The W3C Web Cryptography API: Motivation and Overview

Harry Halpin
World Wide Web Consortium
Massachusetts Institute of Technology
Cambridge, MA, USA
hhalpin@w3.org

ABSTRACT

The W3C Web Cryptography API is the standard API for accessing cryptographic primitives in Javascript-based environments. We describe the motivations behind the creation of the W3C Web Cryptography API and give a high-level overview with motivating use-cases while addressing objections.

Categories and Subject Descriptors

C.v2.0 [**Computer-Communication Networks**]: General

Keywords

Web, cryptography, W3C, API, Javascript, standards

1. INTRODUCTION

The World Wide Web Consortium (W3C) has commenced work on the Web Cryptography API [5], which defines cryptographic primitives to be deployed across browsers and native JavaScript environments. This API is being driven by all major browsers with a process being open to the wider community. The API has made a number of well-motivated design choices about how to best expose cryptographic functionality to Web application developers. In Section 1 we will discuss the motivation of the API and describe how it fits into the (still evolving) Web security model. Next in Section 2, we review the use-cases that motivate the work. Then in Section 3, we will overview the API itself, followed by details in Section 4. Finally, in Section 5 we will outline future research issues. This paper is unusual insofar as its final product, the Web Cryptography API, is still open for comment. Yet it is precisely for these reasons that the W3C would like to ensure wider input from the developer as well as security community at this stage. We do not expect the core design of the API to change, although it will continue to evolve as the Web itself evolves.

2. MOTIVATION AND OBJECTIONS

There has been recently a rebirth of interest in cryptography in the browser environment. As an increasing number of applications transition to the Web, the need of ordinary users to have more secure Web applications has increased and Web developers are attempting to match those expectations. Positively, new kinds of secure Web applications that could handle high-value data present tremendous opportunities; yet simultaneously Snowden's revelations around the pervasive surveillance by the NSA have highlighted the lack of confidentiality of the current Web. A new kind of Web based on secure communication is desperately needed.

However, without the proper cryptographic primitives working cross-browser realizing such a Web is impossible. For example, the 'Cryptocat' encrypted chat application initially not only recreated their own cryptographic routines in JavaScript but also deployed these JavaScript libraries insecurely.[1] After a public workshop in 2012,[2] the W3C decided to charter the creation of a unified Web Cryptography API. The W3C Web Cryptography API has as its mission to give Web application developers the ability to write Web applications that use cryptography by exposing the cryptographic primitives already implemented in the browser to the Javascript runtime environment.

Is JavaScript cryptography doomed on the Web? Objections to cryptography in JavaScript within web browsers has been discussed in a number of blog-posts such as "JavaScript Cryptography Considered Harmful."[3] These objections can generally be phrased as objections to implementing any cryptosystem securely within JavaScript, the general security of the Web itself, or the 'host-based security' of the Web security model. Each objection will dealt with in turn.

2.1 JavaScript Cryptography

There is no a priori reason why cryptographic primitives can not be programmed in JavaScript, as exemplified by the Stanford JavaScript Crypto Library [6].[4]. Due to TLS,[5] every major web browser and operating system already contains well-verified and reviewed cryptographic algorithms. In the case of Mozilla and in some versions of Chrome, this exists in NSS.[6] Other browsers, such as Safari and Microsoft Internet Explorer, calls from the browser to cryptographic routines are passed to the underlying operating system. Thus, the Web Cryptography API simply exposes already existing and often heavily verified cryptographic functionality to Web application developers through a standardized interface. Due

[1] *https://crypto.cat/* has since fixed the problem.
[2] The workshop was called 'Identity in the Browser,' archived at *http://www.w3.org/2011/identity-ws/*
[3] At *http://www.matasano.com/articles/javascript-cryptography/*.
[4] See *https://crypto.stanford.edu/sjcl/*
[5] Transport Layer Security, formerly called HTTPS. See *http://tools.ietf.org/html/rfc5246* for more information
[6] See *https://en.wikipedia.org/wiki/Network_Security_Services*.

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2579224 .

to this constraint, the API itself is somewhat constrained practically by what features already exist cross-browser. Thus, exposing arbitrary elliptic curves is possible but they may not be implemented, even if there are well-motivated use-cases. The Working Group will keep track of supported and requested algorithms backed by use-cases in order to evolve the available functions over time.

Likewise, major changes to JavaScript itself are currently out-of-scope as we assume an off-the-shelf JavaScript environment. The mathematical functions needed to 'roll your own' cryptographic primitives such as modular arithmetic and native 'BigNum' integer support in JavaScript is not feasible at this point, as these changes would impact the entire the entire JavaScript environment and thus are more properly standardized as part of JavaScript itself in ECMA TC 39.[7] However, we provide a simple *BigNum* type definition in the Web Cryptography API, although we do not provide the underlying operations as native code. Furthermore, we do not deal with secure delete from memory, although most browsers can be run in a FIPs-compliant mode that allows them to have secure memory operations with considerable cost to performance. We purposefully also do not address is the malleability of the JavaScript run-time, where one function can be overridden by another function given the same name in a particular runtime (a technique called 'polyfill'). In the hands of responsible and trusted developers operating within the same origin, polyfilling functions is useful as it allows new experimental functions to be used in Web applications without touching the underlying code. While it is tempting to believe that the Web Cryptography API should restrict polyfilling over itself, this would hinder developers that need polyfills and fracture the JavaScript environment from the Web environment. While we can't fix all of JavaScript, we can fix the bare minimum needed to get JavaScript cryptography off the ground.

Lastly, we are producing the Web Cryptography API as a native cross-browser cryptographic library rather than encouraging the use of cryptographic browser plug-ins. While browser code and underlying cryptographic code has been heavily invested in and in the case of open-source code subject to wide review, browser plug-ins are generally not: There have been a large number of 0-day attacks on browser-plug ins, and thus while they may be resistant to cross-site scripting and other common attacks, they often expose new threats to the browser environment. So for the sake of security, most browser vendors encourage developers to write Web applications directly in JavaScript and will phase out plug-ins in the future. Equally objectionable is that the use of plug-ins to provide cryptographic materials binds the application to a single browser environment. For example, the use of a government-mandated ActiveX plug-in binds users to Internet Explorer for e-commerce in Korea, an issue that so irritated users that it has become an election issue. Ironically, this plug-in was later revealed to store its key material insecurely.

2.2 The Web Security Model

The Web Cryptography API does not change the fundamental Web Security model, wherein the same-origin policy is the fundamental security boundary. In other words, key material generated by *http://example.org* is restricted to usage to JavaScript originating from the *example.org* domain. Those who wish to go beyond this should use standardized methods such as Cross-Origin Resource Sharing[8] in conjunction with the use of *postMessage* calls. However, given that a web-browser is often running JavaScript code from multiple domains that is then invoking client-side routines such as the W3C Web Cryptography API, how can one trust that the code is actually are running from the correct domain?

This is a problem even within a single domain due to the fact that the network level must be trusted for the Web application to be trusted. Even with excellent implementations of cryptographic primitives such as the Stanford JavaScript Crypto Library, browsers still have to download via channels that be attacked. This process of downloading code could be hi-jacked by man-in-the-middle attacks while trying to create a trusted TLS connection, which has even been demonstrated by open source software such as *sslstrip*.[9] All is not lost: the larger Web Security model's various access-control mechanisms including HSTS[10] and Content Security Policy [11] allow websites to automatically switch to TLS connections without the ability of facing a man-in-the-middle attack. The work of the IETF on key-pinning[12] and Certificate Transparency [13] should allay many of the concerns over the CA system itself being compromised when using TLS. In final analysis, the Web Cryptography API provides cryptographic primitives, but the API itself does not provide the security to protect those primitives from known attacks on browsers. To mitigate against those attacks, a developer needs to use other specifications and mitigations. Even then, standards would not prevent many attacks, such as lack of proper sanitization in HTML forms leading to SQL injection attacks. However, in principle *any programming environment* is susceptible to poorly designed code. While the Web is exceptionally dangerous insofar as it is a shared and distributed space of code, nonetheless we will posit that it is possible to write JavaScript code that can be secure in all other regards except for dependencies on cryptographic primitives. This assumption is often made in other programming language environments.

2.3 Host-based security

The Web Cryptography API also rely on the client-server architecture of the Web that some in the security community find objectionable, where "your security depends entirely the security of the host. This means that in practice, CryptoCat is no more secure than Yahoo chat, and Hushmail is no more secure than Gmail. More generally, your security in a host-based encryption system is no better than having no crypto at all."[14] The Web Cryptography API is, like other Web APIs, built around a host-based model. Even in perfectly designed Web App, the behavior as regards the DOM (Document Object Model) is *completely controlled* by the host.[15]

We would argue that host-based security can be a feature, not a bug. Simply put, client security can easily be weaker than host-based security. Average users do not in general have any motivation to update their systems more often or with better safeguards than the security professionals that provide Web application hosting. The question is whether or not one should trust the updating mechanism of *any* software. One could argue that at least the user is in control of the updates on the client, while on the server-side the host can invisibly update the software. Again, the assumption is

[7] *http://www.ecma-international.org/memento/TC39.htm*.
[8] *http://www.w3.org/TR/cors/*
[9] See details of Moxie Marlinspike's well-known attack at *http://www.thoughtcrime.org/software/sslstrip/*.
[10] *https://tools.ietf.org/html/rfc6797*
[11] *http://www.w3.org/TR/CSP/*
[12] *https://tools.ietf.org/html/draft-perrin-tls-tack-02*
[13] See *https://tools.ietf.org/html/draft-laurie-pki-sunlight-12* for details.
[14] *https://www.schneier.com/blog/archives/2012/08/cryptocat.html*
[15] The DOM is the primary abstract syntax tree of HTML that is manipulated for presentation and interaction within the browser. See the HTML5 specification *http://www.w3.org/TR/html5/* for more information.

that the host may be compromised and invisibly update malicious software or break its assurances. However, this can be done easily on clients as well (for example, via a rootkit), so we should assume at least a parity in security between host and clients. There may even be reasons to believe that applications on a remote host are superior: most 0-day attacks can be performed before proper updates to client devices can correct the attack. Host-based security may scale better: It is simply easier and more secure to have hosts quickly update JavaScript once rather than deal with the inevitable lags of client-based updates (take browser plug-ins for example).

Lastly, implicit in this argument is that client devices such as the FreedomBox[16] are more secure than storing data on a server. Yet client-device seizures are likely at least as common if not moreso than server seizures and even easier in some jurisdictions than the legal compunction to release data. Mistaking *proximity for security* is a simple conceptual error. However, this does not necessarily mean that the host should have control over private key material. The key management and store of the Web Cryptography API should be implemented in such a way that it should be possible for secret key material to be stored in such a way that the server does not, if the application is built correctly, control the keys of the user. Ideally, this would allow encrypted data to be stored in the host that the host cannot decrypt.

In conclusion, much of the critique of JavaScript cryptography boils down to a critique of the security model of the Web itself, and the Web security model can evolve *in principle* to let one write secure JavaScript code. However, the key insight of the Web Cryptography is that the Web Cryptography API is actually not part of the security model of the Web. Instead, it provides *new standardized cryptographic functionality* to existing Web application development environments that can for particular use-cases improve security. Is releasing this cryptography in JavaScript to developers responsible? The potential benefits of enabling richer web-apps that can use cryptography outweighs some of the controversial points that have been raised about the availability of cryptographic APIs in the browser. Given the current dangerously insecure state of JavaScript cryptography and the fact that developers are already reimplementing cryptographic functions in JavaScript insecurely, we will have to trust the Web developer community to get better at building more secure Web applications.

3. OVERVIEW OF WEB CRYPTOGRAPHY API

The W3C Web Cryptography Working Group has prepared a group of specifications rather around the core Web Cryptography API: one use-case document [4], one normative specification that can fulfill the use-cases [5], as well as one (currently) non-normative document about key discovery [7]. We will go over the use-cases before exploring in detail the Web Cryptography API itself.[17]

3.1 Use-cases and Scope

The amount of possible cryptographic functionality that Web application developers could need is huge, ranging from simple primitives to advanced certificate functionality. Worse, a number of simple tasks are now impossible to do in a secure manner for Web developers, ranging from random number generation to the signing of client-generated documents. A core number of primitives were deemed to be necessary for any application, and these were termed *primary features*. Primary features are: key generation, encryption, decryption, deletion, digital signature generation and verification, hash/message authentication codes, key transport/agreement, cryptographically strong (pseudo)random number generation, key derivation functions, and key storage and control beyond the lifetime of a single session. Encryption and decryption includes both symmetric and asymmetric cryptography. The API must prevent or control access to secret key material and other sensitive cryptographic values and settings, a difficult task in the current browser environment.

A number of features were demanded by the community and would be up for inclusion if suitable use-cases could be found that could not be covered by primary use-cases. These include: control of TLS session login/logout, derivation of keys from TLS sessions, a simplified data protection function, multiple key containers, key import/export, a common method for accessing and defining properties of keys, and the lifecycle control of credentials such enrollment, selection, and revocation of credentials with a focus enabling the selection of certificates for signing and encryption. A number of features are explicitly out of scope, in particular those dealing with device-specific features and larger trust models, such as special handling directly for non-opaque key identification schemes, access-control mechanisms beyond the enforcement of the same-origin policy, and functions in the API that require smartcards. However, these may be revisited in future work if the Working Group gets consensus on a new charter. For more detail, please see the W3C Web Cryptography Working Group Charter.[18]

The use-cases the group has currently accepted are, as given by the use-case document [4]:

Multi-factor Authentication: A web application may wish to extend or replace existing username/password based authentication schemes with authentication methods based on proving that the user has access to some secret keying material.

Protected Document Exchange: When exchanging documents that may contain sensitive or personal information, a web application may wish to ensure that only certain users can view the documents, even after they have been securely received, such as over TLS. One way that a web application can do so is by encrypting the documents with a secret key, and then wrapping that key with the public keys associated with authorized users.

Cloud Storage: When storing data with remote service providers, users may wish to protect the confidentiality of their documents and data prior to uploading them.

Document Signing: A web application may wish to accept electronic signatures on documents. The web application must be able to locate any appropriate keys for signatures, then direct the user to perform a signing operation over some data, as proof that they accept the document. The European eID legislation and work from ABC4Trust [19] provides a number of implementation possibilities.

Data Integrity Protection: When caching data locally, an application may wish to ensure that this data cannot be modified in an offline attack. In such a case, the server may sign the data that it intends the client to cache, with a private key held by the server. The web application that subsequently uses this cached data may contain a public key that enables it to validate that the cache contents have not been modified by anyone else.

[16] *https://www.freedomboxfoundation.org/*

[17] This work presents the work of the larger W3C Web Cryptography Working Group, not just individual author who is Team Contact for the Working Group. For example, many of the key design decisions have been taken by the editor of documents. Ryan Sleevi has been the primary designer and editor of the API, with help from Mark Watson.

[18] *http://www.w3.org/2011/11/webcryptography-charter.html*

[19] See *https://abc4trust.eu/*

Secure Messaging: While TLS/DTLS may be used to protect messages to web applications, users may wish to directly secure messages using schemes such as off-the-record (OTR) messaging. The Web Cryptography API enables OTR by allowing key agreement to be performed so that the two parties can negotiate shared encryption keys and message authentication code (MAC) keys, to allow encryption and decryption of messages, and to prevent tampering of messages through the MACs.

In general, Web applications are developing towards *multi-channel* communication across multiple origins where data and even entity authentication of the client is required. This loosely-coupled design architecture should allow both data integrity and even entity authentication across networks of communicating web applications by the use of the Web Cryptography API combined with CSP and CORS. For example, digital signatures are very useful for authentication across multiple hosts, as in the case when one wishes to involve the client in signing OAuth access and refresh tokens. Not only does this defeat an attack vector of cookie-snatching attacks, it allows possibilities of possibly powerful technologies such as federated identity without HTTP-redirection (and so phishing attacks on the redirect).

3.2 W3C Web Cryptography Specifications

The use-case document has already been described and will be a non-normative deliverable [4]. The main normative deliverable is the Web Cryptography API itself [5] although there is also a non-normative document to describe key discovery [7].

Web Cryptography Use-Cases and Requirements: For each suggested new feature for the Web Cryptography API outside of the primary API features given in the scope (including currently listed secondary API features), a concrete use-case must be described and produce clearly defined requirements that are agreed upon by the Working Group. These are kept track of in a use-case document with sample code for each use-case [4].

Web Cryptography API: Commonly-used cryptographic primitives made available to Web application developers via a standardized API to facilitate their operation. This API is a set of bindings that can be thought of as equivalent in spirit to OpenSSL bindings, and provided natively as constant time functions while relying on platform-specific key storage functionality via a suitable abstraction layer. The API is described in more detail than provided here in the API document itself [5].

Web Cryptography Key Discovery: This specification describes an API for discovering named, origin-specific pre-provisioned cryptographic keys for use with the Web Cryptography API. Pre-provisioned keys are keys which have been made available to the user agent (browser) by means other than the generation, derivation, and importation functions of the Web Cryptography API. These keys are currently limited to origin-specific keys. The issue of using keys as 'super-cookies' caused the separation of Web Cryptography Key Discovery from the Web Cryptography API and so may lead it to be non-normative. [7].

It is well understood that developers may have difficulty in using the core Web Cryptography API, as the API forces a developer to explicitly provide all the default values that they wish to use. While helper functions can be provided for initialization vectors, it was felt that using defaults in the API might lead to possibly very dangerous default usage of operations as the cryptographic landscape changes over time. The Web Cryptography API assumes the user is implementing a cryptosystem where many of the defaults have already been specified in the specification of said system. However, simpler *jQuery*-like libraries will likely be evolved by the market to address the needs of most Web developers who wish to do simple tasks on a per-application basis with the correct default values, for whom the Web Cryptography API is too hard to use. For example, in the high-level API reasonable defaults for key lengths are chosen and encryption always features signing. A sample "high-level" API is further detailed in its own document and so will not be explored further here [2].

Note that the details of any user experience (such as prompts) are not be normatively specified, although they may be informatively specified for certain function calls such as exporting or access to private key material. In general, we imagine there will be cases in which the user is fine with the host controlling even the private key material, but we can imagine many more cases where the private key material must be truly secret from the host. Unfortunately, while this may be desirable for operations involving key import, export, and private material key material access, browser vendors have historically been unable to standardize elements of the graphical user interface, including the infamous 'TLS' icon case.[20] A thorough formal analysis of the security properties and threat models of the Web Cryptography API is necessary for future work before standardization is complete.[21]

Historically, each Javascript execution environment is synchronous, which presents a huge problem for computationally expensive operations such as key generation. After all, one does not want the entire operations of every Web application halted by the generation of a key. Unlike in other libraries [6], asynchronicity of operations has been tackled by API by "@@Promises/Futures"-style API design which involves including the usage of the API with DOM Events.

4. DETAILS OF THE WEB CRYPTOGRAPHY API

The Web Cryptography API is a low-level API that exposes cryptographic functionality via a number of components specified as a WebIDL. A WebIDL is a way of specifying Javascript functions, although it may also in principle be bound to programming languages outside Javascript.[22] The component Javascript features of the WebCryptography API are as follows, with much more detail given in the specification itself [5]:

1. *RandomSource*: Pseudorandom number generation.
2. *Key*: JSON object for key material, with attendant *KeyOperation*.
3. *CryptoOperation*: Finite state may machine that describes how cryptographic operations work, along with error codes.

4.1 RandomSource

The *getRandomValues* method generates cryptographically strong pseudo-random values. The *RandomSource* interface represents an interface to a cryptographically strong pseudo-random number generator (PRNG). Implementations should generate cryptographically random values using well-established cryptographic pseudo-random number generators seeded with high-quality entropy, such as from an operating-system entropy source (e.g., */dev/urandom*). Currently it provides no lower-bound on the information theoretic

[20]See the W3C Web Security Context: User Interface Guidelines *http://www.w3.org/TR/wsc-ui/*.

[21]Currently, the W3C is working with INRIA on such an analysis of key storage and threat models, and would appreciate more review from the wider community.

[22]The WebIDL specification is available here *http://www.w3.org/TR/WebIDL/*.

entropy present in cryptographically random values, but implementations should make a best effort to provide as much entropy as practicable and may provide platform or application specific entropy-related error messages. This should unify the current situation where generating random numbers from a PRNG is not distinguished from numbers that are not cryptographically strong.

4.2 Key

The *Key* object represents an opaque reference to keying material that is managed by the user agent. There are three types of keys: secret keys (opaque keying material, such as that used for symmetric encryption), public keys, and private keys (the latter two types used in asymmetric operations). Most importantly, the API does not expose key material itself, but instead only pass handlers to the key material itself in Javascript. The only exception is when a key is explicitly exported (even then, it would have the same-origin and structured clone properties). Key material that is marked *non-extractable* should have some kind of user interaction authorization when importing/exporting (within the same-origin), and access to secret key material should be forbidden. However, keys that are not marked explicitly as private, secret, or as non-extractable will be accessible to the server with same-origin policy if key export is done.

In the Web Cryptography API, we use the *structured clone* algorithm to store keys.[23] This algorithm is an abstraction on top of existing Web storage mechanisms such as *IndexedDB*[24] that has the same lifetime guarantees as the rest of Web storage. This would allow a user to clear their key material at the same time they 'wipe' cookies from their browser storage. Also, whatever security parameters around same-origin would thus apply not only to cookies and other local data given by HTML5, but to key material. For security reasons keys should in many cases not be stored in *IndexedDB*, but instead in a key store (such as provided by NSS) that has the same security properties as guaranteed by the implementation of key storage in well-known protocols such as TLS. When performing the structured clone algorithm for a *Key* object, it is important that the underlying cryptographic key material not be exposed to a Javascript implementation. Such a situation may arise if an implementation fails to implement the structured clone algorithm correctly, such as by allowing a *Key* object to be serialized as part of a structured clone implementation, but then deserializing it as a *DOMString*, rather than as a *Key* object.

4.3 CryptoOperation

The *CryptoOperation* is the heart of every cryptographic primitive. Given a algorithm and a set of parameters (usually including a handler to a key), the *CryptoOperation* will attempt to commit some operation. Every *CryptoOperation* can be thought of as a named finite state machine with an internal state, an associated algorithm, an internal count of available bytes, and a list of pending data. Every member of the list of pending data represents data that should undergo the associated cryptographic transformation if the operation as a whole is successful. The order of items when added to the list is preserved in processing, so that the first data that is added being the data processed. If the cryptographic operation fails (such as when the key type is wrong or when the algorithm is not supported), the *CryptoOperation* then terminates and produces an error code.

[23]For an explanation of this design pattern in Web standards, see *https://developer.mozilla.org/en-US/docs/DOM/The_structured_clone_algorithm*.

[24]See *http://www.w3.org/TR/IndexedDB/*

4.4 Supported Algorithms

As the API is meant to be extensible in order to keep up with future developments within cryptography and to provide flexibility, there are no strictly required algorithms. However, in order to promote interoperability for developers, there are a number of (currently) recommended algorithms: RSASSA-PKCS1-v1_5, RSA-PSS, RSA-OAEP, ECDSA, AES-CTR, AES-CMAC, AES-CFB, AES-KW, AES-CBC, HMAC, PKCS-v3 Diffie-Hellman (DH), the SHA family, CONCAT, HKDF-CTR, and PBKDF2. These should be tested in the test-suite of the Web Cryptography API so developers will be able to easily ascertain with certainty if they can use these operations across browsers. Also, the current Web Cryptography API exposes legacy cryptographic algorithms that can be used and implemented insecurely. Many consider this objectionable. Yet these are still needed in order to allow Web application developers to create applications with interoperability with widely used applications such as GPG, SSH, and the like. Lastly, as we expect the Web Cryptography Working Group to be maintained over the long-term by the W3C, any requests for new algorithms can be sent to the Working Group for consideration and discussion with implementers.

4.5 Examples

A number of examples may clarify the usage of the API. The first is to generate a signing key pair and sign some data is given in Figure 1. More examples, including that of symmetric key encryption, are given in the specification [5] and the use-case document [4].

5. CONCLUSIONS AND NEXT STEPS

In conclusion, the value of the Web Cryptography API is to enable Web applications that require features such as cryptographically strong random number generation, constant-time cryptographic primitives, and and to the best extent possible, a secure keystore. No one doubts that without these functions, JavaScript web cryptography would be doomed. By exposing the primitives already in the browser to JavaScript and providing a suitable abstraction of a key-store, the API can address the current factors that make cryptography in JavaScript impossible.

The creation of the Web Cryptography API has been heavily influenced by prior academic work such as the Stanford JavaScript Library, but also presented a number of hard practical questions where empirical research is needed [6]. In particular, we need to better ascertain the ability of developers to use cryptography APIs in general. For example, while it seems that users will generally use the highest-level of abstraction available to them, it is still an open empirical debate if one should present a 'dangerous' lower-level API or restrict ordinary developers to a 'higher-level' API. On the other hand, one could present a single API with 'appropriate' defaults. In particular, the Working Group has decided that logically it makes more sense to release a 'lower-level' API first, and given that the field could be in flux, to not in general provide any defaults in the standard. Instead, it is imagined that a 'high-level' API with appropriate defaults would be created that would build from the primitives in the Web Cryptography API. This design could be validated if there was a large-scale study of the usage of the Web Cryptography API amongst web developers attempting to solve common tasks with the API, with an eye towards common errors and mistakes with defaults. In addition, larger and more thorough studies need to be done both of deployed Web application code [6] and with developer interviews.

On the other hand, more formal research is needed on the larger framework of the Web Cryptography API and the Web security

```
var algorithmKeyGen = { name: "RSASSA-PKCS1-v1_5", modulusLength: 2048,
  publicExponent: new Uint8Array([0x01, 0x00, 0x01])};

var algorithmSign = {
  name: "RSASSA-PKCS1-v1_5",
  hash: {name: "SHA-256",} };

var keyGen = window.crypto.subtle.generateKey(algorithmKeyGen, false,["sign"]);

keyGen.oncomplete = function(event) {
  var signer = window.crypt.sign(algorithmSign, event.target.result.privateKey);
  signer.oncomplete = function(event) {
    console.log("The signature is: " + event.target.result);
  }
  signer.onerror = function(event) {
    console.error("Unable to sign");}

  var dataPart1 = convertPlainTextToArrayBufferView("hello,");
  var dataPart2 = convertPlainTextToArrayBufferView(" world!");

  signer.process(dataPart1);
  signer.process(dataPart2);
  signer.finish();
};

keyGen.onerror = function(event) {
  console.error("Unable to generate a key.");
};
```

Figure 1: Public Key Signature Example

model, and recent academic work in formal analysis of APIs such as PKCS#11 may be very useful [3]. The Web security model (as based on standards from the W3C and IETF) is woefully under-analyzed, despite its rising popularity as the preferred method of delivering even high-value applications with security implications. For example, even very basic theoretical divisions such as distinguishing between a *Web Attacker* with control of the JavaScript runtime environment in a browser and a *Network Attacker* with control over the HTTP upgrade to a TLS session have only recently been made and formally studied [1]. Currently, there is a larger problem: that the entire Web Security Model needs to be formalized and modeled, and it only makes sense formalizing the security analysis of the Web Cryptography as part of this larger analysis. Although one imagines that the API, like any API, can never guarantee absolute security, it would make sense to engage in a thorough study to be able to determine important security properties such as safe key storage in both the specification and implementations thereof.

The core design of the W3C Web Cryptography API has been finalized and satisfies all primary features [5]. However, the API will continue to change in response to open commentary and the inevitable process of discovering ambiguity that results from establishing a uniform test-suite in order to determine compliance. Thus, we would appreciate further review of the specification itself, further implementation, and concrete proposals for improving the API or future versions with a more expansive scope that may include hardware tokens and certificates. Although no API as a magic bullet, this API should allow developers to do what they could not do before: use cryptography in Web applications! The exact applications that will drive the further evolution of the Web Cryptography API are left as exercises to the wider community of Web developers and users.

6. REFERENCES

[1] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a formal foundation of web security. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, IEEE Computer Society (Washington, DC, USA, 2010), 290–304.

[2] Dahl, D. High-level Web cryptography API. Working draft, W3C, 2013. https://dvcs.w3.org/hg/webcrypto-highlevel/raw-file/tip/Overview.html.

[3] Delaune, S., Kremer, S., and Steel, G. Formal security analysis of PKCS#11 and proprietary extensions. *J. Comput. Secur. 18*, 6 (Sept. 2010), 1211–1245.

[4] Rangathan, A. Web Cryptography Use-cases. Working draft, W3C, 2013. http://dvcs.w3.org/hg/webcrypto-usecases/raw-file/tip/Overview.html.

[5] Sleevi, R. Web Cryptography API. Working draft, W3C, 2013. http://www.w3.org/TR/WebCryptoAPI/.

[6] Stark, E., Hamburg, M., and Boneh, D. Symmetric cryptography in Javascript. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, IEEE Computer Society (Washington, DC, USA, 2009), 373–381.

[7] Watson, M. Web Cryptography Key Discovery. Working draft, W3C, 2013. https://dvcs.w3.org/hg/webcrypto-keydiscovery/raw-file/tip/Overview.html.