Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks

Soumya Indela
Electrical and Computer Engineering, University of Maryland - College Park, USA
sindela@umiacs.umd.edu

Mukul Kulkarni
Electrical and Computer Engineering, University of Maryland - College Park, USA
mukul@umd.edu

Kartik Nayak
Department of Computer Science, University of Maryland - College Park, USA
kartik@cs.umd.edu

Tudor Dumitraș
Electrical and Computer Engineering, University of Maryland - College Park, USA
tdumitra@umiacs.umd.edu

Abstract

Several mature cryptographic frameworks are available, and they have been utilized for building complex applications. However, developers often use these frameworks incorrectly and introduce security vulnerabilities. This is because current cryptographic frameworks erode abstraction boundaries, as they do not encapsulate all the framework-specific knowledge and expect developers to understand security attacks and defenses. Starting from the documented misuse cases of cryptographic APIs, we infer five developer needs and we show that a good API design would address these needs only partially. Building on this observation, we propose APIs that are semantically meaningful for developers, we show how these interfaces can be implemented consistently on top of existing frameworks using novel and known design patterns, and we propose build management hooks for isolating security workarounds needed during the development and test phases. Through two case studies, we show that our APIs can be utilized to implement non-trivial client-server protocols and that they provide a better separation of concerns than existing frameworks. We also discuss the challenges and potential approaches for evaluating our solution. Our semantic interfaces represent a first step toward preventing misuses of cryptographic APIs.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures; E.3 [Data Encryption]

ACM [to be supplied]. . . $15.00

Keywords cryptography, semantic API, functionality engineers, security engineers, regulator pattern

1. Introduction

Cryptographic algorithms are often a necessary building block for complex applications and libraries, for instance to implement secure client-server communications, to store data securely or to process payments. Several mature cryptographic frameworks are currently available for this task, including Oracle JSSE, IBM JSSE, BouncyCastle, and OpenSSL. These frameworks are implemented by cryptography experts, include state-of-the-art algorithms, and their code has been audited and analyzed with formal verification tools. They have also used to build real world software that provides strong security.

Unfortunately, software developers who lack cryptography expertise often make critical mistakes when using these frameworks, including exchanging keys without authenticating the endpoint [25], storing sensitive information in cleartext [25] or with weak protection [13], using parameters known to be insecure (e.g. the Electronic Codebook mode or non-random initialization vectors) for block ciphers [13] using encryption keys that are constant [13] or are generated from insufficient randomness [13, 25], or performing improper TLS certificate validation [14, 16, 25]. The Common Weaknesses Enumeration dictionary [9], which provides a comprehensive taxonomy of frequent programming mistakes, includes 14 common implementation errors related to the use of cryptography. These errors allow attackers to impersonate legitimate users [14, 16, 25], to harvest sensitive personal information [14, 16, 25] and even to steal money [25].

The solutions that have been proposed for this problem include simplified cryptographic APIs [1, 6, 14, 33], secure default values for the parameters of cryptographic algo-

rithms [16] and using static analysis tools to discover bugs related to cryptography misuse [3, 13, 22]. These solutions do not address a more fundamental problem: the fact that current cryptographic frameworks *erode abstraction boundaries*, as they do not encapsulate all the framework-specific knowledge and expect developers to understand security attacks and defenses.

In this paper, we characterize the semantic gap between the needs of developers and the cryptographic APIs, and we present techniques for bridging this gap. For example, simplified cryptographic APIs do not provide a true separation of concerns as they do not provide the flexibility that developers need to implement the complex business logic required (e.g. authentication and authorization services, e-commerce SDKs and integrated shopping carts [16] or mobile payment systems [25]). This problem could be addressed in part by following best practices in API design [7], but some challenges are specific to the cryptography domain. In particular, the security of applications using cryptography often depends on *information external to the system*. For example, the SHA-1 cryptographic hash function, introduced two decades ago, is no longer considered secure given the performance of modern hardware, yet many web sites still advertise TLS certificates that use SHA-1 for generating digital signatures. The checks for implementation choices that may lead to insecurity must be done are runtime, as in some cases the information changes frequently. For example, when TLS certificates are compromised, they must be revoked and reissued immediately to prevent man in the middle attacks,[1] and client-side code must check for the revocation status of these certificates. Moreover, developers need the flexibility to select the most appropriate mechanism for incorporating this information. For example, an application can check the revocation status of TLS certificates by downloading certificate revocation lists (CRLs), by using the Online Certificate Status Protocol (OCSP) or by implementing OCSP stapling. There is currently *no agreement about what method is best*, and the choice is likely to be platform dependent [24]. In consequence, it is difficult to define simplified APIs or statically verifiable security protocols that cover all the ways cryptography is used in the real world.

Another domain specific challenge is that developers often need to *disable security checks in the development environment* in order to run and test their application (e.g. by disabling server authentication with self-signed TLS certificates [14]), and sometimes these checks remain disabled in production because the developers do not understand the security threats associated with these workarounds [14, 25]. This suggests that designing good cryptography APIs is not sufficient for addressing the problem, and the solution must extend to the build management system.

[1] Exploits for the Heartbleed vulnerability, which enabled breaking TLS certificates at scale, were observed in the wild less than 24h after the vulnerability was disclosed [12].

To address these challenges, we propose *semantic APIs* for cryptographic libraries, which expose the security decisions without requiring in depth knowledge of attacks and defenses. We describe several *design patterns for implementing these APIs*, including three ways of incorporating external information, and we demonstrate how our APIs can be implemented on top of the existing cryptographic frameworks. Our APIs represent a first step toward striking the right balance between restricting the security decisions that developers make and giving them the flexibility needed for complex applications that use cryptography. In addition to these semantic APIs, we propose *compile-time checks* to separate the development environment from the production environment. This allows for a clean definition of workarounds during development that should not be used in production.

In summary, we make three contributions:

1. We identify new problems with the existing cryptographic APIs, and we classify the root causes of the new and the known programming mistakes related to using these APIs.
2. We present a solution to these problems by introducing semantic APIs for cryptographic operations. We also discuss design patterns for implementing these interfaces on top of existing cryptographic frameworks.
3. We propose build management hooks for isolating the workarounds used during development and testing.

The rest of the paper is organized as follows. In Section 2 we outline our goals and non-goals. In Section 3 we review problems with existing cryptographic APIs, not described in the prior work, and we categorize the needs of developers who use these APIs. In Section 4 we describe our solutions to these problems. In Section 5 we validate our solutions through several case studies. In Section 6 we review the prior work and in Section 7 we discuss the remaining challenges.

2. Problem Statement

Consider a developer Alice who wants to develop an application in Python, and she wants to use the HTTPS protocol in her software to communicate securely with a web service called Binary Object Broker (BOB). The HTTPS protocol allows clients to connect to servers they have not encountered before, and with which they have no shared secrets, by utilizing the TLS protocol to exchange keys during the initial handshake. A common misuse of cryptographic APIs (CWE-322) is to perform a key exchange without first authenticating the server [9, 14, 25], which results in the establishment of a secure channel without first ensuring that the client has connected to the correct server. This programming mistake allows an adversary to intercept the communication through a man-in-the-middle attack [20]. To prevent this attack, an HTTPS server must present a digital certificate, signed by a Certification Authority that the client trusts, which authenticates the server to the client. While Alice is not a cryptography expert, she tries to avoid

```
1 import socket, ssl
2 context = ssl.create_default_context()
3 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 ssl_sock = context.wrap_socket(s, server_hostname='www.google.com')
5 ssl_sock.connect(('www.google.com', 443))
```

Figure 1: HTTPS request in Python.

such common mistakes by looking up the best practices for using Python libraries to establish a secure connection with the web server. This results in the code shown in Figure 1.[2] As the figure shows, Alice creates a default context, which authenticates the BOB service by requesting a certificate and verifies that the web server's certificate hostname matches the one from the certificate. Although Alice believes that her implementation is secure, her code never checks if the certificate has been revoked, leaving the application exposed to a man-in-the-middle attack.[3] This is because the default option (`ssl.VERIFY_DEFAULT`) in Python does not check for revoked certificates. Alice must explicitly specify `context.verify_flags = ssl.VERIFY_CRL_CHECK_CHAIN` in Figure 1 to ensure this. The fundamental cause of this error is that the library expects developers to have a good understanding of security attacks and defenses and to know details about the library's implementation and configuration.

This is just one example of security vulnerabilities that can be introduced by misusing cryptography; recent studies [13, 14, 16, 25] have reported that such mistakes are common. The root causes of these mistakes fall into four categories:

1. **No separation of concerns.** Existing cryptographic frameworks are implemented by experts and can be used correctly. However, the APIs they expose do not encapsulate all the cryptographic knowledge and expect users such as Alice to understand security attacks and defenses and the subtle impact of various parameters used in the framework implementation. In consequence, when using these frameworks developers cannot focus only on the application logic but must also learn about cryptography.
2. **Diverse needs of developers.** Multiple types of developers may need to use cryptographic frameworks. For example, *functionality engineers* (such as Alice) often have simple requirements, such as communicating securely over the Internet, while *security engineers* must implement more complex services that rely on cryptography. Prior efforts to simplify cryptographic APIs, to make them more suitable for developers who lack cryptographic expertise [6, 14], do not take into account the diversity of these developers' needs.
3. **Need to incorporate external information.** One of the reasons why existing cryptographic frameworks expose a dizzying array of options and parameters is to allow developers to react to the community's evolving understanding of cryptographic attacks and defenses. Some implementation choices are found to jeopardize security, and the rate at which this information is updated ranges from years (in the case of cryptographic algorithms, e.g. SHA-1) to days (in the case of certificate revocations). Developers must find a way to incorporate such external information into their systems.
4. **Reliance on secure default values.** The prior efforts to simplify cryptographic APIs remove choices available to developers, which leads to additional mistakes when programmers develop workarounds. For example, while the default settings for the Android SSL library ensure correct certificate validation, developers often need to disable these validation checks during development and testing, by using self-signed certificates [25], and do not understand that using this workaround in production code fails to authenticate the server [14]. In addition to cryptographic APIs that ensure a separation of concerns and that provide secure defaults, developers need appropriate build management tools to define workarounds that should only be used in the development environment.

Goals and non-goals. We present ideas that would help developers to use cryptography correctly, by addressing the four challenges identified above. However, we do not aim to enforce a secure usage of cryptographic APIs. We do not think this is currently feasible as developers could always resort to do-it-yourself crypto to bypass our enforcement points. In other words, we assume that, like Alice in our example, the developers are trying to write secure software and we aim to make it easier for them to achieve this goal. Furthermore, we do not propose new cryptographic techniques or algorithms, and we do not describe any cryptographic weaknesses in the existing algorithms. Instead, we focus on preventing common mistakes in the way these algorithms are used in applications. Finally, we do not consider the mistakes made by developers who implement their own custom ways of encrypting data, authenticating users, etc., thereby bypassing cryptographic frameworks entirely.

[2] From `https://docs.python.org/2/library/ssl.html#ssl-security`;

[3] Because TLS certificates are sometimes compromised (e.g. in the wake of the Heartbleed vulnerability [38]), client-side TLS code must check the revocation status of certificates presented by servers during the TLS handshake to prevent an adversary from eavesdropping on the connection.

3. Needs of Developers

From the context of programming mistakes reported in prior work [13, 14, 16, 25], we identify five general needs of developers who must use cryptographic APIs. By further investigating these needs, we also identify several challenges, not documented before, for using existing APIs correctly.

Need 1: *Establish secure connections.* One of the most common reasons to use cryptographic frameworks is to implement client-server applications that communicate over secure channels, often using the HTTPS protocol. However, the counter-intuitive interfaces and parameters exposed by these frameworks can lead to programming mistakes. For example, JSSE performs hostname verification only if an algorithm field is correctly set to "HTTPS" [16], but developers who are not familiar with the steps from the TLS handshake are likely to establish an insecure connection. Similarly, in Python, hostname verification is skipped if the developer forgets to specify `ssl.CERT_REQUIRED` (Figure 1). In the `cURL` library, the parameter for requesting hostname verification and certificate validation is 2 (an integer), while in JSSE it is TRUE (a boolean), which confuses developers who sometimes invoke the `cURL` API with 1 (the integer that corresponds to TRUE) [16].

We also identified cases where *different cryptographic frameworks behave differently when providing the same functionality*. This can lead to mistakes when developers move from one framework to another. For example, if a trusted certificate has expired, in an IBM JSSE implementation, the handshake will fail, even though the expired certificate is trusted. An Oracle JSSE implementation will flag such a connection as secure and expect the developer to check for expired certificates. Developers who are not security experts need a cryptographic API that abstracts away these details and that minimizes astonishment [7].

Need 2: *Store data securely.* Another frequent requirement is to store sensitive data, e.g. personally identifiable information, keys, credit card information, account balances and other application related information. However, developers sometimes store such information on the device in plaintext, use hard-coded keys or insufficiently random values for encryption, or allow the sensitive information to leak through log files [25]. Developers usually know which information handled by their applications is sensitive, but they make mistakes because they must invoke correct encryption operations each time the data is written to disk, in some form. Instead, a cryptographic API should decouple the task of specifying that certain data structures contain sensitive information and the secure storage primitives. For data marked as sensitive, these primitives should automatically encrypt the data in a way that ensures confidentiality (an unauthorized party cannot decrypt it) and integrity (the data cannot be forged or tampered with).

Need 3: *Incorporate security-critical external information.* Systems that were designed and implemented correctly a decade ago may be vulnerable today because of changes in the security landscape. For example, the DES encryption algorithm or the MD5 hashing algorithm were considered secure in the past, but today they are known to be insecure. Nevertheless, there are still applications that use DES and MD5. Similarly, as computing power increases, increasingly longer keys are necessary for providing security against brute force attacks. The cryptographic framework should check and enforce these recommendations transparently. This can be achieved by requesting the information, in a machine readable format, from a trusted third party—perhaps the National Institute of Standards and Technology (NIST), which periodically publishes recommendations for the use of cryptographic algorithms and key lengths [4]. Web browsers use a similar pattern for determining the revocation status of TLS certificates, which is another type of external information that is critical for security.

Need 4: *Use default parameters securely.* Developers who lack a background in cryptography will often end up using the default values of parameters required by various algorithms. Sometimes, the default values compromise security; for example, when the AES block cipher is used, the insecure ECB mode is the default in Python's PyCrypto [23], Java JSSE libraries (as well as resulting Android libraries) [13]. Cryptographic frameworks should provide default values that ensure security.

Need 5: *Disable security checks during development and testing.* Because security implies that certain operations will be disallowed, developers often need a way to bypass security checks in the development environment in order to test all the code paths. For example, when a developer starts building an SSL application, the code throws exceptions either due to the absence of a certificate or to the use of a self signed certificate. Many such developers then bypass SSL certificate validation, to be able to continue writing and testing their code. While these workarounds have a legitimate purpose in the development environment, they are sometimes deployed in a production environment, making the application vulnerable [14]. Cryptographic frameworks should provide a way for developers to specify that certain workarounds should be executed only in the development environment.

4. Unified Framework for Secure Application Development

The needs and few mistakes identified in the previous section have been summarized in Table 1. To address these needs, we propose a unified framework with the following components:

1. Semantic APIs, which present the high level functionality and security guarantees to the developers without exposing low level implementation specifics. We show that these APIs can be implemented consistently on different plat-

Table 1: **Mapping developer needs, mistakes and solution** - CWE provides a unified, measurable set of software weaknesses [9]. For example from the CWE 297 in the list corresponds to the mistake "Improper Validation of Certificate with Host Mismatch" `https://cwe.mitre.org/data/definitions/297.html`.

Developer Needs	Example Mistake	Solution	CWE	Section
1. Establish secure connection	Allow expired certificates, Skip Hostname Verification	Semantic APIs, Integrating external information	295, 297, 599, 319, 321, 322, 324, 327	4.1, 4.2
2. Store data securely	Secret keys stored unencrypted	Semantic APIs	311, 312, 532	4.1
3. Incorporate security critical information	Using SHA1 digest	Integrating external information	299, 327 370, 676	4.2
4. Use default parameters securely	Using AES in ECB mode	Semantic APIs, Setup build configuration	276, 453	4.1, 4.3
5. Disable security checks during development	Using self-signed certificates in production environment	Setup build configuration	296	4.3

forms, without modifying the underlying cryptographic framework (Section 4.1).

2. Design patterns for integrating information from external trusted sources, transparently and at runtime (Section 4.2).
3. Ensuring correct compile time procedures to separate development environment from production environment. This can be done using code annotations in Java (for instance, using profiles in Spring framework) or using preprocessor macros in languages like C (Section 4.3).

4.1 Semantic APIs

In this section we introduce our semantic APIs, along with the use cases that motivated their design. Table 2 lists these APIs.

Our design goal is to allow only a few developers, which we call the Security Engineers, to be involved in making security decisions and provide them with the tools they need to implement custom protocols that meet certain security requirements, whereas the other developers, called Functionality Engineers should focus on functionality specific needs and their design choices should not affect security. We achieve this goal by ensuring that the functions written by the Functionality Engineers do not involve any security decisions whereas the functions written by the Security Engineers perform all the security tasks and these functions can be used by the Functionality Engineers directly without the complete knowledge of the underlying function and cannot be modified by the Functionality Engineers.

4.1.1 Functionality Engineers

Communicate Interface. The functionality required by developer need 1, described in Section 3, can be implemented with `connect`, `secureConnect`, `send`, `secureSend`, `receive` and `secureReceive` functions. `connect`, `send` and `receive` allow communication without authentication or encryption, whereas `secureConnect`, `secureSend` and `secureReceive` ensure both properties.

From a functionality engineers' perspective, the secure functions would be used in the same manner as their insecure counterparts. Such developers would use `secureConnect/secureSend` for sending sensitive data such as login credentials, SSN, credit card numbers, etc. We must therefore ensure that these functions perform all security checks that are required for what developers expect to be a secure channel. A detailed description of a TLS connection establishment for HTTPS and the necessary check to be performed is provided in Appendix A. `send` can be used for all other communications and for channels that cannot be secured (e.g. text messages sent by a smartphone).

When sending a message on a secure channel, the functionality engineer specifies only the address `addr` and the data `msg`. The API implementation should ensure that confidential message is always sent using `secureSend` (as shown in Figure 2). This can be achieved by requiring security engineers to explicitly specify an `isConfidential` function, which takes a `msg` as a parameter and returns a boolean value, indicating whether the data is confidential.

The `send` function invokes `isConfidential` and returns an error rather than sending confidential data over an insecure channel. The method returns `true` by default, explicitly whitelisting data that is not sensitive. The security engineers define `msg` to be an instance of a specific superclass, which tags all data as sensitive or non-sensitive.

This decouples the task of specifying which data is confidential from the task of implementing network communications, and ensures that the network programmer cannot compromise security by mistakingly using `send` in cases where `secureSend` should be used.

Storage Interface. The functionality required by developer need 2 can be implemented with the `write`, `secureWrite`,

Table 2: Proposed Semantic API with functions for both application and library developers [15].

API	Parameters	Semantics	Design Pattern
Functionality Engineers			
`Communicate Interface`			
`send(sock, msg)`	`addr`: address of the sender/receiver `msg`: data to be sent/received `sock`: socket for communication	Sends a message to the receiver	Regulator, Proxy, Template
`secureSend(sock, msg)`		Sends authenticated encrypted message	
`receive(sock)`		Receive a message from `addr`	
`secureReceive(sock)`		Receives authenticated encrypted message	
`connect(addr)`		Returns an established connection	
`secureConnect(addr)`		Returns a secure SSL connection (authenticity, confidentiality, integrity)	
`disconnect(sock)`		Disconnects connection	
`Storage Interface`			
`write(key, val)`	`key`: search key (or filename) `val`: data to be stored	Writes data in plaintext	Proxy, Template
`secureWrite(key, val)`		Writes encrypted data to file	
`read(key)`		Reads data from file	
`secureRead(key)`		Reads encrypted data from file	
Security Engineers			
`dispatch(sock, msg)`	`sock`: Receiver end-point	Sends the message through socket	
`receive(sock)`	`sock`: Sender end-point	Receives message through socket	
`isConfidential(msg)`	`msg`: data to be checked for	Returns whether data is confidential	

```
1  send(sock, msg) {
2    if(!isConfidential(msg)) {
3        dispatch(msg, sock);
4    } else
5        throw Exception("msg is confidential. Use secureSend()!");
6  }
7
8  secureSend(sock, msg) {
9    assert(sock instanceof SslSocket);
10   dispatch(sslSocket, msg);
11 }
12
13 connect(addr) {
14   /* create an HttpUrlConnection object and return socket */
15 }
16
17 secureConnect(addr) {
18   /* create HttpsUrlConnection object, which performs some validation */
19   /* Perform required SSL checks that are not performed by HttpsUrlConnection */
20   /* Perform appropriate check for certificate revocation */
21   /* return sslSocket */
22 }
```

Figure 2: **Communicate interface** - example `send` and `secureSend` functions to perform a connection using HTTP and HTTPS protocols.

`read` and `secureRead` functions. A `write` allows writing a value (or a file) to the storage system in plaintext whereas `secureWrite` ensures the integrity and confidentiality of the file. Whether the data stored is sensitive or not, a functionality engineer only needs to specify a value that is to be stored and a key that can be used to read the value. As in the Communicate interface, an `isConfidential` function must be specified that checks whether some data written in plaintext is indeed non-sensitive, thus decoupling the data sensitivity checks from the application logic.

The APIs presented to the functionality engineer when using the `write` and `secureWrite` (or `send` and `secureSend`) methods are very similar and do not require any parameters that control how security is achieved. The Communicate and Storage interfaces provide a clean separation of concerns by hiding all the cryptography details from the functionality engineer and by decoupling the security specification from the implementation of the input-output functionality.

4.1.2 Security Engineers

Unlike functionality engineers, security engineers implement more complex interaction protocols (e.g. authentication and authorization, online shopping, payment processing), and the resulting libraries may be included in third party applications. In consequence, security engineers need a better understanding of security principles, and are responsible for exposing intuitive APIs to the functionality engineers. However, security engineers can also benefit from semantic cryptographic APIs. For example, a security engineer might implement the `send` and `secureSend` functions discussed above. Figure 2 outlines an implementation, using the proxy design pattern [15] which adds a wrapper and delegation to protect the real component from undue complexity.

connect/send. The `connect` function creates a socket for communication whereas `send` transmits the message to the destined address. In this scenario, the data may be sent in plaintext. As mentioned earlier, this function invokes the `isConfidential` function to ensure that the data sent over the network is not sensitive.

secureConnect/secureSend. The `secureConnect` function creates a secure communication channel, e.g. by using `HttpsUrlConnection` for HTTPS in Java. The function performs all SSL checks that are not performed by the underlying libraries. For example, if the security engineer uses a JSSE library by Oracle, then this function checks for the expiry of certificates. The function then performs necessary checks for certificate revocation before transmitting the message to the destination. This requires incorporating information from external sources, and can be achieved using the Regulator pattern described in Section 4.2.

4.2 Integrating External Information

In this section we describe the Regulator pattern for incorporating external information, such as parameters known to be

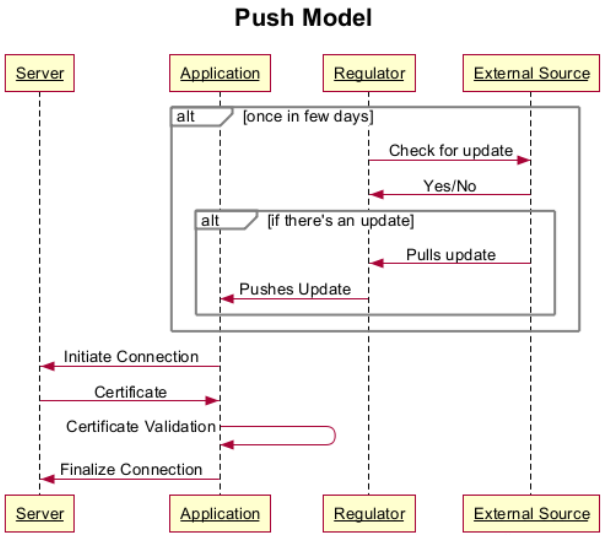

Figure 3: **Sequence Diagram showing the Push Model**

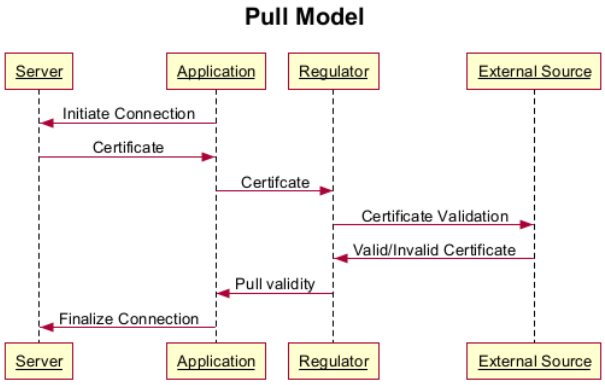

Figure 4: **Sequence Diagram showing the Pull Model**

insecure or certificates that have been revoked. We will start with a motivating example for the proposed design pattern and then will explain the pattern in standard format.

- ***Regulator:*** This pattern can address one of the most common mistakes when using cryptographic APIs: not checking if TLS certificates have been revoked and thus accepting potentially compromised certificates for authentication. Certificate revocation check can be implemented by downloading certificate revocation lists (CRLs), through the Online Certificate Status Protocol (OCSP) or OCSP stapling [24]. Each of these methods require a different mechanism to integrate the information provided by a trusted external source and hence we propose separate (but closely related) design patterns for each method. For completeness the background information about the methods of certificate revocation check is provided in Appendix B.

  Pattern Name and Classification: Regulator is a behavioral design pattern. Like the traditional Observer pat-

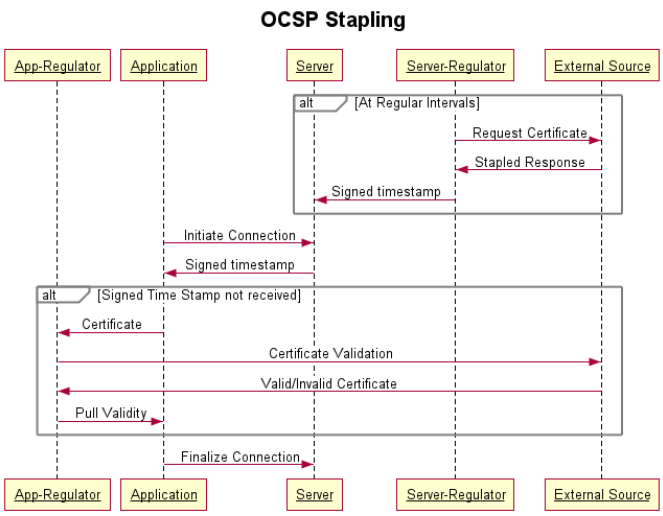

Figure 5: **Sequence Diagram showing the Selective Pull Model**

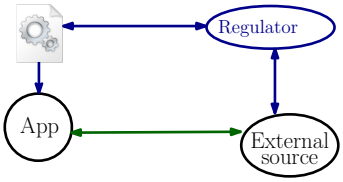

Figure 6: **General structure of Regulator pattern** The darkblue arrows are used to indicate updating of parameters by a regulator, which retrieves this data from an external source, intermittently. The dark green arrows indicate update where an application directly contacts the subject.

tern [15], Regulator defines a one-to-many relationships between objects and allows the dependent objects to update themselves automatically whenever the subject changes its state. However, there are two differences between the Observer and Regulator patterns:

1. The *Regulator* pattern does not expect the subject to be aware of all the dependent regulators, and hence the subject does not notify the regulators whenever the subject changes its state. On the contrary, it is regulators' responsibility to check if the subject has changed the state.
2. The *Regulator* pattern *does not* allow the regulators to modify the state of the subject. This is crucial since the subject is expected to be a standard or benchmark for all the regulators.

Intent: The intent of this design pattern is to facilitate mechanism for integration of information from external sources. For example it allows one to update various critical information based on the current standards. This ensures that the security checks are performed according to the current standards instead of outdated ones, thus avoiding any weak implementations. It is recommended that the regulator pattern should be used for every interface which requires to choose parameters from multiple available options by comparing them with some benchmarks.

Motivation: The prime reason for using this pattern is the unfortunate observation that many of the critical errors in the cryptographic implementations are caused by implementations of weak cryptographic algorithms like `RC4` or `MD5` hash function. Another example is using obsolete certificate revocation lists (CRLs) to validate revoked certificates. We therefore propose that the implementation of such interfaces should incorporate the regulator pattern which will update the lists periodically on its own and synchronize the lists with the current approved standards (like those published by the NIST [4]). This pattern could be used for periodic updates of CRLs and list of secure algorithms used for SSL handshake negotiation. We wish to highlight that attacks on implementations using specific parameters are much more frequent than attacks on the algorithms themselves. It is of utmost importance, therefore to keep the parameter selection up-to-date with standards at run-time instead of compile-time.

Applicability: The regulator design pattern should be used in the design whenever security depends on performing checks with reference to content published by a trusted authority.

Structure: Figure 6 shows the high level idea of how the regulator design pattern can be used for updating list of secure algorithms or list of secure parameters used for encryption. We later present detailed examples of regulator pattern that can be used in different models of updating the standard information.

We now propose 3 ways of implementing the *regulator* pattern. These 3 implementation choices differ in the situations where they can be applied. We explain these differences below and give an example for each of the implementations.

- *Push*: This implementation is illustrated in Figure 3, which corresponds to certificate revocation checks using CRLs. Here, the application stores a local copy of CRL and uses it to verify whether the certificate provided by the server is valid or not. The regulator also maintains a copy of CRL and updates its own copy periodically by downloading the the CRL updates from the certification authority (CA). The regulator then pushes the updated CRL to application and replaces the local CRL of the application with the updated one. The main advantage of this method is the application (or client) can check for the revocations locally. However,

the disadvantage of this method is the application is required to store the (potentially large) list locally.[4]

This method can also be used in scenarios where the regulator must update the lists from multiple external sources.

- *Pull*: This implementation is illustrated in Figure 4, which corresponds to certificate revocation checks using the OCSP protocol. Here, the application checks for the validity of the certificate by sending the certificate to an OCSP server (or to another trusted authority, e.g. a CA). The certificate is validated based on the response from the OCSP server (or CA). Here, the regulator relays the responses between OCSP server (or CA) and the application.

  We do not recommend this method in practice, because it adds overhead to the secure channel establishment and because of the potential privacy concerns related to disclosing to a third party every HTTPS site visited. As a result of these drawbacks, OCSP is used only in rare cases [24].

- *Selective Pull*: This implementation is similar to the *Pull* model, but in this case the information is downloaded only in certain conditions. This is illustrated in Figure 5, which corresponds to certificate revocation checks using OCSP Stapling. Here, the application requests the regulator to validate the certificate. The certificate is accepted as valid based on the timestamp when the certificate is provided. If the regulator is not able to verify the certificate's validity locally then it forwards the certificate to OCSP server (or CA). The decision regarding accepting the certificate is then taken based on the response from the OCSP server (or CA) and relayed back to the application.

  This method may be used instead of the *Pull* method, as it reduces the latency by allowing certificates to be caches for a certain period of time.

Beside certificate revocation, these patterns can also be used in other scenarios where information from some external source needs to be integrated transparently, without requiring the application developers to retrieve and interpret the information. For example, the Regulator pattern could be used is updating the list of secure algorithms; a trusted third party (e.g., the NIST, who already publishes standards for secure algorithms and key lengths [4]) could provide this information in a machine readable format, suitable for ingestion by our framework.

Consequences: The Regulator pattern allows application developers to use only algorithms and parameters that are considered secure, and it allows library developers to control the mechanism for retrieving external information. This is important, as there is currently no agreement on the best way to check the revocation status of TLS certificates [24], and the standards and protocols will likely continue to evolve. The main advantage of this pattern is that the subject does not need to be aware of all the regulators.

Further benefits and liabilities of the regulator pattern are:

1. It decouples the regulator from the subject, allowing the subject to be external and independent from the regulator.
2. The regulator can never alter the state of the subject, which ensures that all the regulators have the exact same view of the subject's state when they update themselves.
3. One of the liability is that the regulator adds an extra round of communication when used in the pull model.

The Regulator Pattern is related to the observer, mediator and iterator patterns. Similar to the observer pattern, it defines one-to-many relationships between objects, but in the regulator pattern the subject need not be aware of it's objects and does not notify any change in it's state. Similar to the mediator pattern, it encapsulates how objects interact and additionally, the regulator pattern captures changes in external information and pushes the changes to the objects. Similar to iterator pattern, it defines how objects access external information iteratively without loss of information. The regulator pattern is different from the observer pattern as it checks for updates and pushes the updates to it's objects besides just defining how the information is accessed.

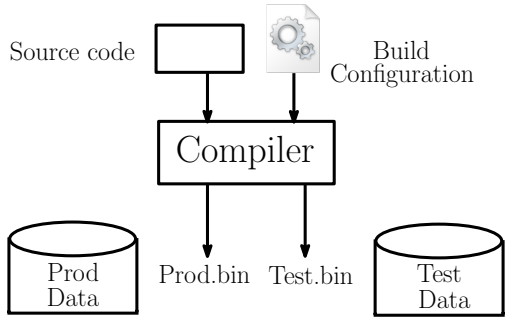

Figure 7: Producing separate binaries for test and production environments using a build configuration.

4.3 Managing Security Checks during Development and Testing

When developers disable security checks during development and testing, e.g. by utilizing self-signed certificates to bypass certificate validation checks [14], they need a way to specify that these workarounds should not be included in production releases. This separation should be ensured at compile time, which allows production releases to avoid the overhead of these workaround checks. As software projects typically maintain separate build environments for development and production, a natural way to fulfill developer need 5 is to

[4] In the wake of events that triggered mass revocations (e.g. the Heartbleed vulnerability [38]), CRLs grew by up to two orders of magnitude.

ensure that workarounds are confined to the development build environment.

For example, when using self-signed certificates during development, developers could maintain separate validation rules for certificates. The validation rules for the test environment would check *TestKeystore*, which may contain self-signed certificates. The validation rules for the production environment would check *ProdKeyStore*, which only accepts valid certificates. The build configuration defines the properties of the two build environments, as illustrated in Figure 7, and the resulting binaries use the appropriate keystores. This provides a clean separation between environments, which reduces the risk that security bypasses will propagate to production releases. This can be implemented with Maven build manager by specifying the tasks in the build file to select the correct keystore (*TestKeystore* or *ProdKeystore*) for the different build profiles corresponding to the different environments. The source code would check the certificates with the keystore that was selected during compilation. A detailed picture of the implementation is provided in Appendix C.

5. Case Studies

We conduct a qualitative evaluation of our proposed APIs, seeking to answer the questions: *Can our semantic APIs be utilized to implement non-trivial client-server?* and *Can they help prevent the common misuses of cryptography?* We consider two realistic case studies. In Section 5.1 we discuss the implementation of a mobile payments app, and in Section 5.2 we explore a range of options for implementing a secure messaging service.

5.1 Case Study 1: Mobile Money Application

Mobile money applications allow users to send and receive money without exchanging physical currency, which reduces the risk of theft. Reaves et al. analyzed seven mobile money apps for Android and reported severe vulnerabilities resulting from misuses of cryptography [25].

Security Objectives. Any mobile money application would require the user to register with the app and login to view account balances or make financial transactions. For this, the application is required to connect to the banking server and to authenticate this server. This can be achieved by ensuring proper certificate validation and verification checks are performed (the detailed TLS handshake protocol is reviewed in Appendix A). Once a secure connection is established with the banking server, the sensitive information like credit/debit card numbers, account numbers, SSNs, passwords must be sent over the secure channel. Additionally, the user's data and preferences must be stored on the device but these files are usually a system level resource that can be easily accessed by other applications. As the data may include sensitive information related to financial transactions, it is important to store these files securely.

Using Semantic APIs. We can utilize the semantic APIs proposed to isolate the security related decision making from the application developer. We establish a secure channel to the banking server, while authenticating the server correctly, by invoking the `secureConnect` function, which implements all the necessary validation and verification checks. We can then send sensitive information over this channel by using the `secureSend` function. Internally, this function ensures that an authenticated connection is established and encrypts the data with secure implementations of standard algorithms before transmitting the data. In addition, the developer implements an `isConfidential` function, which flags all the financial information and the private user information as sensitive. While we employ self-signed certificates during development and testing, this is disabled at compile time when building the production release.

Avoiding Common Errors. Reaves et al. reported that the most common mistake in practice is to disable hostname and certificate validation to be able to use self-signed certificates [25]. This leaves the application vulnerable to man-in-the-middle attacks. With our API, developers do not have the opportunity to make this mistake, as they only invoke the `secureSend` and `secureConnect` functions. The necessary checks are performed by these functions, thus preventing the developer from bypassing these checks. The flexibility to test using self-signed certificates is ensured by using a build system as described in Section 4.3.

Another common mistake is not to check whether the TLS certificate presented by the server had been revoked. This check is performed automatically by `secureConnect`, using the revocation interface provided to the library developers. The library developers can chose the method to check for revoked certificates. In the case of mobile applications downloading large CRLs is not a good choice, so we utilize OSCP stapling instead.

`secureSend` ensures that sensitive information is encrypted and sent to an authenticated endpoint. This is transparent to application developers, as the actual parameters, algorithms and keys used for encryption are not exposed. This prevents other mistakes such as the key publicly exchanged or using static keys.

Finally, because sensitive data is flagged by the `isConfidential` function, invocations of `send` or `write` on such data will fail. Sensitive data can be stored in logs and user preference files only by using `secureWrite`, and can be accessed only by legitimate users using `secureRead`.

Summary. This case study illustrates the separation of concerns provided by our APIs, which insulate the developers from the low level cryptographic operations, while still allowing them to decide which channels should be secure and which data is sensitive. The APIs also prevent the most common mistakes involved in the implementation of secure protocols and in storing data in local files.

5.2 Case Study 2: Secure Messaging

Secure messaging services aim to provide a secure alternative to sending unencrypted emails or text messages. In this scenario, the messages exchanged among users are encrypted end-to-end.

Security Objectives. Here we will classify the security goals as stated by Unger et al. defined three security goals of secure messaging applications: trust establishment, message privacy and integrity (or conversation security) [34]. Trust establishment refers to ensuring that the secure communication is established with the intended party. This can be implemented with a key exchange, along with certificate validation. Conversation security refers to the privacy and integrity of the messages sent in the presence of an active network adversary. As illustrated in [34] it is impossible to accommodate all the security goals in a single application. This is due to the fact that many times the security goals (including usability and feasibility constraints) offer a trade-off between themselves. In fact, usually it is not feasible even to accommodate the most inclusive security features together due to the trade-offs between variety of security goals. For readers interested in the further discussion on this topic we redirect them to [34].

In this case study, we consider an example application which meets the following security goals, which provide an intuitive notion of security:

- Trust Establishment
  - Network MITM protection: Prevent man-in-the-middle attacks by network adversaries. This means that, at the time of connection establishment, no third party can claim the identity of the desired end point.
  - Key Revocation Possible: Support mechanism to allow easy revocation (and/or renewal) of keys (or credentials). This refers to the capability of removal of compromised secrets used for authentication.
- Conversation Privacy
  - Confidentiality: Only the intended participants are able to read the message.
  - Integrity: Any message modified in the transit will never be accepted by an honest party.
  - Authentication: All participants are able to verify the message was sent by the claimed source.

These security guarantees suffice the required level of security by most of the common applications. Moreover, as illustrated in [34] these are also the security guarantees provided by most of the usable (and hence widely deployed) protocols and applications.

Using Semantic APIs. This level of security can be achieved by using the `secureSend` function. The developer can simply invoke the function and send the message to its destination, without explicitly choosing an encryption method and connection protocols. As all messages sent by the application are considered confidential, the insecure `send` function cannot be used.

In some cases, the applications may wish to use additional verification methods to ensure privacy from service providers or CAs. These additional measures (e.g., a secure multi-party computation algorithm called the Socialist Millionaire Protocol [34]) could be incorporated in our API by modifying the `secureConnect` function. For instance, this could be done by a library developer in a manner that is transparent to the application developer who invokes the `secureSend` or `secureConnect` functions. To implement this functionality, the library developer can invoke an additional function, beside the certificate validation checks already present. This new function would implement the checks required by the Socialist Millionaire Protocol.

Avoiding Common Errors. Owing to the separation of concerns, the application developer is never exposed to low level decision making of selecting *secure* encryption algorithms or parameters. In fact, these can be specified in the configuration file as environment specific parameters which gives the flexibility to implement different levels of security in the different settings. This configuration file is used during the build phase and can also be updated without application developers' knowledge using the Regulator pattern. Thus, this approach eliminates the mistakes caused due to complicated and inconsistent checks required to establish a secure connection for various implementations (and various protocols). It also prevents using self-signed certificates or other security workarounds in the production environment. Moreover, the Regulator pattern allows the transparent integration of standards which again deters the attacks exploiting the use of broken algorithms or parameters.

Summary. Unlike popular cryptographic frameworks like JSSE, our APIs avoid exposing low level implementation details while allowing library developers to extend the existing functionality with new security protocols. This flexibility is key for the secure messaging use case. Additionally, the level of protection against cryptographic misuse would be difficult to achieve with a library that simplifies the cryptographic APIs too much. For example, NaCl [6] provides only a `crypto_box` function for performing authenticated encryption, but does not provide methods for establishing secure connections or for integrating external information. In consequence, NaCl does not provide protection against the common mistakes involving certificate validation.

6. Related work

The related work includes empirical observations of cryptography misuse and attempts to prevent such misuse by formulating security design patterns and by simplifying the usage of cryptographic APIs.

Misuse of Cryptography. Egele et al. [13] performed an empirical study of cryptographic misuse in android applications and found that 88% of Android applications using cryptographic APIs make at least one mistake. Reaves et al. [25] studied 46 Android applications that perform financial transactions and reported instances of incorrect certificate validation, storing login credentials in clear text and using poor authentication practices such as do-it-yourself cryptography. Georgiev et al. [16] found that many widely used applications such as Amazon's EC2 Java library, PayPal's merchant SDK, shopping carts such as osCOmmerce, Ubercart, etc. and applications such as Chase mobile banking all perform broken certificate validation. They find that the root causes of these vulnerabilities are badly designed APIs of SSL implementations (such as JSSE, OpenSSL, and GnuTLS) and data-transport libraries (such as cURL) which present developers with a confusing array of settings and options. Fahl et al. [14] showed that root causes in SSL development are not simply careless developers, but also limitations and issues of the current SSL development paradigm. Acar et al. [2] perform a user study to show that participants who use Stack Overflow produced significantly less secure code than those using official documentation or published books. We add to this body of work by identifying systematic behavior differences among popular cryptographic frameworks when implementing the same functionality. However, our main focus is in presenting a solution to these problems.

Design patterns for application security and privacy. Prior work has introduced several design patterns for application security and privacy [10, 17, 27, 29–31]. The early work of Yoder et al. [36] introduced several architectural patterns for enabling application security. Using these patterns, they present a framework to build secure applications. Sommerlad [30] introduced reverse proxy patterns to protect servers at the application layer at the network perimeter. For example, the Integration Reverse Proxy integrates a collection of servers and the Front Door pattern enables single sign on on web applications. Schumacher [28] proposed patterns for protection against cookies and pseudonymous email in the seminal paper on privacy patterns, and Hafiz [17] extended this work by suggesting patterns for the design of anonymity systems. An authentication enforcer pattern [32] is used to ensure that authentication happens at all relevant parts of the code. Kern et al. [22] designed a database query API that avoids SQL injection vulnerabilities by ensuring that the query has no data flow dependency on untrusted inputs (the string parameters passed are compile-time constants), and they use a static analysis tool to enforce this property.

The effectiveness of these patterns has been questioned in recent studies. Heyman et al. [19] identify 220 security patterns, introduced over a ten year period, and Hafiz et al. [18] report duplicates among these patters, as different authors describe similar concepts but give them different names. Yskout et al. [37] quantify the benefits of security patterns to developer productivity by performing a controlled experiment, with a group of 90 students and a reduced catalog of 35 patterns. They conclude that design patterns do not reduce development time. However, they do not assess whether the use of these patterns leads to more secure code.

Rather than identify design patterns that capture common implementation techniques, our paper aims to characterize and bridge the semantic gap between the needs of developers and the cryptographic APIs. Starting from common misuse patterns of existing APIs and from incorrect recommendations found in widely used programming resources, we propose semantic APIs that can be used correctly by developers who lack an in-depth knowledge of cryptography.

Simplified usage of cryptographic APIs. The NaCl cryptographic library [6] introduced simplified APIs, aiming to avoid some misuse patterns observed with existing cryptographic libraries. NaCl provides all-in-one `crypto_box` and `crypto_sign` APIs, for authenticated encryption and digital signatures respectively, that replace sequences of library calls in OpenSSL. Cryptography.io [1] is a new cryptographic library for Python. The library's API is divided into two levels: high level cryptographic recipes, which are easy to use and do not require developers to make many decisions, and low level cryptographic primitives, which can be used incorrectly and are suitable for developers with an in-depth knowledge of the cryptographic concepts. Keyczar [33] is a simple API with safe default algorithms, modes, and key lengths and implemented in Java, Python, and C++. This cryptographic toolkit does not allow the developers to decide which algorithm to use, the key length to use, the mode of operation, how to handle initialization vectors, how to rotate keys, and how to sign ciphertexts. It simplifies the choices to an existing keyset by choosing safe defaults and automatically tagging outputs with key version information. Fahl et al. [14] propose modifying the Android OS to provide the main SSL usage patterns as a service that can be added to apps via configuration, to prevent developers from implementing their own SSL code. While these approaches been used successfully in real applications, our analysis from Section 3 suggests that the needs of developers are more diverse and that many applications require more flexibility. Rather than creating a new library or a new service, we introduce a semantic layer on top of existing libraries, which allows us to provide unified APIs across different libraries, programming languages and platforms.

The work most closely related to ours is OpenCCE [3], a tool for managing software product lines that builds on the observation that many cryptographic solutions represent combinations of common cryptographic algorithms, parameterized at compile time. OpenCCE guides developers through the selection of the appropriate algorithms, and synthesizes both Java code and a usage protocol. This approach requires monitoring the code changes over time, through static analysis, to ensure that the usage protocol is not violated. Addi-

tionally, the code synthesized is tied to the library used (for example, the subtle differences in how SSL hostname verification is performed in different JSSE libraries can lead to vulnerabilities when changing libraries) and does not account for the need to incorporate external information at runtime. Instead of static analysis tools, we propose portable semantic APIs for bridging the gap between developer needs and cryptographic expertise.

Static analysis and type systems. Recent work on type systems and static analysis aims to remove the burden of implementing security checks from developers and to generate proofs that an application satisfies certain security properties. For example, the analysis of information flow allows determining if a program satisfies certain confidentiality policies [11, 21]. Van Delft et al. [35] extended this approach to information-flow properties which change during program execution. FlowTracker [26] focuses on discovering time-based side-channels in cryptographic libraries. Bodei et al. [8] focus on finding weaknesses in cryptographic protocols. We focus on misuses of the APIs exposed by popular frameworks, rather than on attacks against cryptographic protocols. Moreover, we observe that, for some security checks, developers need the flexibility to chose the most appropriate implementation. For example, there is currently no agreement about the best method for checking the revocation status of TLS certificates, and the choice is likely to be platform dependent [24]. Type systems and static analyses are complementary to a better API design, and they can improve the performance of well established security protocols by moving the checks to compile-time.

7. Discussion

Most of the documented misuses of cryptographic APIs can be explain by the inappropriate abstractions provided to developers who lack a background in security or cryptography. However, achieving an effective separation of concerns requires understanding of the security decisions that developers must make. For example, it is not reasonable to expect these developers to know in which circumstances the SHA-1 hash function can be used securely (it is currently not recommended for digital signature generation, but it is allowed for all other applications [4]), but we should expect them to determine the sensitivity level of the data handled by their code. Other security decisions may be less obvious. In particular, developers have a legitimate need to disable security checks during development and testing, and they must also be able to select the mechanism for retrieving information about revoked TLS certificates, as there is currently no agreement about what method is best and the choice is likely to be platform dependent [24].

We infer and classify the developer needs based on the misuse cases that have been documented so far. This is only a first step toward understanding how to help developers make fewer mistakes with cryptography, as other applications may give rise to additional security needs. However, by trying to address the needs we currently understand, we identify two challenges that are specific to our domain: how to incorporate external information, at run time, and how to define compile-time checks for excluding security workarounds needed during development. This highlights the fact that a solution to the problem of cryptographic mistakes must go beyond a good API design. Additionally, providing developers with good documentation about the cryptographic framework is an important, but this is likely insufficient [2].

Another avenue for future research is to develop a method for evaluating the effectiveness of our solutions. A potential approach is to conduct an controlled experiment with two teams of programmers, similar to Yskout et al. [37]. The key challenge is to assess the impact of our solutions on the security of the resulting code (rather than on the programmer's productivity), as creating meaningful security metrics is difficult, in general [5]. Such an experiment would likely require a systematic way of finding the cryptographic vulnerabilities introduced by each programmer, for example by having a panel of experts separately inspect the code that participants write.

8. Conclusions

Modern cryptographic frameworks like Oracle JSSE, IBM JSSE, BouncyCastle, OpenSSL are widely used to develop secure applications. However, software developers who lack a background in security or cryptography often make mistakes when using these frameworks, and these mistakes usually introduce severe security vulnerabilities. We identify four root causes for these mistakes: the lack of a separation of concerns, the diverse needs of developers, the fact that security often depends on information external to the system and reliance on secure default values. We also describe five concrete needs of developers who utilize cryptographic frameworks.

Starting from these needs, we propose semantic APIs, which allow developers to make effective security decisions but do not expose low level implementation details. These APIs can be implemented consistently on different platforms by using novel and known design patterns. In particular, we propose a Regulator pattern for incorporating external information from trusted sources, e.g. the revocation status for TLS certificates or knowledge about insecure cryptographic algorithms and key lengths. We also propose compile-time checks to isolate the certain workarounds to the development build environment, in order to address the legitimate need to disable security checks during development and testing. Finally, we discuss the research avenues opened by these ideas.

Acknowledgments

We thank Jonathan Katz and Michael Hicks for early feedback on this work. This research was partially supported by the

Maryland Procurement Office (contract H98230-14-C-0127) and by the Department of Defense.

References

[1] Cryptography.io - cryptographic standard library, 2013. URL https://cryptography.io/en/latest/.

[2] Y. Acar, M. Backes, S. Fahl, D. Kim, M. Mazurek, and C. Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Security & Privacy*, 2016.

[3] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini. Towards secure integration of cryptographic software. In G. C. Murphy and G. L. S. Jr., editors, *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 1–13. ACM, 2015. ISBN 978-1-4503-3688-8. doi: 10.1145/2814228.2814229. URL http://doi.acm.org/10.1145/2814228.2814229.

[4] E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Technical Report 800-131A Revision 1, NIST Special Publication, 2015.

[5] S. Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security & Privacy*, (4), 2006.

[6] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology–LATINCRYPT 2012*, pages 159–176. Springer, 2012.

[7] J. J. Bloch. How to design a good API and why it matters. In P. L. Tarr and W. R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 506–507. ACM, 2006. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176622. URL http://doi.acm.org/10.1145/1176617.1176622.

[8] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *J. Comput. Secur.*, 13(3):347–390, May 2005. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1145948.1145950.

[9] T. M. Corporation. Common weaknesses enumeration. https://cwe.mitre.org.

[10] N. Delessy-Gassant, E. B. Fernandez, S. Rajput, and M. M. Larrondo-Petrie. Patterns for application firewalls. In *Pattern Languages of Programs Conference (PLoP)*, 2004.

[11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7): 504–513, 1977.

[12] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Proceedings of the Internet Measurement Conference*, Vancouver, Canada, Nov 2014.

[13] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[14] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60. ACM, 2013.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[16] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[17] M. Hafiz. A collection of privacy design patterns. In *Pattern Languages of Programs Conference (PLoP)*, 2006.

[18] M. Hafiz, P. Adamczyk, and R. E. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, 2007. doi: 10.1109/MS.2007.114. URL http://dx.doi.org/10.1109/MS.2007.114.

[19] T. Heyman, K. Yskout, R. Scandariato, and W. Joosen. An analysis of the security patterns landscape. In *Third International Workshop on Software Engineering for Secure Systems, SESS 2007, Minneapolis, MN, USA, May 20-26, 2007*, page 3. IEEE Computer Society, 2007. ISBN 0-7695-2952-6. doi: 10.1109/SESS.2007.4. URL http://dx.doi.org/10.1109/SESS.2007.4.

[20] L. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged SSL certificates in the wild. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 83–97, 2014. doi: 10.1109/SP.2014.13. URL http://dx.doi.org/10.1109/SP.2014.13.

[21] S. Hunt and D. Sands. On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90, Jan. 2006. ISSN 0362-1340. doi: 10.1145/1111320.1111045. URL http://doi.acm.org/10.1145/1111320.1111045.

[22] C. Kern. Preventing security bugs through software design, August 2015. URL https://www.usenix.org/conference/usenixsecurity15/symposium-program/presentation/kern.

[23] D. Litzenberger. Pycrypto - the python cryptography toolkit. URL https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.AES-module.html.

[24] Y. Liu, W. Tome, L. Zhang, D. Choffnes, D. Levin, B. Maggs, A. Mislove, A. Schulman, and C. Wilson. An end-to-end measurement of certificate revocation in the web's pki. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 183–196. ACM, 2015.

[25] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. R. Butler. Mo (bile) money, mo (bile) problems: analysis of branchless banking applications in the developing world. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 17–32, 2015.

[26] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International*

Conference on Compiler Construction, CC 2016, pages 110–120, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4241-4. doi: 10.1145/2892208.2892230. URL http://doi.acm.org/10.1145/2892208.2892230.

[27] T. Saridakis. Design patterns for fault containment. In *Pattern Languages of Programs Conference (PLoP)*, 2003.

[28] M. Schumacher. Security patterns and security standards. In *EuroPLoP*, pages 289–300, 2002.

[29] M. Schumacher. Firewall patterns. In *EuroPLoP*, 2003.

[30] P. Sommerlad. Reverse proxy patterns. In *Pattern Languages of Programs Conference (PLoP)*, 2003.

[31] K. E. Sorensen. Session patterns. In *Pattern Languages of Programs Conference (PLoP)*, 2002.

[32] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Core Series. Prentice Hall PTR, 2005. ISBN 9780131463073. URL https://books.google.com/books?id=u6tQAAAAMAAJ.

[33] G. S. Team. keyczar - easy-to-use crypto toolkit, 2008. URL https://github.com/google/keyczar.

[34] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: Secure messaging. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 232–249. IEEE, 2015.

[35] B. van Delft, S. Hunt, and D. Sands. Very static enforcement of dynamic policies. *CoRR*, abs/1501.02633, 2015. URL http://arxiv.org/abs/1501.02633.

[36] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In *Pattern Languages of Programming Conference (PLoP)*, volume 51, page 61801, 1997. URL http://www.idi.ntnu.no/emner/tdt4237/2007/yoder.pdf.

[37] K. Yskout, R. Scandariato, and W. Joosen. Does organizing security patterns focus architectural choices? In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 617–627. IEEE, 2012. ISBN 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227155. URL http://dx.doi.org/10.1109/ICSE.2012.6227155.

[38] L. Zhang, D. Choffnes, T. Dumitras, D. Levin, A. Mislove, A. Schulman, and C. Wilson. Analysis of SSL certificate reissues and revocations in the wake of Heartbleed. In *Proceedings of the Internet Measurement Conference*, Vancouver, Canada, Nov 2014.

A. HTTPS Protocol

An HTTPS connection used for connecting to web servers is HTTP protocol using an SSL/TLS connection. HTTPS is primarily used for two purposes: 1. Authenticating the server 2. Ensuring confidentiality of the message sent to the server. An SSL connection is established by the following steps:

1. First, a *hello* message is exchanged between the client and the server. The client sends all the cryptographic information such as cipher suites that it supports, the SSL/TLS protocol version it supports, etc.
2. Based on the client's information, the server responds with the cipher suite and the version of protocol that will be used.
3. The server then presents a SSL certificates to the client to prove its identity. Each certificate contains details of the server such as name, location, etc., the time for which it is valid, a public key associated with a certificate and a digital signature from root certificate authority (or an intermediate certificate authority). Root certificate authority's certificates are self signed.
4. For every certificate, the client checks whether it is correctly signed by another certificate authority or whether it implicitly trusts the certificate authority (if the certificate is self signed). In addition, the client validates each certificate's expiry date and verifies the hostnames of the certificates.
5. In addition, the client needs to verify that the certificate has not been revoked in the recent past. This is performed by one of three ways: 1. Checking with a list of revoked certificates that was updated recently (CRLs) 2. Obtaining the revocation status for every request using online certificate status protocol (OCSP) 3. The server sends a time-stamped OCSP response in addition to the certificate (OCSP stapling).
6. Once the client validates the certificate and the server is authenticated, the client and server perform a key exchange protocol to setup a symmetric secret key that would be used to subsequently encrypt communication.

B. Certificate Revocation and Broken Algorithms

Certificate Revocation. X.509 certificates are issued to a web server by certificate authorities (CAs). These certificates serve as a *proof* to the client on the server that they are connecting to. The certificates are valid for a certain duration of time after which they need to be reissued. However, due to potential key compromises and other such reasons, the CA may revoke the certificate. Thus, despite proving to the client about the existence of a certificate, the client needs to additionally verify if the certificate has been revoked.

We present some of the methods used to check for revocation in the following.

1. **Certificate Revocation Lists (CRLs):** CRL is a list of revoked certificates maintained by the CAs. To use CRLs, the client intermittently retrieves a fresh CRL from the CA and verifies a certificate against this CRL. Whether a client has the most recent list depends on the frequency of requests made to the CA and this impacts the number of queries that the CA needs to support. An intermediary can reduce load on the CA by retrieving these lists more frequently and pushing them to the clients.

```
<profiles>
  <profile>
  <id>test</id>
  <build>
  <plugins>
     <plugin>
        ....
        <executions>
           <execution>
              <phase>test</phase>
              <goals>
                 <goal>run</goal>
              </goals>
              <configuration>
              <tasks>
                <delete file = "${project.build.outputDirectory}/keystore"/>
                <copy file = "src/main/resources/TestKeystore"
                     tofile = "${project.build.outputDirectory}/keystore"/>
              </tasks>
              </configuration>
           </execution>
        </executions>
     </plugin>
  </plugins>
  </build>
  </profile>
  <profile>
  <id>prod</id>
  <build>
  <plugins>
     <plugin>
        ....
        <executions>
           <execution>
              <phase>test</phase>
              <goals>
                 <goal>run</goal>
              </goals>
              <configuration>
              <tasks>
                <delete file = "${project.build.outputDirectory}/keystore"/>
                <copy file = "src/main/resources/ProdKeystore"
                     tofile = "${project.build.outputDirectory}/keystore"/>
              </tasks>
              </configuration>
           </execution>
        </executions>
     </plugin>
  </plugins>
  </build>
 </profile>
</profiles>
```

Figure 8: A pom.xml file which selects different keystores for development and production environment.

2. **Online Certificate Status Protocol (OCSP):** OCSP is an alternative to CRLs where, for every certificate, the client verifies for the certificate's revocation status with the CA (OCSP server).
3. **OCSP with Stapling:** In OCSP stapling the server provides the certificate and client validates the certificate. If the certificate is revoked, the client then checks the time elapsed since the revocation. If the certificate is presented in some pre-defined time-frame since the revocation, the client accepts the certificate. Otherwise the client forwards the certificate to CA (OCSP server) for validation.

Avoiding the Use of Broken Algorithms. As mentioned in Section 3, NIST periodically publishes recommendations about secure cryptographic algorithms to use. Presently, adhering to these recommendations is up to application developers, leading to wide use of old broken algorithms such as MD5 hashing. Instead, we propose that a trusted authority (such as NIST) maintains an service with a list of all secure algorithms at any point in time. Every client can periodically (say once every three months) update their local list using an automated procedure. The application would hence use only those algorithms that are recently listed as secure by NIST.

C. Maven Build System

Figure 8 shows a segment of the `pom.xml` build file that defines two build profiles, `test` and `production`. The tasks for each of the profiles define which of the keystores (*TestKeystore* or *ProdKeystore*) from among the `resources` should be selected and the corresponding keystore is selected based on the profile chosen during compilation (the profile id is passed as an argument to the -P option). The selected keystore is then used by the source code to check for valid certificates.