Scrutinize custom Trust Managers carefully to ensure that they perform sufficient checks or are only used in debug/testing mode.

Part I. Android (Java) capabilities

Trust management for SSL/TLS

```
javax.net.ssl.HttpsURLConnection
   .setHostnameVerifier(HostnameVerifier v)
```

Specifies a custom hostname verifier that can permit certificate hostnames that do not match the DNS host name

Problem: Defining a custom HostnameVerifier that is too accepting (e.g. of any hostname) to work around SSL errors. This allows the attacker to use a valid certificate for his own domain to become a man-in-the-middle.

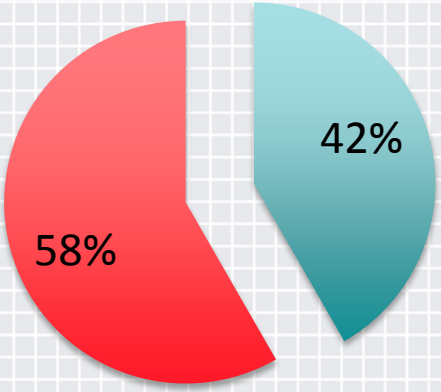
Solution: Try to avoid overriding this option. Scrutinize custom Hostname Verifiers carefully to ensure that they perform sufficient checks or are only used in debug/testing mode.

Automated vulnerability detection for apps

Android code analysis engine

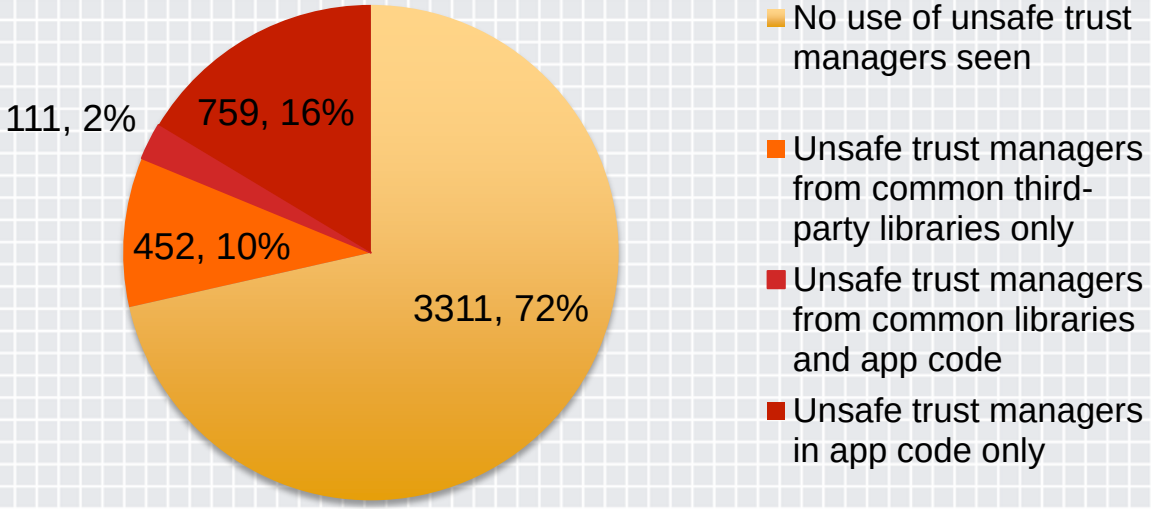
- Abstract interpretation-based dataflow analysis on Android apps
- Dalvik bytecode translated to intermediate language for analysis
- Reachability of Hostname Verifiers and Trust Managers verified by dynamic analysis

Part I. Statistics



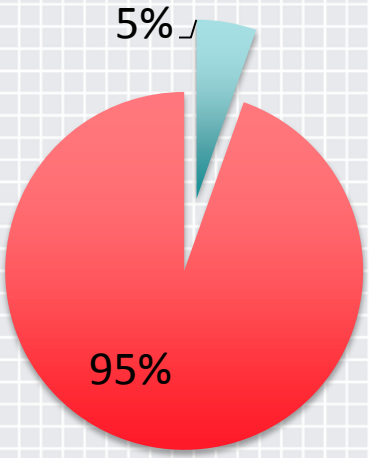
The stats are for the 42% of applications that use SSL/TLS

Trust managers in SSL/TLS

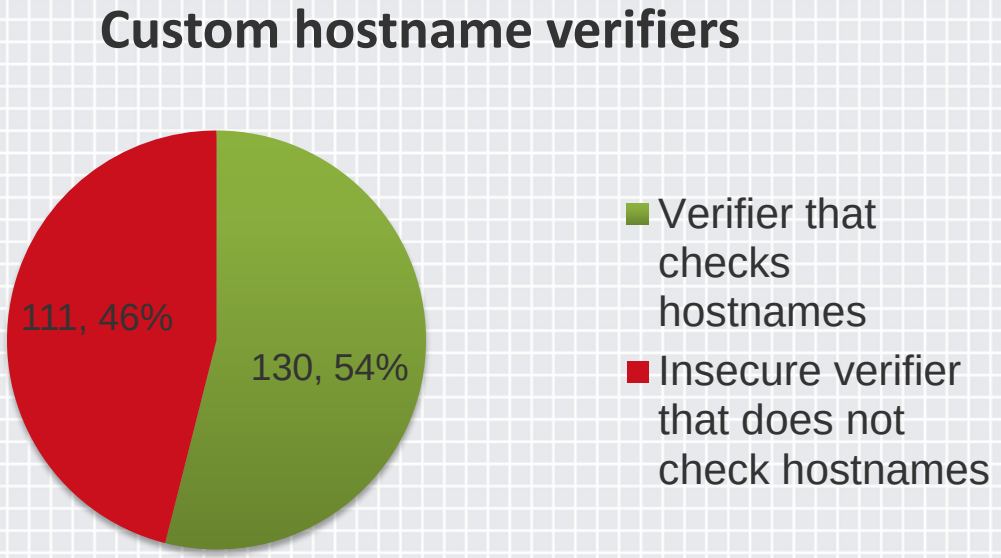


Unsafe trust managers do not check certificates

Part I. Statistics



Stats are for the 5% of applications using SSL/TLS that pass a HostnameVerifier to HttpsURLConnection



Apps in red effectively disable hostname verification, allowing a man-in-the-middle attack

Vulnerabilities are sometimes not your fault

- Crypto is hard to get right, even for experts
- OpenSSL is the most popular crypto library, and has a history of severe vulnerabilities with cute names. Some are implementation bugs, others are protocol design flaws.
- If you package your own crypto library with your app, it is important to keep it up to date.

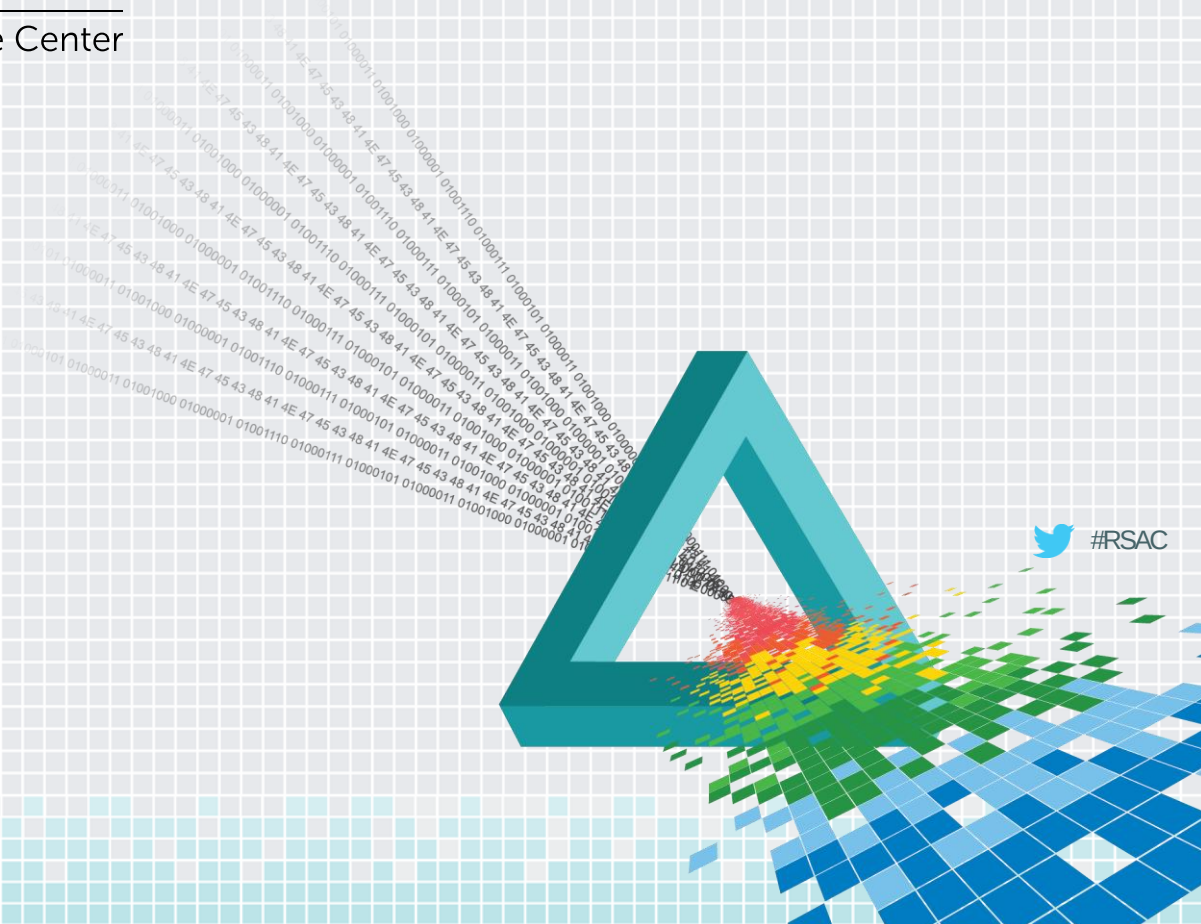
OpenSSL vulnerability du jour: FREAK

- Attack spoofs the communication endpoints into using weak, “export” grade ciphers (remember those?)
- Both client and server must be vulnerable to exploit
- 11.2% of apps on Google Play with over one million downloads were vulnerable to FREAK as of last month
  - They connect to a vulnerable server
  - They use Android’s built-in OpenSSL implementation or package their own vulnerable version

RSAConference2015

San Francisco | April 20-24 | Moscone Center

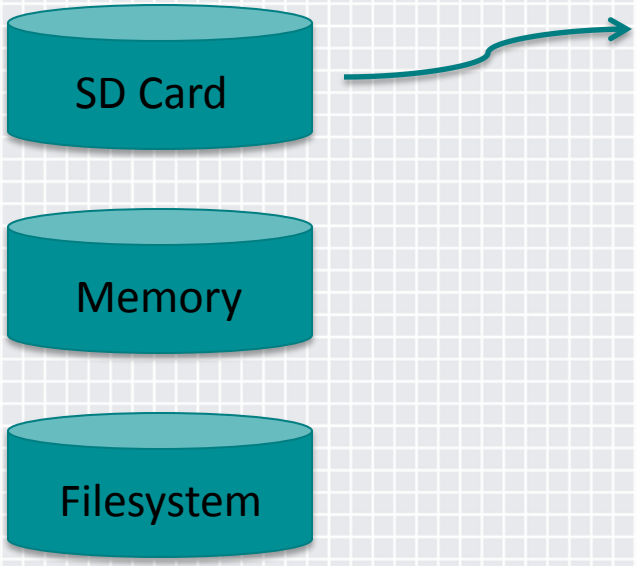
Data Encryption



Part II. Encrypting data: Why?

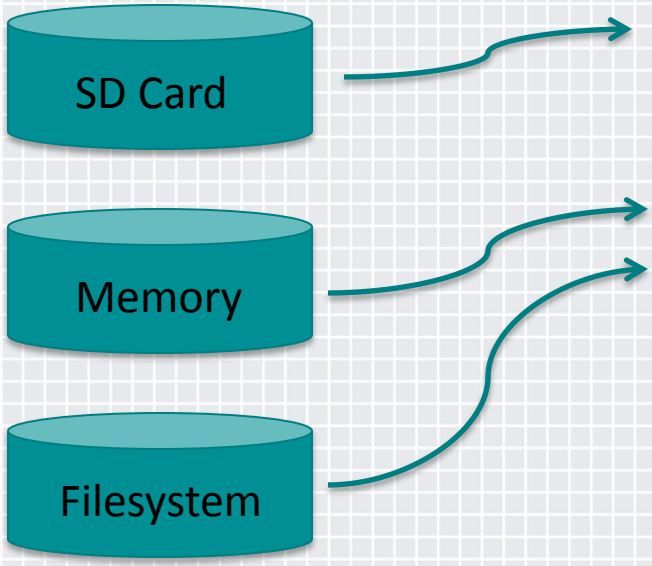
- Encryption makes it difficult for a malicious entity to read sensitive information on the device
  - Data stored on the filesystem, in memory, or even the SD card
- Encryption allows sensitive data sharing with untrusted parties
  - Data encrypted by an endpoint device can be kept private even from a service provider that stores or transfers the data

Part II. Data stored on the device



1 Any application can read data on the SD card

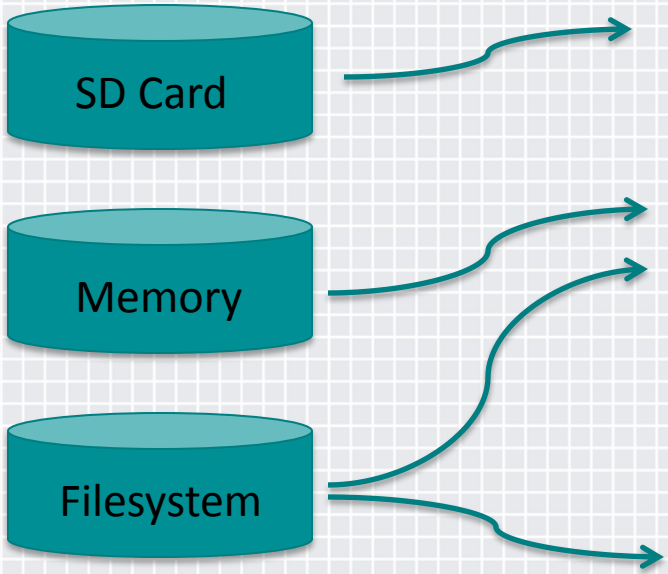
Part II. Data stored on the device



1 Any application can read data on the SD card

1 Code injection using other vulnerabilities provide a backdoor to data in memory or the filesystem

Part II. Data stored on the device



1 Any application can read data on the SD card

1 Code injection using other vulnerabilities provide a backdoor to data in memory or the filesystem

2 A compromised machine can access the device filesystem over USB using ADB

Part II. The need for encryption

Demo…

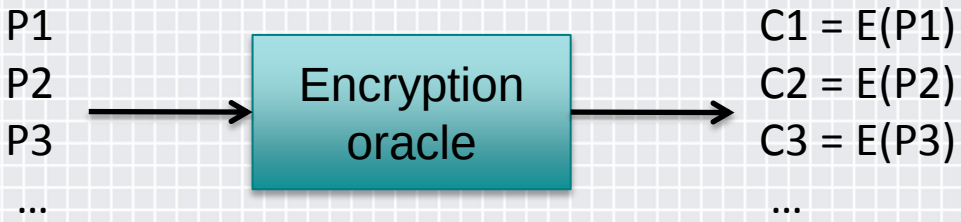
Part II. Desirable security properties

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Encryption should be hard to reverse

Indistinguishability under Chosen Plaintext Attack (IND-CPA):
First, the attacker is allowed to have many messages of their choice encrypted and see the results.

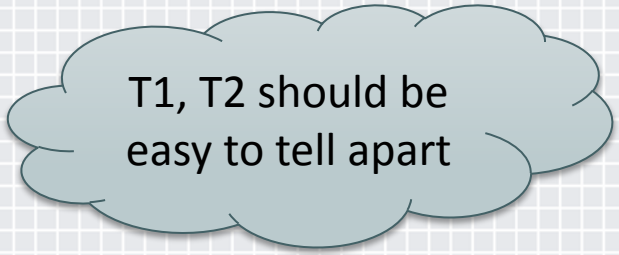


Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Encryption should be hard to reverse

Indistinguishability under Chosen Plaintext Attack (IND-CPA):
First, the attacker is allowed to have many messages of their choice encrypted and see the results. Attacker then specifies two messages of their choice that they will try to distinguish between.

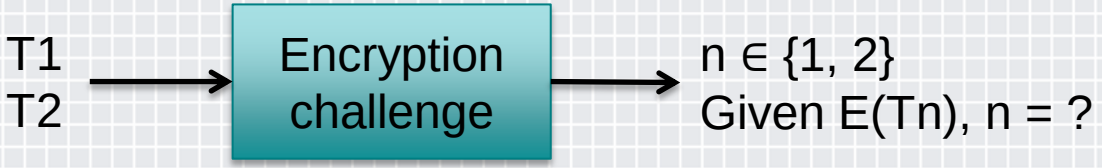


Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Encryption should be hard to reverse

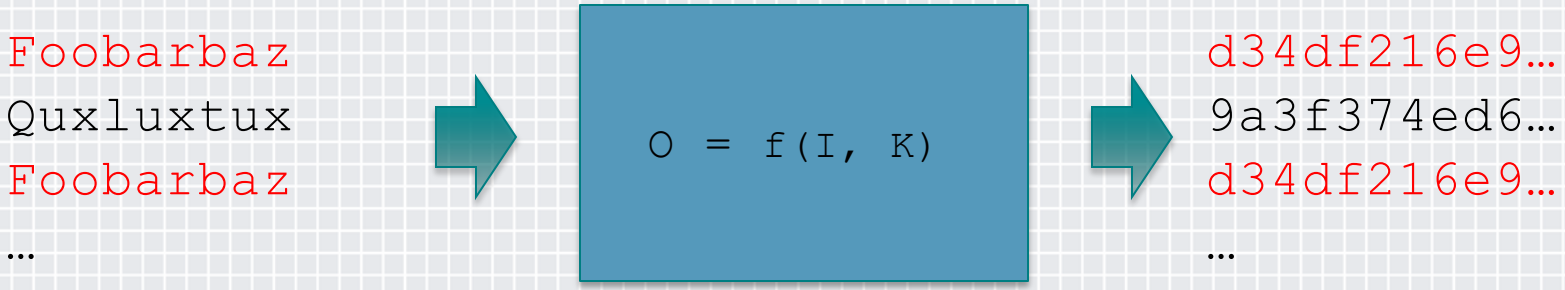
Indistinguishability under Chosen Plaintext Attack (IND-CPA):
Given the encryption of one of these chosen at random, they try to guess which one it is. If no attacker can guess correctly more than half the time, the algorithm is IND-CPA.



Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

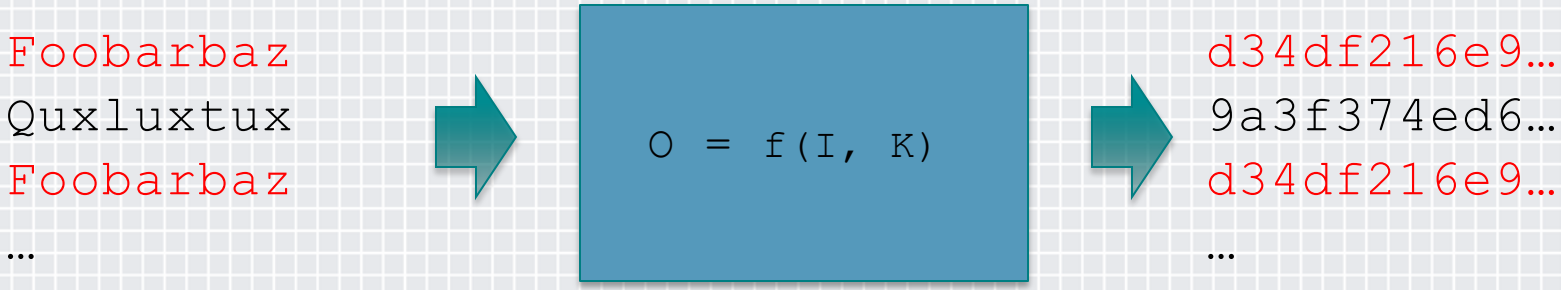
- Robustness against Chosen Plaintext Attack



Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness against Chosen Plaintext Attack

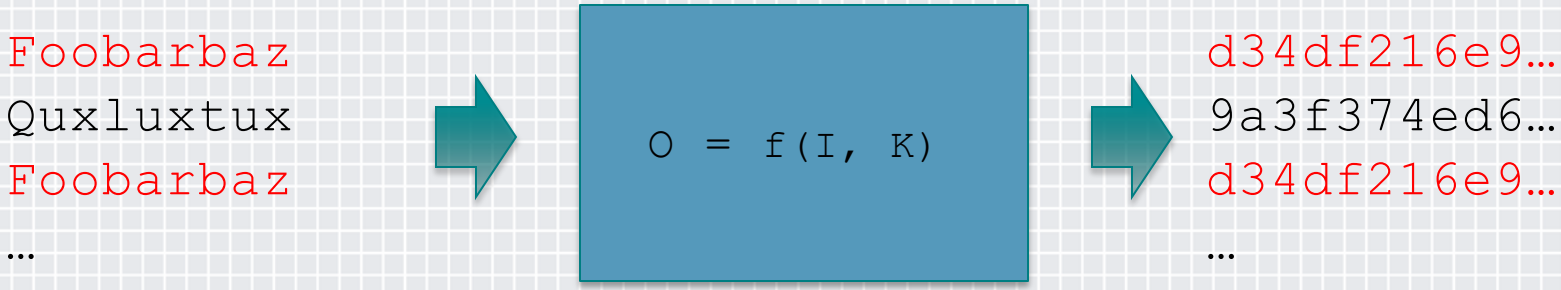


Identical inputs (I) get encrypted using key (K) to identical outputs (O)!
Algorithm is ***stateless***, and thus gives a trivial attack.

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness against Chosen Plaintext Attack



Attacker can easily build a reverse dictionary by choosing a large number of inputs and recording the outputs

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness against Chosen Plaintext Attack

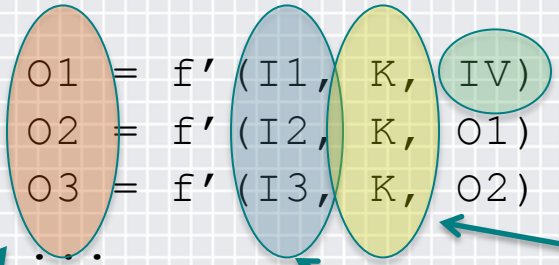
```
O1 = f'(I1, K, IV)
O2 = f'(I2, K, O1)
O3 = f'(I3, K, O2)
...
```

A better algorithm that is
Stateful

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness against Chosen Plaintext Attack



Initial Vector

K is the encryption key

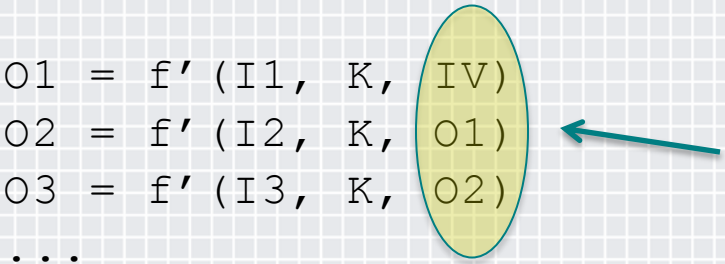
Output blocks

I1, I2, I3, … are fixed size blocks of input text

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness against Chosen Plaintext Attack



K and IV are used to encrypt O1
K and O1 are used to encrypt O2
K and O2 are used to encrypt O3
etc.

Algorithm is ***stateful***. Additional input to each block means identical blocks differ. The **IV** is used to ensure that similar or identical messages will have no similarities in cyphertext

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness against Chosen Plaintext Attack

```
O1 = f'(I1, K, IV)
O2 = f'(I2, K, O1)
O3 = f'(I3, K, O2)
...
```

Given appropriate function `f'`, inputs are ***indistinguishable under chosen plaintext attack***, offering a strong security guarantee.

Overall goal: make it computationally infeasible to break the crypto

Part II. Android (Java) capabilities

Encryption mode

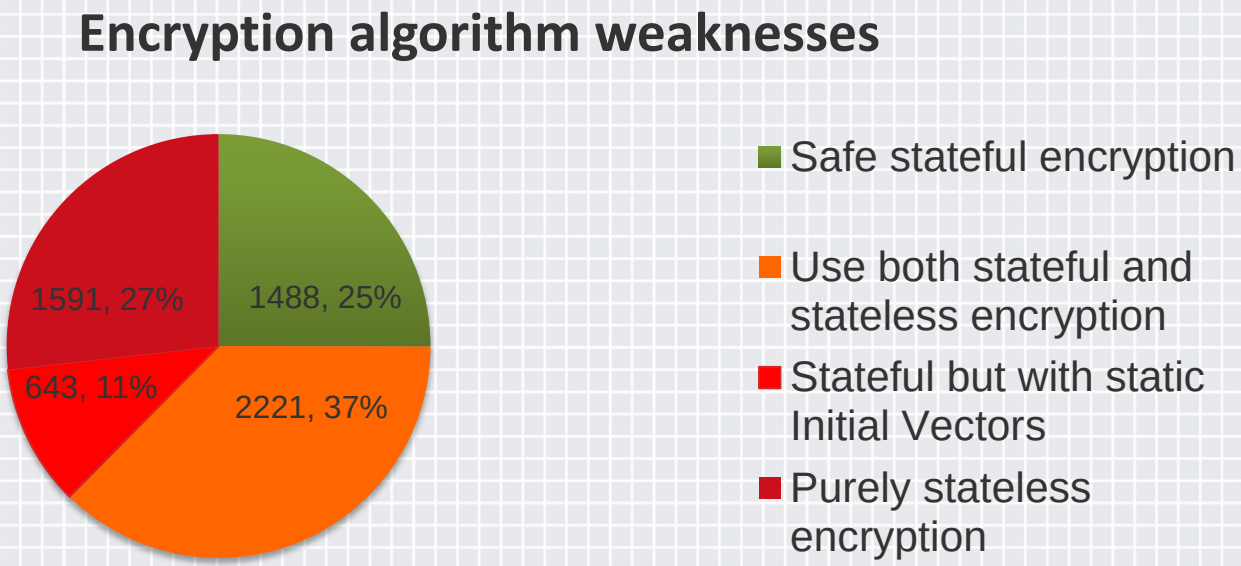
```
javax.crypto.Cipher.getInstance(String transformation)
```

The Cipher class is a general mechanism for obtaining instances of different cryptographic block ciphers in various operation modes, with a string of the form algorithm/mode/padding

Problem: The mode “ECB” is stateless and vulnerable to chosen plaintext attack. The other, stateful modes are only IND-CPA secure if a randomized initial vector is used

Solution: Use a stateful mode like “CBC” or “CTR”. Use a securely random (high entropy) value for the initial vector each time.

Part II. Statistics



Apps in red and orange use encryption modes or parameters that are not secure against chosen plaintext attack

Part II. Android (Java) capabilities

Encryption key selection

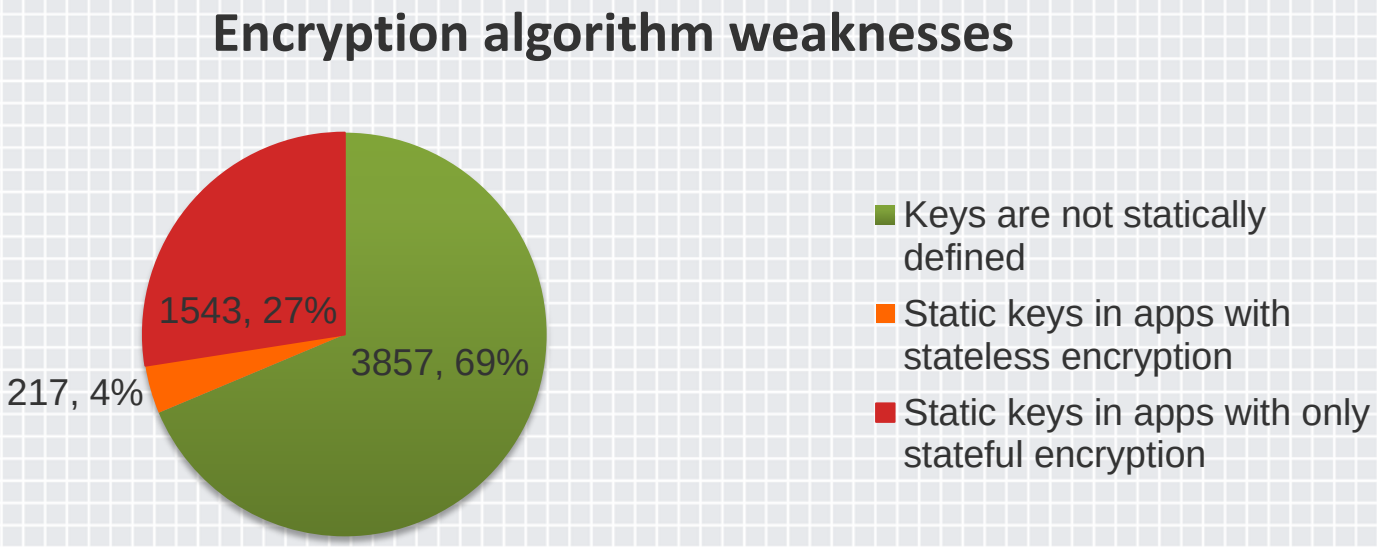
```
javax.crypto.Cipher.init(…,Key key,…)
```

Cipher class is initialized with a passed-in key

Problem: The key should not be a fixed value, where it could be recovered from one device and used to decrypt data from another device.

Solution: Reliably protecting data belonging to the user requires use of a user-specific key that is not stored persistently on the device, such as a password-based key. But a password-based key should be hashed with many iterations to make password-guessing attacks impractical.

Part II. Statistics



Orange and red apps encrypt with static (fixed) keys that can be easily extracted by reverse-engineering the application and used to decrypt stored data

Part II. Desirable security properties

- Entropy in random sources
  - High entropy implies more unpredictability
  - Low entropy implies more predictability

We want a high degree of entropy (unpredictability) in the random numbers that are used for security-sensitive purposes like key generation. They should not follow any guessable pattern.

Overall goal: make it computationally infeasible to break the crypto

Part II. Android (Java) capabilities

Random number generation

```
java.security.SecureRandom(byte[] seed)

secureRandom.setSeed(byte[] seed)
```

The SecureRandom class defaults to seeding itself from /dev/urandom, but also lets the user intialize the generator with their own seed.

Problem: If a fixed value is used, it renders the output of the “random” generator completely predictable. We’ve seen cases where the value is a static constant of the program.

Solution: The default seed uses the Linux kernel’s random number generator, which should suffice. Only seed the generator explicitly if you need and have a better entropy source than this.

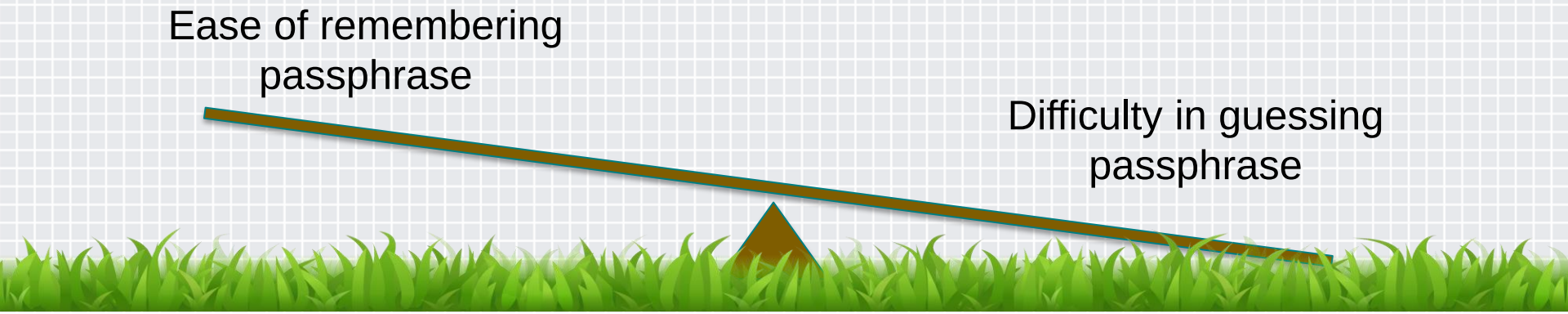
Part II. Statistics

Low entropy in random number generation
(for the 7% of apps that explicitly set random seeds)

Apps in red have low entropy since they seed the random generator with static strings defined in the application bytecode, resulting in completely predictable “random” numbers

Part II. Desirable security properties

- Robustness of Password-Based Encryption
  - Given a password how do we encrypt data?



Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness of Password-Based Encryption

```
K1 = augment(Passwd, Salt)
K2 = augment(K1, Salt)
K3 = augment(K2, Salt)
...
K1000 = augment(K999, Salt)
```

Augment key from previous step with a Salt to generate key for the next step

Overall goal: make it computationally infeasible to break the crypto

Part II. Desirable security properties

- Robustness of Password-Based Encryption

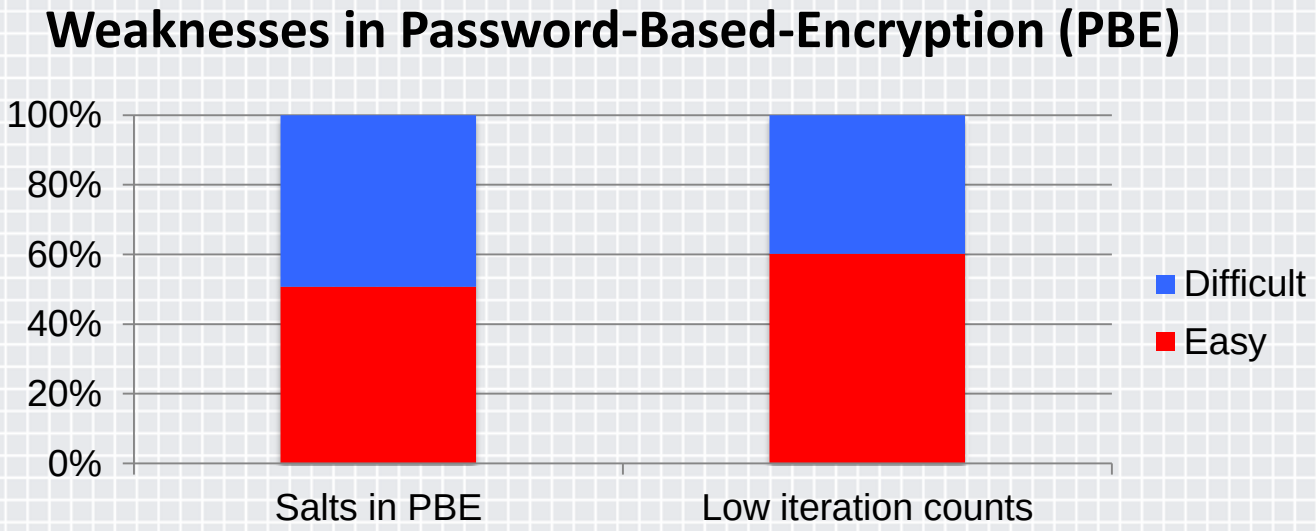
```
K1 = augment(Passwd, Salt)
K2 = augment(K1, Salt)
K3 = augment(K2, Salt)
...
K1000 = augment(K999, Salt)
```

Augment key from previous step with a Salt to generate key for the next step

K1000 is used as the final encryption key. The extra iterations slow down a password-guessing attack
(RFC 2898 recommends >=1000 iterations.)

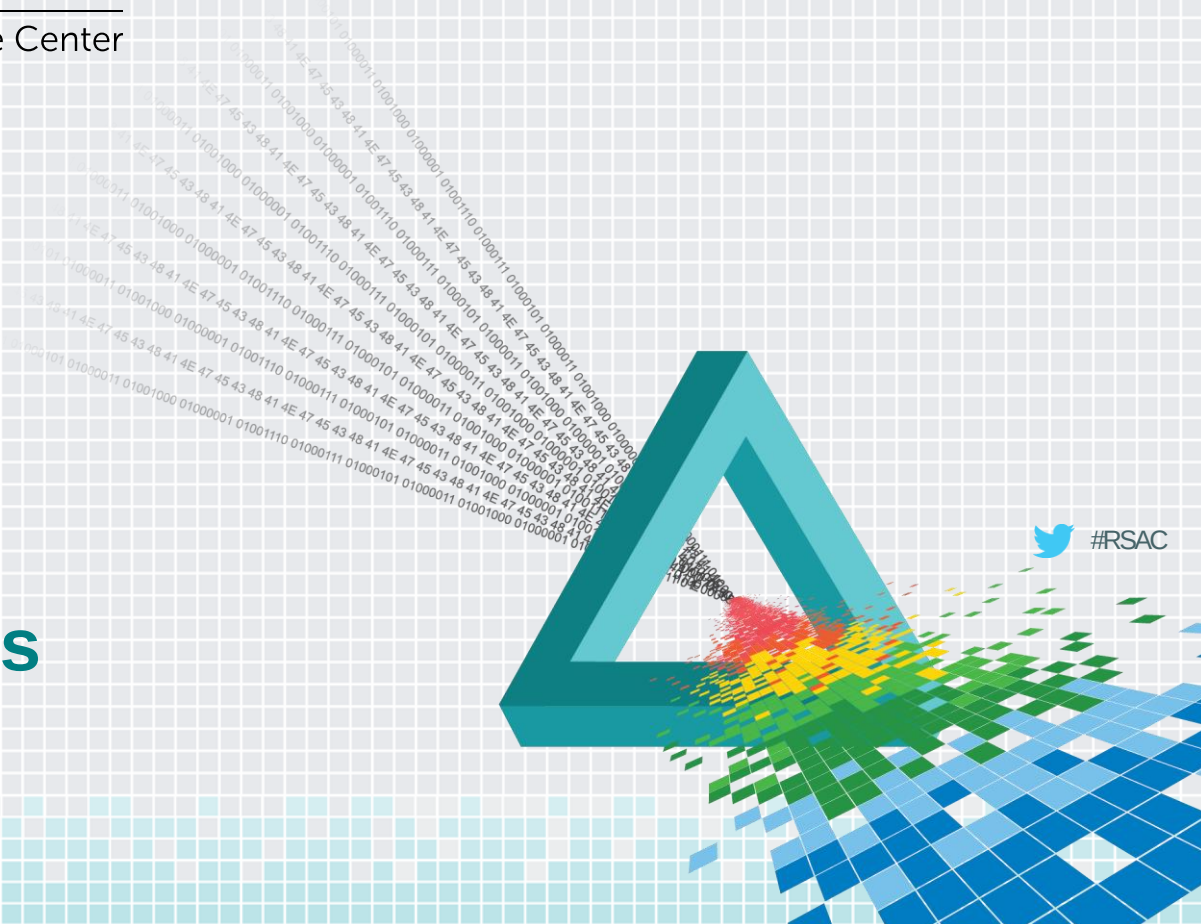
Overall goal: make it computationally infeasible to break the crypto

Part II. Statistics



Red apps have with vulnerabilities that make it faster to break encryption keys by guessing the password

Summary and Recommendations



Best Security Practices for SSL/Cryptography

SSL/TLS

- Make sure that you are not bypassing important security features by using insecure TrustManagers or HostnameVerifiers in production code
- Keep your libraries up to date

Cryptographic operations

- Use algorithms that are *stateful* and have the *indistinguishability under chosen plaintext attack* property
- Pick keys, initial vectors, and salts using a source of random numbers with high entropy
- Use at least 1000 iterations in computing keys to use with *Password-Based Encryption*

Apply Slide

- Things to do soon:
  - Verify that your apps don't use unsafe HostnameVerifiers or TrustManagers. Doing so can defeat the protections provided by SSL.
  - Update your SSL implementations on app and server
- In the next few months
  - Consider how you store potentially sensitive user data on devices. Can you improve the handling of this data to improve user privacy?
  - Audit apps used by your organization for security vulnerabilities
- Long-term, consider
  - Features and design approaches for your apps to help user privacy and security