Toward Semantic Cryptography APIs

Soumya Indela * Mukul Kulkarni † Kartik Nayak ‡ Tudor Dumitraș §
Electrical and Computer Engineering * † Computer Science ‡ Electrical and Computer Engineering §
University of Maryland - College Park
sindela@umiacs.umd.edu * mukul@umd.edu † kartik@cs.umd.edu ‡ tdumitra@umiacs.umd.edu §

***Abstract*—While several mature cryptographic frameworks exist, and have been utilized for building complex applications, developers often use these frameworks incorrectly and introduce security vulnerabilities. This stems from several challenges, including (i) an expectation that framework users understand security attacks and defenses and the subtle impact of various low level parameters; (ii) the need to take into account information external to the system to ensure security (e.g. TLS certificate revocations); and (iii) the frequent need to disable security checks during development and testing, as sometimes these checks remain disabled in production. We propose guidelines for designing cryptography APIs that are semantically meaningful for developers and that can be implemented consistently on top of existing frameworks. We also propose the Regulator design pattern, for incorporating security-critical external information, and build management hooks for isolating security workarounds needed during the development and test phases. Our API is 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.**

I. Introduction

Several mature cryptographic frameworks are currently available for implementing secure client-server communications, to store data securely or to process payments, 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 static analysis tools and are used to build real world software that provides strong security. Unfortunately, software developers often make critical mistakes when using these frameworks, including exchanging keys without authenticating the endpoint [1], storing sensitive information in cleartext [1] or with weak protection [2], using parameters known to be insecure (e.g. the Electronic Codebook mode or non-random initialization vectors) for block ciphers [2] using encryption keys that are constant [2] or are generated from insufficient randomness [1], [2], or performing improper TLS certificate validation [1], [3], [4].

The solutions that have been proposed for this problem include simplified cryptographic APIs [4]–[7], secure default values for the parameters of cryptographic algorithms [3] and using static analysis tools to discover bugs related to cryptography misuse [2], [8], [9]. These solutions do no 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. For example, different cryptographic frameworks can behave differently when providing the same functionality: a TLS handshake will fail in an IBM JSSE implementation when a trusted, but expired, certificate is provided, while in an Oracle JSSE implementation the handshake will succeed, as the developer is expected to check for expired certificates [10]. This problem could be addressed in part by following best practices in API design [11], 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 is no longer considered secure given the performance of modern hardware, and compromised TLS certificates are revoked and reissued to prevent man-in-the-middle attacks. Applications must implement runtime checks that take such external information into account, in order to use cryptography securely, but they must also have the flexibility to select the 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 [12]. Another domain specific challenge is that, 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 [4]. While these workarounds have a legitimate purpose in the development environment, they are sometimes deployed in production because the developers do not understand the security threats associated with these workarounds [1], [4].

To address these challenges, we make three contributions:

- We propose guidelines for designing *semantic APIs* for cryptographic libraries, which expose the security decisions without requiring in depth knowledge of attacks and defenses. To demonstrate that these APIs can be implemented consistently on top of existing frameworks, without the need for new cryptographic protocols or algorithms, we present a proof-of-concept implementation of secure interfaces for communication and storage (Section II).

- We describe a general *design pattern*, called *Regulator*, for transparently incorporating information from external trusted sources. We also propose three possible implementations for this pattern (Section III).
- We propose *compile-time checks* to separate the development environment from the production environment. This allows for a clean definition of security workarounds that should not be used in production.

II. Semantic APIs

Developers utilize cryptographic APIs in a variety of applications, e.g. authentication and authorization services, e-commerce SDKs and integrated shopping carts [3] or mobile payment systems [1], and they need the flexibility to implement the custom protocols required in these applications. However, we believe that only a few developers on the team should be involved in making security decisions. We call these developers *security engineers*. The rest of the developers, which we call *functionality engineers*, should be free to focus on the application logic and their design choices should not affect security.

This can be achieved by providing APIs that are semantically meaningful for the functionality engineers, such as `secureConnect` `secureSend`, `secureReceive` for establishing secure communication channels or `secureWrite` and `secureRead` for storing data securely. The APIs also include corresponding functions for communicating and storing data without authentication or encryption. Table I lists these APIs. The API implementation should ensure that confidential data is always handled by the `secure*` functions. This can be achieved by requiring security engineers to explicitly specify an `isConfidential` function, which takes a data object as a parameter and returns a boolean value, indicating whether the data is confidential. The method returns `true` by default, explicitly whitelisting data that is not sensitive. The functions that are not designed to provide security invoke `isConfidential` and return an error rather than sending confidential data over an insecure channel.

1) Functionality Engineers:

a) Communicate Interface: From a functionality engineer's perspective, the `secure*` functions would be used in the same manner as their insecure counterparts. `secureConnect/secureSend` are suitable for sending sensitive data such as login credentials, SSN, credit card numbers, etc. To send a message, the functionality engineer specifies only the address `addr` and the data `msg`, and the API implementation should hide the complexity of establishing a TLS connection (e.g. endpoint authentication, cipher suite negotiation, certificate validation). The API implementation should also ensure that confidential messages are always sent using `secureSend`. To prevent functionality engineers from accidentally sending confidential data over an insecure channel, the `send` implementation returns an error when `isConfidential(msg)` returns `true`. The security engineers defines `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.

b) Storage Interface: A `write` allows writing a value (or a file) to the storage system in plaintext whereas `secureWrite` encrypts and hashes the file and authenticates the storage system before sending data to it, to ensure integrity and confidentiality. Whether the data stored is sensitive or not, a functionality engineer only needs to specify the `data` to be stored and a `filespec` that specifies the storage location. As in the Communicate interface, the security engineers must specify an `isConfidential` function that checks whether the data is indeed non-sensitive, thus decoupling the data sensitivity checks from the application logic.

In both the interfaces, data can either be sent as only packet per connection or the connection persists till all the data packets are sent, which is a performance-security tradeoff. In the latter case, if each data packet is tagged based on the `isConfidential` function, then it would help improve the security.

2) Security Engineers: Unlike functionality engineers, security engineers implement more complex interaction protocols (e.g. authentication and authorization, online shopping, payment processing). As a consequence, these engineers need a better understanding of security principles, and are responsible for exposing intuitive APIs to the functionality engineers.

a) 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.

b) 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 library developer 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 III.

III. Integrating 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).

TABLE I
PROPOSED SEMANTIC APIS WITH FUNCTIONS FOR BOTH FUNCTIONALITY AND SECURITY ENGINEERS.

API	Parameters	Semantics	Design Pattern [13]
Functionality Engineers			
`Communicate Interface`			
`send(sock, msg)`	`addr`: address of the sender/receiver	Sends a message to the receiver	Regulator, Proxy, Template
`secureSend(sock, msg)`		Sends an authenticated encrypted message	
`receive(sock)`	`msg`: data to be sent/received	Receive a message from `addr`	
`secureReceive(sock)`		Receive an authenticated encrypted message	
`connect(addr)`	`sock`: socket for communication	Returns an established connection	
`secureConnect(addr)`		Returns a secure SSL connection (authenticity, confidentiality, integrity)	
`disconnect(sock)`		Disconnects connection	
`Storage Interface`			
`write(filespec, val)`	`filespec`: search key (or filename)	Writes data in plaintext	Proxy, Template
`secureWrite(filespec, val)`		Writes encrypted data to file	
`read(filespec)`	`val`: data to be stored	Reads data from file	
`secureRead(filespec)`		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	

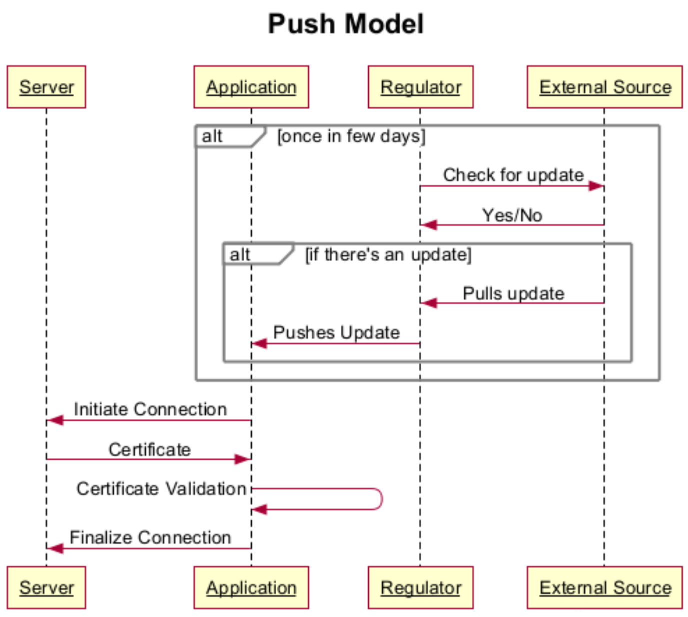

Fig. 1. Sequence Diagram showing the Push Model.

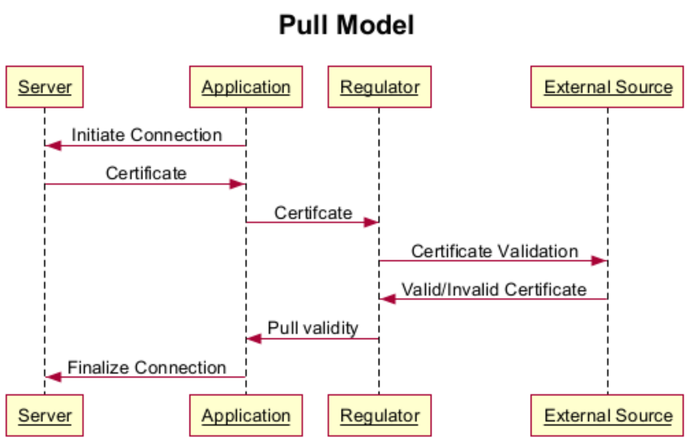

Fig. 2. Sequence Diagram showing the Pull Model.

We propose the Regulator pattern for incorporating external information about insecure design choices. Regulator is a behavioral design pattern [13], which enforces the runtime checks specified in the external information. The participants in the pattern are a trusted External Source, a Regulator object and various objects in the Application that check with the Regulator whether the external information has been updated. The External Source is a trusted third party, such as a certification authority or the National Institute of Standards and Technology (NIST). The External Source provides a machine-readable list of certificates, cryptographic algorithms, key lengths, etc. that are considered insecure. The Regulator is an object that retrieves this list and that performs the corresponding checks when requested by the Application objects. Delegating these checks to a separate object allows security engineers to decouple the runtime checks from the mechanism for retrieving the external information, allowing several implementation options. Like the traditional Observer pattern [13], Regulator defines a one-to-many relationships between objects and allows the Application objects to update themselves automatically whenever the external information changes. However, there are two differences between the Observer and Regulator patterns:

1) The *Regulator* pattern does not expect the External

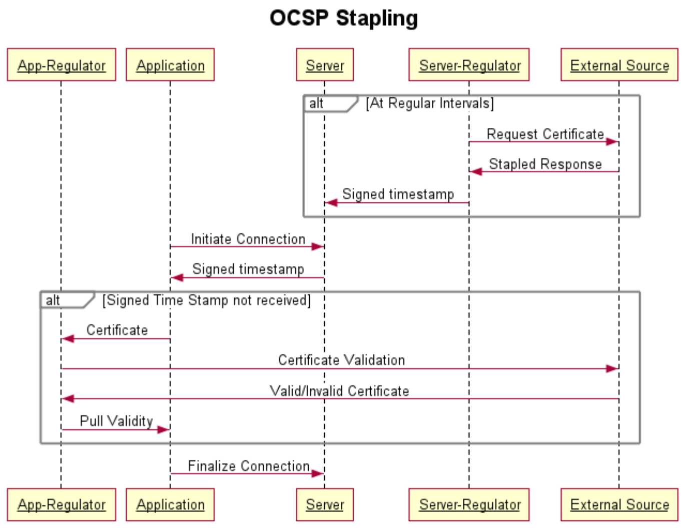

Fig. 3. Sequence Diagram showing the Selective Pull Model.

Source to be aware of all the dependent regulators, and notify the regulators whenever its state changes. Instead it is the regulators' responsibility to check if the external information has changed.

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.

We propose 3 ways of implementing the regulator pattern. These 3 implementations correspond to the different mechanisms currently used for checking for revoked TLS certificates and are suitable for different situations.

a) Push: This implementation corresponds to certificate revocation checks using CRLs and is illustrated in Figure 1. Here, the application stores a local copy of the CRL and uses it to verify whether the certificate provided by the server is valid or not. The regulator maintains a copy of CRL and updates its own copy periodically by downloading the the CRL updates from the certification authority (CA) and updates the applications' local copy. The main advantage of this method is that the application (or client) can check for the revocations locally. But, this requires the application to store the (potentially large) list locally.[1] This method can also be used in scenarios where the regulator must update the lists from multiple external sources.

b) Pull: This implementation corresponds to certificate revocation checks using the OCSP protocol and is illustrated in Figure 2. 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. This method 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, OCSP is rarely used [12].

c) Selective Pull: This implementation corresponds to certificate revocation checks using OCSP Stapling and is illustrated in Figure 3. Here, the server-side regulator obtains a signed time-stamp from the OCSP server at regular intervals, which the server sends when the application initiates a connection. If the application does not receive a timestamp, it must fall back to another method for obtaining the information (e.g. the pull method). 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.

Besides certificate revocation, these patterns can also be used in updating the list of secure algorithms; a trusted third party (e.g., the NIST, who already publishes standards for secure algorithms and key lengths [14]) could provide this information in a machine readable format, suitable for ingestion by our framework.

IV. Managing Security Checks during Development and Testing

When developers disable security checks in the development environment, e.g. by utilizing self-signed certificates to bypass certificate validation checks [4], 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. 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* that may contain self-signed certificates. The validation rules for the production environment would check *ProdKeyStore* that only accept valid certificates.

As software projects typically maintain separate build configurations for development and production, a natural way protection is to ensure that security workarounds are confined to the development build environment. For example, a Maven build profile can specify the properties of the two build environments, including environment variables that indicate which keystore should be used. The source code conditionally selects the keystore based on the environment variable. This provides a clean separation between environments, which reduces the risk that security bypasses will propagate to production releases.

V. Related work

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

a) Misuse of Cryptography.: Egele et al. [2] 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. [1] studied 46 Android applications that perform financial

[1]In the wake of events that triggered mass revocations (e.g. the Heartbleed vulnerability [15]), CRLs grew by up to two orders of magnitude.

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. [3] 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. [4] 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. [16] 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.

b) Simplified usage of cryptographic APIs.: The NaCl cryptographic library [5] 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. Fahl et al. [4] 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. 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 [8], 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. Additionally, 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.

Full disk encryption is hardware encryption for storage and is useful for devices that can be physically lost or stolen and requires storage of the encryption key. It can be used only for storage and not for transmitting data. Instead of different functionality for storage and communication, a similar function for both the interfaces would be preferable.

VI. Discussion

Most of the documented misuses of cryptographic APIs can be explained 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 that developers know the circumstances in which 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 [14]), 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 [12].

Our work is a first step toward helping developers make fewer mistakes with cryptography, as other applications may give rise to additional security needs. However, 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 [16].

A potential evaluation approach is to conduct a controlled experiment with two teams of programmers, similar to Yskout et al [**?**]. The key challenge is to assess the impact of our solutions on the security of the resulting code (rather than on the programmers productivity), as creating meaningful security metrics is difficult, in general [**?**]. 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.

VII. 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 that there is a need to incorporate external information to improve security. Based on this, 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.

Acknowledgements

We thank Jonathan Katz and Michael Hicks for their feedback on this project. This research was partially supported by the Maryland Procurement Office (contract H98230-14-C-0127) and by the Department of Defense.

References

[1] 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)*, 2015, pp. 17–32.

[2] 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*. ACM, 2013, pp. 73–84.

[3] 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*. ACM, 2012, pp. 38–49.

[4] 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*. ACM, 2013, pp. 49–60.

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

[6] "Cryptography.io - cryptographic standard library," 2013. [Online]. Available: https://cryptography.io/en/latest/

[7] G. S. Team, "keyczar - easy-to-use crypto toolkit," 2008. [Online]. Available: https://github.com/google/keyczar

[8] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, and M. Mezini, "Towards secure integration of cryptographic software," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, G. C. Murphy and G. L. S. Jr., Eds. ACM, 2015, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/2814228.2814229

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

[10] K. N. Soumya Indela, Mukul Kulkarni and T. Dumitras, "Helping johnny encrypt: Toward semantic interfaces for cryptographic framework," https://onward16.hotcrp.com/paper/29, to be published.

[11] J. J. Bloch, "How to design a good API and why it matters," in *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*, P. L. Tarr and W. R. Cook, Eds. ACM, 2006, pp. 506–507. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176622

[12] 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*. ACM, 2015, pp. 183–196.

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

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

[15] 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.

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