Security Analysis of the W3C Web Cryptography API

Kelsey Cairns[1], Harry Halpin[2](✉), and Graham Steel[3]

[1] Washington State University, PO Box 442, Seattle, WA 98194, USA
kelsey.cairns@email.wsu.edu
[2] INRIA, 2 Simone Iff, 75012 Paris, France
harry.halpin@inria.fr
[3] Cryptosense, 19 Boulevard Poissonnire, 75022 Paris, France
graham@cryptosense.com

Abstract. Due to the success of formal modeling of protocols such as TLS, there is a revival of interest in applying formal modeling to standardized APIs. We argue that formal modeling should happen as the standard is being developed (not afterwards) as it can detect complex or even simple attacks that the standardization group may not otherwise detect. As a case example of this, we discuss in detail the W3C Web Cryptography API. We demonstrate how a formal analysis of the API using the modeling language AVISPA with a SAT solver demonstrates that while the API has no errors in basic API operations and maintains its security properties for the most part, there are nonetheless attacks on secret key material due to how key wrapping and usages are implemented. Furthermore, there were a number of basic problems in terms of algorithm selection and a weakness that led to a padding attack. The results of this study led to the removal of algorithms before its completed standardization and the removal of the padding attack via normalization of error codes, although the key wrapping attack is still open. We expect this sort of formal methodology to be applied to new standardization efforts at the W3C such as the W3C Web Authentication API.

1 Introduction

The World Wide Web Consortium (W3C) has commenced work on the Web Cryptography API [3], which defines cryptographic primitives to be deployed across browsers and native Javascript environments. This process has begun in the W3C Web Cryptography Working Group, driven by all major browsers and also open to the wider community.[1] Started in 2012, the W3C Web Cryptography Working Group is finalizing the standard for completion by the end of 2015, with the design being led by Ryan Sleevi of Google with Mark Watson of Netflix as co-editor. The API is already implemented across Chrome 37 and above (including Android), Mozilla version 36 and above, Opera 27 and above, Safari 8 and above,

[1] http://www.w3.org/2012/webcrypto/.

L. Chen et al. (Eds.): SSR 2016, LNCS 10074, pp. 112–140, 2016.
DOI: 10.1007/978-3-319-49100-4_5

and Internet Explorer 11 and Microsoft Edge. Thus, the W3C Web Cryptography API is the primary Web-facing cryptography API for the foreseeable future.

Like any API, the Web Cryptography API (informally called the "WebCrypto API") needs an impartial and thorough analysis to determine its security properties. Cryptographic APIs, and even cryptographic libraries such as OpenSSL, that have not received such an analysis until after widespread deployment have resulted in dangerous security incidents in validating TLS certificates [20]. Given that the W3C's mission including security reviews, the W3C explicitly worked with the larger community discover possible security vulnerabilities and formally prove the guarantees that the Web Cryptography API could provide. Due to an unfortunate tendency of Web developers to bring incorrect expectations (brought from other environments) to Javascript and to (incorrectly) believe that the Web Cryptography API 'magically' makes the Javascript browser environment a suitable platform for secure Web applications, it is important to be able to state precisely the security properties of the Web Cryptography API and what attacks are inherent in the API design as well as its operation in the Javascript browser environment. In the future, these kinds of attacks need to be mitigated so that the use of the Web Cryptography API matches intuitive developer expectations around the use of security APIs.

Section 2 explains in detail the role of formal modeling. Section 3 overviews existing background on Javascript cryptography, followed by relevant literature describing the formal analysis of APIs and Web applications. In Sect. 4, we describe the formal modeling of the Web Cryptography API using the AVISPA language, and describe the experiments we used to verify various security properties in a number of scenarios, including a successful attack on key-wrapping that can be generalized to attacks on key exchange. The behavior of key wrapping and key usages in the API would seem to violate the expectations of most developers who want to use the API. In Sect. 5 we discuss algorithm selection in the WebCrypto API, pointing out well-known errors in their algorithm selection and error codes, and these problems were accepted and our proposed fixes became part of the current WebCrypto API. In Sect. 6 we summarize what properties future standards need to improve the security properties of the Web Cryptography API in particular and the future application of formal API modeling to new standardized APIs and protocols at the W3C.

2 The Role of Formal Modeling in Standardization

In the process of standardization, there is a desire to offer as much functionality to developers as possible, while simultaneously preventing them from making mistakes. In terms of cryptographic APIs like the WebCrypto API, this can lead to handing the application developers a "kitchen sink" of cryptographic primitives, which inadvertently may give a developer "enough rope to hang themselves." Unlike protocols, APIs typically do not have precisely stated threat models and security properties. This is a mistake, as security flaws at the API level are automatically inherited by application that deploy the API and the primitives provided by the API.

Although there is a reasonable argument to give developers only “high level” APIs that may include suitable defaults, these APIs by nature must build on “low level” APIs that provide access to a large range of cryptographic primitives even if the “low level” API is not accessible or hidden from the developer. In the Web Cryptography API, it was chosen to release the “low level” WebCrypto API and not explicitly work on a “high level” cryptographic API or provide defaults. While it seems that users will generally use the highest-level of abstraction available to them, the Working Group has decided that given that the field of cryptography is in flux around issues such as elliptic curves and attacks on RSA, it would be unwise to provide any defaults that may become outdated in the standard. Instead, a ‘high-level’ API with appropriate defaults can be created that would build from the primitives in the Web Cryptography API.

The process of standardization in the field of security needs to incorporate formal methodology in order to state the security properties and discover attacks *before* a standard is released. As security standardization is difficult due to the complexity of maintaining security properties throughout the lifetime of a standardization process, there is a clear use-case for formal modeling.

The general insight behind formal modeling is that the traditional method of discovering new attacks on security APIs (and security protocols in general), by being based on human insight, may miss important attacks. While a single human may be able to discover by sheer insight an important attack, the state-space of possible combinations of items such as keys, messages, cryptographic primitives, and various desired properties may simply be difficult to discern without the assistance of automated or semi-automated tools. Similar to the automated discovery of proofs, the ideal automatic security checker would essentially be a “machine attacker” that would try out an large number of attacks using all possible combinations of cryptographic primitives and their parameters over messages in all possible states. The general technique is the reduction of maintaining security properties to a boolean satisfiability problem (SAT), where a model-checker is used to see if the security properties hold via automatically checking the property exhaustively (rather than theorem-proving) [18]. Although the problem is well-known to be undecidable, efficient SAT solvers exist for large classes of problem. Once a problem is detected via formal modeling, it may be fixed in the standard before deployment. If the standard has already been implemented, the flaw is usually then tested against real-world implementations, hopefully to be fixed once the flaw is shown to be valid.

This approach of formal modeling has shown itself to be successful against many already deployed protocols, in particular against TLS 1.2 [10]. Sometimes attacks on standards incorporate errors in the choice of cryptographic primitives, which are usually widespread in standardization as the time it takes to update. While usually the choice of a vulnerable cryptographic primitives is easily discovered, attacks on the protocol itself can be discovered years after the release of the protocol [9] due to fundamental problems in the protocol such as the lack of a well-defined state machine.

One area where formal modeling is just beginning to be applied to in standardization is in security API design. A *security API* consists of a set of functions that are offered to some other program that uphold some security properties, regardless of the program making the function calls and what functions are called [13]. For example, one would hope that an API like PKCS#11 that provides access to key material in hardware tokens would prevent any private key material from being tampered with, regardless of the application [17]. These kinds of security properties are particularly critical in many applications, and classically security APIs have been studied in the realm of hardware security modules [13].

Early work did not use generalizable formal techniques, but customized each technique for the API at hand [13]. Only more recently has fully automated analysis in terms of model-checking and theorem-proving been deployed, usually based on the Dolev-Yao (DY) abstract model where cryptographic primitives are given as functions on bitstrings in an abstract algebra [19]. This methodology has shown to be successful by its ability to compromise from non-standardized solutions such as an authentication server and steal private keys from the Yubikey USB hardware token [27]. Formal modeling has then be used to successfully reveal a number of API-based attacks on standards, including the commercially available tamper-resistant hardware security tokens PKCS#11 [17]. Currently, a large number of security APIs are under process of standardization at the IETF and W3C. Although formal modeling is not part of the current required security review of protocols in the IETF and the optional security review of protocols in the W3C, we believe it should be encouraged in the future as a mandatory part of the security review before and after implementation.

3 Background

In Sect. 3.1 we give relevant background on Javascript Web Cryptography. Section 3.2 reviews the existing academic literature on formal modeling that serves as the basis of our work on the Web Cryptography API, as well as mentioning previous usages of formal modeling for security properties on the Web. Section 3.3 summarizes the W3C Web Cryptography API (abbreviated as the "Web Cryptography API").

3.1 Javascript Cryptography

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. Although there was initial hostility to the idea of cryptography in Javascript as exemplified by "Javascript Cryptography Considered Harmful,"[2] there has nonetheless been widespread interest in creating secure Web applications [21]. Yet without the

[2] http://matasano.com/articles/javascript-cryptography/.

proper cryptographic primitives working cross-browser, Javascript cryptography would indeed be dangerous. For example, the initial version of the 'Crypto.cat' encrypted chat application initially not only recreated their own cryptographic routines in Javascript but also deployed these Javascript libraries insecurely.[3] In a remarkable turn-around, Crypto.cat has since become the first formally verified Javascript-based cryptographic application. Although a number of well-designed Javascript cryptographic libraries exist such as the Stanford Javascript Crypto Library [38], there are certain properties even the most well-designed Javascript cryptography library presents, such as the problem of accessing the library itself securely. Although well-designed libraries can prevent this, common libraries *OpenPGP.js*[4] are vulnerable to side-channel attacks and critically use built-in weak number generation given by default by *Math.random*.[5] Furthermore, even well-designed libraries that feature native Javascript password-based key derivation using algorithms such as PBKDF2 are still simply too slow for widespread high security deployment (i.e. if a sufficient number of iterations are used). After a public workshop in 2012,[6] the W3C decided to create a cross-browser Web Cryptography API that would offer a number of standardized, constant-time primitives to be accessed by Web developers. This API does not address larger concerns with the Web Security Model, such as cross-origin code injection (as currently addressed by the Web Application Security Group[7]) and completely trusted servers (i.e. Javascript as remote code execution), as well as problems inherent in Javascript itself such as prototype inheritance and the lack of availability of efficient big integer operations.

3.2 Formal Modeling Literature Review

There is still no single dominant formal modeling language for modeling security. Alloy [22], a language based on the Z specification language that uses SAT solving, has been popular and used against APIs such as the Trusted Platform Module 1.2 API [40]. It has recently been used for discovering security vulnerabilities in Web applications, although it was not used to investigate the properties of the Web Cryptography API [30]. Alloy is a well-developed framework that allows infinite models. Scyther can work with an unbounded number of sessions but does not allow the modeling of control flows [16]. ProVerif is a cryptographic protocol verifier that works as a sequence of Horn clause and allows unbounded verification on smaller protocols [11]. Tamarin also provides unbounded session support with the required mutable global state [36].

AVISPA provides automatic validation and verification of security protocols based on the DY formalism given by re-writing rules, where the knowledge of

[3] https://crypto.cat/.

[4] http://openpgpjs.org/.

[5] See the results of the 2014 penetration testing report by Cure53.de available here: https://cure53.de/pentest-report_openpgpjs.pdf.

[6] The workshop was called 'Identity in the Browser,' archived at http://www.w3.org/2011/identity-ws/.

[7] https://www.w3.org/2011/webappsec/.

the attacker can also be modeled using standard re-write rules rather than an entirely different set of rules based on, for example, belief logic. AVISPA supports multiple model-checkers over bounded sessions, and features both high and low-level formats for specifying protocols. Although unbounded sessions are of interest for some scenarios, given that in our scenarios the Web application operates over bounded sessions given the ephemeral nature of Javascript sessions (with the exception of 'cookies'). We chose AVISPA for our analysis since it takes into account mutable global state shared between sessions, i.e. in particular keys in a key store that have attributes that change over time and that affect the execution semantics of protocols for operations such as signing and encryption in an API.

Earlier work in formal analysis of the Web did conceptual work such as dividing the attacker spaces of web attackers, who attack Javascript run-time environments in the browser via cross-site scripting (XSS) attacks, from network attackers who would attack the underlying TCP/IP connections between sites and attack the certificate authority infrastructure [2]. More recent work has used Proverif to model the properties of so-called "safe" cloud storage providers via the Web [4], verifying subsets of Javascript [39], and interactive proofs of security properties of Web applications [30]. However, none of these previous works were aimed at the Web Cryptography API. This paper presents the first security analysis and formal modeling of the Web Cryptography API.

3.3 W3C 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.[8] The main features of the Web Cryptography API are as follows, with much more detail given in the specification itself [3]:

1. *RandomSource*: Pseudorandom number generation.
2. *CryptoKey*: JSON object for key material.
3. *CryptoOperation*: Functions such as encryption and wrapping, along with error codes.

RandomSource. 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. Currently it provides no lower-bound on the information theoretic 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.

[8] http://www.w3.org/TR/WebIDL/.

CryptoKey. The *CryptoKey* object represents an opaque reference to keying material that is managed by the user agent. There are three types of keys: secret keys (for symmetric encryption), public keys, and private keys (for asymmetric encryption). Most importantly, the API does not expose key material itself, but instead only pass handlers to the key material itself in Javascript and so access to secret key material is forbidden. The only exception is when a key is explicitly given a boolean *extractable* set to true and then exported (even then, it would have the same-origin and structured clone properties). Keys that are not marked explicitly as private, secret, or as non-extractable (i.e. *extractable* set to false) will be accessible to the server with same-origin policy if key export is done. A simplified (types not being given for all values) Javascript WebIDL interface for CryptoKeys is given in Fig. 1.

```
KeyType { public, private, secret };

KeyUsage { encrypt, decrypt, sign, verify,
           deriveKey, deriveBits, wrapKey, unwrapKey };

CryptoKey { KeyType type;  boolean extractable;
            object algorithm; object usages; };
```

Fig. 1. CryptoKey WebIDL

In the Web Cryptography API, we use the *structured clone* algorithm to store keys.[9] This algorithm is an abstraction on top of existing Web storage mechanisms such as *IndexedDB*[10] 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. So keys are restricted to the same origin policy in storage and are essentially ephemeral as they can be removed when session state is cleared.

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

[9] See https://developer.mozilla.org/en-US/docs/DOM.
[10] See http://www.w3.org/TR/IndexedDB/.

```
encrypt(algorithm, key, data);
decrypt(algorithm, key, data);
sign(algorithm, key, data);
verify(algorithm, key, signature, data);
digest(algorithm, data);
generateKey(algorithm, extractable, keyUsages );
deriveKey(algorithm, baseKey, derivedKeyType,
          extractable, keyUsages );
deriveBits(algorithm, baseKey, length);
importKey(format, keyData, algorithm,
          extractable, keyUsages );
exportKey(format, key);
wrapKey(format, key,wrappingKey, wrapAlgorithm);
unwrapKey(format, wrappedKey, unwrappingKey,
          unwrapAlgorithm,  unwrappedKeyAlgorithm,
          extractable, keyUsages);
```

Fig. 2. CryptoOperation WebIDL

Table 1. CryptoOperations per Algorithm

Algorithm	encrypt	decrypt	sign	verify	digest	generateKey	deriveKey	deriveBits	importKey	exportKey	wrapKey	unwrapKey
RSAES-PKCS1-v1_5	•	•				•			•	•	•	•
ECDSA			•	•		•			•	•		
RSASSA-PKCS-v1_5			•	•		•			•	•		
RSA-PSS			•	•		•			•	•		
RSA-OAEP	•	•				•			•	•	•	•
ECDSA			•	•		•			•	•		
ECDH						•	•	•	•	•		
AES-CTR	•	•				•			•	•	•	•
AES-CBC	•	•				•			•	•	•	•
AES-CMAC			•	•		•			•	•		
AES-GCM	•	•				•			•	•	•	•
AES-CFB	•	•				•			•	•	•	•
AES-KW						•			•	•	•	•
HMAC			•	•		•			•	•		
DH						•	•	•	•	•		
SHA-1					•							
SHA-256					•							
SHA-384					•							
SHA-512				•								
CONCAT							•	•	•			
HKDF-CTR							•	•	•			
PBKDF2						•	•	•	•			

produces an error code. A simplified (no types) Javascript WebIDL interface for CryptoOperations is given in Fig. 2. Each algorithm then gives support for a number of operations as given in Table 1.

Examples may clarify the usage of the API. An example generate a signing key pair and sign some data is given in Fig. 3. More examples, including symmetric key encryption, are given in the specification [3].

Supported Algorithms. Each algorithm type is given by the CryptoOperation and the key generation. Keys generated with particular algorithms thus can have

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

var algorithmSign = {
  name: "RSA-PSS",
  saltLength: 32,
  hash: {
    name: "SHA-256"
  }
};

window.crypto.subtle.generateKey(algorithmKeyGen, false, ["sign","verify"]).then(
  function(key) {
    var dataPart1 = convertPlainTextToArrayBufferView("hello,");
    var dataPart2 = convertPlainTextToArrayBufferView(" world!");
    return window.crypto.subtle.sign(algorithmSign, key.privateKey)
      .process(dataPart1)
      .process(dataPart2)
      .finish();
  },
  console.error.bind(console, "Unable to generate a key")
).then(
  console.log.bind(console, "The signature is: "),
  console.error.bind(console, "Unable to sign")
);
```

Fig. 3. Public Key signature example

their usages restricted to only those CryptoOperations permitted by the algorithm. 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. 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 algorithms that the API supports by default: 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. RSAES-PKCS1-v1_5 was supported but removed due to attacks described in this paper, see Sect. 5. These will 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.

4 Formal Analysis

4.1 Threat Model

The threat model needs to be realistic in terms of actual attacks on the Web, and not too powerful. If we assume the origin is completely untrusted or compromised by an attacker, then the attacker can easily steal the application's secrets directly before they are encrypted. Thus, we assume the origin is trusted when the WebCrypto API is initialized and secrets are successfully encrypted and stored on the client.

Our threat model is then a temporary compromise of the Javascript environment being used by the server or client after secrets have been encrypted by WebCrypto and stored on the client. This accurately models most cross-site scripting (XSS) attacks on the Web, including DOM-based attacks on the client and temporary compromises of Javascript delivered by the server.

The security property that we want to maintain is that access to the raw key material that is private, secret, or explicitly typed as non-extractable should not be accessible to Javascript. These keys should only be accessible to a server with same-origin policy if key export is explicitly done to extractable key material.

The goal of the attacker is to retrieve previously encrypted secrets. This threat model's assumptions are built into our formalization, as seen from the rule definitions in Fig. 4. The inputs and outputs to each rule are either known by the attacker or stored on the client device.

4.2 Model

The models we used were constructed using the AVISPA toolset,[11] which was built to enable easy translation from protocol to model. The AVISPA toolset forms a hierarchical set of languages which take in a high-level protocol description and translate it through a series of steps to a low level description that functions as input to a model checking engine. Since AVISPA's high level language is tailored towards protocols and not API's, we designed our models in AVISPA's intermediate format (IF). AVISPA's IF format describes protocols modeled as an infinite state machine whose semantics is given via set re-writing.[12] Protocols are described unambiguously by sets of typed predicates which define states and rules which define state transitions. For example a predicate might take the following form:

$$keystore(K) : key \rightarrow fact$$

Which represents a *fact*-type predicate relating to a variable K of type *key*. States are defined by a list of applicable predicates. Transition rules take the form of having a list of predicates on the left hand side which must be true for the transition to occur. The right hand side lists predicates which are true

[11] http://www.avispa-project.org/.

[12] The formal semantics of AVISPA's higher-level HLPSL that subsumes IF are out of scope but are given here: http://www.avispa-project.org/delivs/2.1/d2-1.pdf.

$generateKey(key\ K, type\ T):$
$\quad \rightarrow keystore(K,T) \wedge usages(K,T) \wedge extractable(K,T)$

$importKey(key\ K, type\ T):$
$\quad iknows(K)$
$\quad \rightarrow keystore(K,T) \wedge usages(K,T) \wedge extract(K,T)$

$extractkey(key\ K, type\ T):$
$\quad keystore(K,T) \wedge extract(K,T)$
$\quad \rightarrow iknows(K,T)$

$encrypt(key\ K, type\ T, message\ M):$
$\quad keystore(K,T) \wedge encryptUsage(K) \wedge pub(T)$
$\quad \rightarrow iknows(crypt(K,M))$

$sencrypt(key\ K, type\ T, message\ M):$
$\quad keystore(K,T) \wedge encryptUsage(K) \wedge sym(T)$
$\quad \rightarrow iknows(scrypt(K,M))$

$decrypt(key\ K, type\ T, message\ M):$
$\quad keystore(K,T) \wedge decryptUsage(K) \wedge iknows(crypt(K,M)) \wedge priv(T)$
$\quad \rightarrow iknows(M)$

$sdecrypt(key\ K, type\ T, message\ M):$
$\quad keystore(K,T) \wedge decryptUsage(K) \wedge iknows(scrypt(K,M)) \wedge sym(T)$
$\quad \rightarrow iknows(M)$

$sign(key\ K, type\ T, message\ M):$
$\quad keystore(K,T) \wedge signUsage(K) \wedge priv(T)$
$\quad \rightarrow iknows(crypt(K,M))$

$verify(key\ K, type\ T, message\ M):$
$\quad keystore(K,T) \wedge verifyUsage(K) \wedge iknows(crypt(K,M)) \wedge pub(T)$
$\quad \rightarrow iknows(M)$

$wrap(key\ K, type\ T, key\ WK):$
$\quad keystore(K,T) \wedge wrapUsage(K) \wedge pub(T) \wedge keystore(WK) \wedge extract(WK)$
$\quad \rightarrow iknows(crypt(K,WK))$

$swrap(key\ K, type\ T, key\ WK):$
$\quad keystore(K,T) \wedge wrapUsage(K) \wedge sym(T) \wedge keystore(WK) \wedge extract(WK)$
$\quad \rightarrow iknows(scrypt(K,WK))$

$unwrap(key\ K, type\ T, key\ WK, type\ WT):$
$\quad keystore(K,T) \wedge unwrapUsage(K) \wedge iknows(crypt(K,WK)) \wedge priv(T)$
$\quad \rightarrow keystore(WK,WT) \wedge extract(WK) \wedge usages(WK)$

$sunwrap(key\ K, type\ T, key\ WK, type\ WT):$
$\quad keystore(K,T) \wedge unwrapUsage(K) \wedge iknows(scrypt(K,WK)) \wedge sym(T)$
$\quad \rightarrow keystore(WK,WT) \wedge extract(WK) \wedge usages(WK)$

Fig. 4. Model for each API call. Note that all usages are allowed for created and imported keys, simplifying the model and giving the advantage to the attacker.

following the transition. The following shows an example rule which models encryption:[13]

$do\_encrypt(M, K) :=$
$\quad private\_data(M) \wedge keystore(K)$
$\quad \Rightarrow private\_data(scrypt(K, M))$

Initial states are described by declaring initial terms and predicates on them. Lowercase letters are used to represent instantiated terms. Uppercase letters denote free terms that may be bound to instance of the same type.

$k, K : key$
$m, M : message$

Initial predicates use instantiated terms:

$private\_data(m)$
$keystore(k)$

This example would initialize a state machine with the predicates $keystore(k)$ and $private\_data(m)$. The *do_encrypt* rule is applicable when $M = m$ and $K = k$.

AVISPA assumes an attacker following the standard DY model (where the attacker is called the "intruder") and is represented functionally by an *iknows* predicate which dictates information known to the attacker. Further, the attacker has basic cryptographic capabilities. For example, the following rules would be applicable to the attacker independently of the modeled protocol:

$i\_encrypt(M, K) :=$
$\quad iknows(M) \wedge iknows(K)$
$\quad \Rightarrow iknows(scrypt(K, M))$
$i\_decrypt(M, K) :=$
$\quad iknows(scrypt(K, M)) \wedge iknows(K)$
$\quad \Rightarrow iknows(M)$

Consistent with the DY model, information communicated over the channel between actors is predicated with *iknows*. Thus, inputs to rules may be attacker created values and outputs are assumed to be learned to the by the attacker. This paradigm allows us to model compromised Javascript, where inputs may come from any source and outputs may be sent anywhere. The only state accessible to the API is the keys stored on the host, which we modeled with a *keystore* predicate. The attacker in this model uses keys stored on the host. Our API rules use *iknows* or *keystore* to predicate inputs:

$api\_encrypt(M, K) :=$
$\quad iknows(M) \wedge keystore(K)$
$\quad \Rightarrow iknows(scrypt(K, M))$

The attacker goal states specify the conditions of a successful attack. For example, an attacker goal when testing confidentiality would be defined as a state in

[13] Throughout this paper we omit many AVISPA-specific constructs in order to focus on the underlying model. This includes statements that are necessary for modeling protocols but not APIs, but will nonetheless cause errors if omitted. The complete rules are available here: http://www.w3.org/2012/webcrypto/webcrypto_if_files.tgz.

which both the *iknows* predicate applies to a variable already declared secret by the *secret* predicate, for example:

$$\text{Goal}: \quad secrecy(M) := iknows(M) \wedge secret(M)$$

4.3 API Model

To test properties of the API, we built a general API model which we then varied slightly to perform different tests. Creation of the general model includes custom predicates, transition rules representing API calls, and handling of key objects. The API call transition rules are built from both AVISPA's default predicates (*crypt, scrypt, iknows*, etc.) and custom predicates. The modeling for each rule is described in Fig. 4.

In addition to AVISPA's default predicates, several custom predicates were necessary to handle the modeling of key objects. The actual CryptoKey objects associates raw key data and the following set of attributes:

Type	Public, private or secret (symmetric)
Extractable	A boolean specifying whether the key material may be exported to Javascript
Algorithm	The algorithm used to create the key
Usages	attributes which specify the key's allowed operations

Our modeled CryptoKey objects only represent the parts of the actual CryptoKey object. For efficiency reasons, our model expresses keys as *(type, value)* pairs. A key's attributes (*extractable, usages*) are represented by inclusion of that key in a set representing the particular attribute. For example, all keys with the *encrypt* usages are contained in a set named *Javascript_encrypt*. We ignore the algorithm attribute in our model.

Each entity is associated with a store of keys known to that entity. Each WebCrypto operation requires that the keys it will use be present in its associated entity's key store. Some operations (*generate, import, unwrap*) will add a key to the key store.

WebCrypto calls were translated directly into transition rules for our model. The predicates used are a combination of AVISPA defined (*crypt, scrypt, iknows*, etc.) and some that were specifically defined for this model. The predicates we defined are:

$keystore(K, T)$:	key is stored in local storage
$extract(K)$:	in extractable set
$usages(K)$:	all usages apply to key
$xUsage(K)$:	usage x applies to key
$sym(T)$:	key type is symmetric
$pub(T)$:	key type is public
$priv(T)$:	key type is private

Modeling Specific Scenarios. Each individual scenario was created by customizing the models initial state and attack goal. After this step is done, the discovery of attacks is then fully automated by AVISPA. Some scenarios also included additional transition rules which allow more control over the behavior of the model. The additional rules serve as "unit operations" for each scenario. These operations model the equivalent of a sequence of individual API operations. Building unit operations for each test had two advantages. First, it optimizes the number of steps needed by the model checker in order to find attack sequences that include this sequence of steps. Second, constraints can be added to the model which require any found attack sequences to contain these operations. This allows modeling a scenario with the requirement that either the server or client fulfilled their role properly. A large number of scenarios were formalized, building up from simple to more complex in terms of properties by the use of these unit operations.

As an example, we look at the model used to check confidentiality of wrapped key exchange messages sent from client to server. This model is initialized with three key objects. The intent is to model two keys that belong to the client: one (*swkey*) for wrapping and the other (*skey*) to be exchanged securely. The third (*ikey*) key is known to the attacker and can be used in whatever way aids the attacker:

$$\begin{aligned}\text{Instance Variables}: \; & skey, swkey, ikey : key\\ & st, iwt, it : type\end{aligned}$$

$$\begin{aligned}\text{Initial State}: \; & sym(st) \wedge secret(skey) \wedge secret(swkey)\\ & \wedge keystore(skey, st) \wedge keystore(swkey, swt)\\ & \wedge keystore(ikey) \wedge usages(ikey)\\ & \wedge iknows(ikey) \wedge extract(skey)\\ & \wedge wrapUsage(swkey) \wedge unwrapUsage(skey)\end{aligned}$$

The predicates in the initial state describe the properties of the keys using the predicates as described earlier. The goal state for this case was described by:

$$\text{Goal}: secrecy(K) : secret(K) \wedge iknows(K)$$

This goal specifies that for some variable key K, K has been defined to be both secret and known by the attacker. This goal was trivially achieved because *extract*(*skey*) lets a secret key be marked as extractable which allows the attacker to export *skey* and learn its value.

To force the model to find attack sequences that show how export attacks can effect operations such as key exchange with the key being explicitly extractable (as would be the case with secret key material by default), we modified our model slightly. First, we remove *extract*(*skey*) from the initial state. Next we added a *c_send*() unit operation which wraps and sends a key without requiring either keys to be extractable:

$$\begin{aligned}& c\_send(key\ K, type\ T, key\ WK, type\ WT) :\\ & \quad keystore(K, T) \wedge wrapUsage(K) \wedge keystore(WK, WT)\\ & \quad \Rightarrow iknows(\{WK\}_K) \wedge has\_sent(K) \wedge has\_sent(WK)\end{aligned}$$

The *has_sent* fact is used to force this rule to be used. This is accomplished by modifying the goal state to be require that $has\_sent(K)$ be true, which can only happen after the *c_send* rule is used:

$$\text{Goal}: \ secrecy(K) : secret(K) \wedge iknows(K) \wedge has\_sent(K)$$

The attack found by the model checker for this set of modification is discussed in Sect. 4.4.

4.4 Tests and Results

We tested security properties by systematically modeling different use cases and assessing the resulting attacks. The attacks we found existed due to potentially unintuitive traits of the API, which would have negative security implications if misunderstood by a large audience. The interesting attacks fell into two types:

- **Export Attack:** Exporting extractable key data and changing usages.
- **API Attacks:** Using client API calls to recover clear text of encrypted communication via an attack on key wrapping.

To summarize, our analysis found that keys managed by the API, if wrapped and then unwrapped, then lose their usage properties. In particular this can be used to subvert operations such as key exchange and so reveal private key material.

Export Attack. While unextractable keys are appear safe, our attack shows there are no safeguards in place to preserve the usage attributes on extractable keys. Furthermore, any wrapped key can be unwrapped and then given arbitrary usage attributes. Thus, there is no guarantee that a key transmitted by wrapping will be used with the intended usages.

The test that revealed this property was modeled with a client initialized with two symmetric keys. One was an unextractable key with the wrap and unwrap usage enabled. The other key was extractable but had no usages enabled. The initial state and goal state are given below, where *skey* is the secret key and *ikey* is the key being under possession of the attacker (note that *i* is used as the "attacker" is called "the intruder" in AVISPA):

$$\begin{aligned} \text{Instance Variables}: \ & key, ikey : key \\ & st : type \end{aligned}$$

$$\begin{aligned} \text{Initial State}: \ & sym(skey) \wedge sym(ikey) \\ & \wedge keystore(skey, st) \wedge keystore(ikey, st) \\ & \wedge extract(skey) \wedge usages(ikey) \end{aligned}$$

$$\text{Goal}: \ addUsage() : encryptUsage(skey)$$

Not only the *encrypt* usage, but all usages could be added simply by wrapping and unwrapping the extractable key: $wrap(skey, ikey)$, $unwrap(skey, ikey)$. This simple single-host attack extends to wrapped keys transmitted between multiple hosts, and demonstrates the lack of control over usages: Once a key has been wrapped, the original usages with which it was created are lost, and new usages, as well as the choice to designate a key extractable, can be added during the unwrap operation.

Key Exchange API Attacks. The test case in Sect. 4.4 revealed the lack of key attribute preservation, and an attacker can be successful in deploying this strategy to reveal secret key material in key exchange and message passing protocols that use the WebCrypto API. A set of experiments, also done with the AVISPA model, involved keys sent between a client and server using various combinations of authentication and key wrapping.

Enumerating these cases also gives us insight into the security of general message exchanges based on WebCrypto: As key wrapping is a composition of export and encrypt, if an attack existed on a wrapped key, then the same attack would apply to an encrypted message. The combinations of encryption and authentication our model discovered compromises in are:

Symmetric encryption – The sender wraps the key using a symmetric key shared with the receiver who unwraps the key

Asymmetric encryption – The sender wraps the key using public key for the receiver who unwraps with the corresponding private key

Symmetric encryption with asymmetric signing – The symmetric encryption case augmented by signing with the sender's private key

Asymmetric encryption with asymmetric signing – The asymmetric encryption case augmented by signing with the sender's private key

Each test was initialized with enough keys to allow the client and server's task to be modeled as well as the attacker. We modeled multiple versions of each scenario: one matching the current API specification and a second restricted version designed to show changes that could reduce attacks. The attacks are described in a number of tables. Operations in the attack sequences are prepended with an identifier specifying the entity that performed the operation: **ijs-** malicious Javascript controlled by the attacker, **i-** the attacker, **c-** the client Javascript running honestly, and **s-** the honest server.

Table 2 shows attacks found by testing confidentiality of keys sent from client to server. A successful attack involves the attacker learning a key that was also defined as secret. In the cases using symmetric encryption, the basic model used a symmetric wrapping key that had both wrap and unwrap usages enabled. These cases allowed API attacks where the secret key was unwrapped and given export privileges and then extracted. The restricted cases were modeled by removing the unwrap usage from the client's wrapping key, which removed this attack as well as the export attack on the key. The asymmetric case did allow export attacks but not API attacks.

Table 2. Client → Server confidentiality attacks

Scenario	Export	API
Symmetric Encryption		
Single key for wrap and unwrap	Yes	*c-send, ijs-unwrap, ijs-extractKey*
Different key for each direction	Yes	*None*
Asymmetric Encryption		
No Restrictions	Yes	*None*
No key extraction	None	*None*
Symmetric Encryption with Asymmetric Authentication		
No Restrictions	Yes	*c-send, i-verify, ijs-unwrap, ijs-extractKey*
Client wrapping key cannot unwrap	None	*None*
Asymmetric Encryption with Asymmetric Authentication		
No Restrictions	Yes	*None*
No key extraction	None	*None*

Table 3. Server → Client confidentiality attacks

Scenario	API
Symmetric Encryption	
No Restrictions	*s-send, ijs-unwrap, ijs-extractKey*
Different keys for wrap and unwrap	*s-send, s-unwrap, s-extractKey*
Asymmetric Encryption	
No Restrictions	*s-send, ijs-unwrap, ijs-extractKey*
Different keys for wrap and unwrap	*s-send, s-unwrap, s-extractKey*
Symmetric Encryption with Asymmetric Authentication	
No Restrictions	*s-send, i-verify, ijs-unwrap, ijs-extractKey*
Asymmetric Encryption with Asymmetric Authentication	
No Restrictions	*s-send, i-verify, ijs-unwrap, ijs-extractKey*

Table 3 covers confidentiality attacks but this time for keys sent from server to client. In these scenarios all base cases were susceptible to an API attack which caused the key received from the server to be imported as extractable and then immediately exported. No modifications were found which prevented this attack.

Table 4 shows integrity attacks on the same set of scenarios as Table 2. The successful attack was modeled as a key, originally known only to the attacker, being stored in the server's key store. For most cases, both symmetric and asymmetric, API attacks allowed an attacker to send a key to the server by importing that key into the client and using API calls to wrap and possibly sign the key. The only modification we found preventing this attack was to disallow use of one of the keys, but this may not be practical in real world use cases.

Table 4. Client → Server integrity attacks

Scenario	API
Symmetric Encryption	
Single key for wrap and unwrap	*ijs-importKey, ijs-wrap, s-receive*
Different key for each direction	*ijs-importKey, ijs-wrap, s-receive*
Asymmetric Encryption	
No Restrictions	*ijs-encrypt, s-receive*
Signing key removed before malicious code runs	*None*
Symmetric Encryption with Asymmetric Authentication	
No Restrictions	*ijs-importKey, ijs-wrap, ijs-sign, s-receive*
Client wrapping key cannot unwrap	*None*
Asymmetric Encryption with Asymmetric Authentication	
No Restrictions	*ijs-importKey, ijs-wrap, ijs-sign, s-receive*
Signing key removed before malicious code runs	*None*

Table 5. Server → Client integrity attacks.

Scenario	API
Symmetric Encryption	
Same key for wrap and unwrap	*ijs-importKey, ijs-wrap, c-receive*
Different keys for wrap and unwrap	*None*
Asymmetric Encryption	
No Restrictions	*i-wrap, c-receive*
Symmetric Encryption with Asymmetric Authentication	
No Restrictions	*None*
Asymmetric Encryption with Asymmetric Authentication	
No Restrictions	*None*

The integrity attacks shown in Table 5 on keys sent from server to client yield fewer API attacks. API attacks exist for the cases where the attacker has access to the wrapping key. This is the symmetric case where the client's key has wrap and unwrap usages as well as the asymmetric case where the encryption key is public by default. With authentication required, no API attacks were found.

These results lead to a few general observations. Export attacks are often available because keys that can be wrapped are also then extractable; any key that can be exported from the client can be retrieved in the clear by an attacker even though the wrapping is intended to keep the key secret. The found API attacks have a common element of using a key stored on the client to perform cryptographic operations. Some of these attacks are caused by the fact that the *extractable* attribute and *usages* array are not preserved for wrapped keys, and unwrapped keys can be given any new combination of attributes, including *extractable*. Other attacks could be mitigated by limiting the usability of stored keys. For example in the symmetric encryption case, if one key is used for both directions, the attacker can use the client's keys to both encrypt and decrypt the communication. However, using distinct keys for each direction of communication and reinforcing this behavior with *usages* attributes prevents this type of attack assuming the usages are not changed. Thus, the successful API attacks could be prevented if usages were bound to key material in general and not allowed to be altered while the key is being stored. Lastly, authenticating via asymmetric keys where extractability of key material is not allowed prevents the attacks on confidentiality and integrity of keys from the server to the client.

5 Algorithm-Level Analysis

In our formal analysis, we treated algorithms as "black boxes" in the analysis of cryptographic primitives. This is because some of the attacks on security APIs are beyond the scope of the DY model employed by AVISPA. For example, formal models do not in general deal with attacks like *oracle attacks* that observe the error messages that are returned by the API. Furthermore, some algorithms have well-known weaknesses.

In this review, we limit ourselves to peer-reviewed results on the algorithms which have been included in the first Candidate Recommendation version of the WebCrypto API, although the precise algorithms are still in flux due to interoperability testing. Table 6 summarizes the results. Although none of these results or attacks are new in terms of cryptanalysis, the fact that they were present in the WebCrypto API should be explicitly noted. After this analysis, RSAES-PKCS1-v1_5 was removed from the specification and the problems with padding error return codes were corrected.

There is at least one annual publication, the ENISA "Algorithms, Key Size and Parameters Report," whose aim is to track ongoing developments, which discusses a much larger set of algorithms in much greater depth. Our results are in general the same except for algorithms ENISA does not cover like PBKDF2 and AES-KW [37].[14] We note that HKDF has security proofs [26] but needs more study. Security models for password-based key derivation functions are still in a state of flux [42]. PBKDF2 has known weaknesses [43], and many implementations do not use enough iterations.

In detail, the main problematic algorithm originally included in WebCrypto was *RSAES-PKCS1-v1_5*, which has been known to be vulnerable to a chosen ciphertext attack (CCA) since 1998 [12]. The attack has recently been improved to require a median of less than 15 000 chosen ciphertexts on the standard oracle [5]. Instances of the attack in widely-deployed real-world systems continue to be found [23]. Finally, note also that as of version 1.3, RSAES-PKCS1-v1_5 will be dropped from the TLS standard.[15] In terms of alternatives, there are no publicly known attacks on RSASSA-PKCS1-v1_5 but no security proofs and

[14] Note as of September 2016, the 2014 report is currently under revision.

[15] http://www.ietf.org/mail-archive/web/tls/current/msg12362.html.

Table 6. Algorithm summary

Algorithm/Mode	Ok legacy	Ok future	Note
RSAES-PKCS1-v1_5	×	×	
RSA-OAEP	✓	✓	
RSASSA-PKCS1-v1_5	✓	×	No security proof
RSA-PSS	✓	✓	
ECDSA	✓	×	Weak provable security results
ECDH	✓	✓	
AES-CBC	✓	✓	NB not CCA secure
AES-CFB	✓	✓	NB not CCA secure
AES-CTR	✓	✓	NB not CCA secure
AES-GCM	✓	✓	
AES-CMAC	✓	✓	
AES-KW	✓	×	No public security proof
HMAC	✓	✓	
DH	✓	✓	
SHA-1	×	×	See text
SHA-256	✓	✓	
SHA-384	✓	✓	
SHA-512	✓	✓	
CONCAT	✓	✓	
HKDF-CTR	✓	✓	
PBKDF2	✓	×	Known weaknesses (see text)

no advantages compared to other RSA-based schemes, while RSA-PSS has a security proof due to Bellare and Rogaway [8] in the random oracle model.

There are also some inevitable issues with elliptic curve cryptography, which is in an ongoing state of flux in both WebCrypto and wider internet standards. In particular, ECDSA has some provable security results but only in weak models [42]. There is debate on elliptic curves.[16] ECDH has provable security results [14], but like other plain DH modes it offers no authenticity, so this must be handled separately. A proposal exists to include Curve25519 [32] after the browsers are finished implementing the CFRG recommendations. In general, we recommend using only named curves with wide public review.

In terms of AES, there are well-known issues with *AES-CBC* mode that are not currently believed to pose a practical threat [25], and it is not CCA secure. Both *AES-CBC* and *AES-CFB* are secure against chosen plaintext attacks (CPA-secure) if the IV is random, but not if the IV is a nonce [35]. In particular *AES-CFB* does not tolerate a padding oracle [41] - indeed, in practice,

[16] http://safecurves.cr.yp.to/.

padding oracle attacks are common [29,31,33]. The padding mode [24] is exactly that which gives rise to most of these attacks. *AES-KW* has received various criticisms, for example being inconsistent in its notions of security (requiring IND-CCA from a deterministic mode), but though it has no public security proof, it has no known attacks either [34]. *AES-CTR* is probably the best mode of the traditional AES modes, although the mode is easy to mis-use and thus in general *AES-GCM* should be preferred (ideally with an explicit safeguard to prevent re-usage of the IV). Since WebCrypto does not contain guidance on composing AES modes with a MAC and does not prevent the re-usage of an IV, care needs to be taken by developers.

Due to the inclusion of AES-CBC and the consideration of RSAES-PKCS1-v1.5, padding attacks against these protocols would be a threat to both encrypted

Table 7. Explanation of padding attacks

	Attacking Encrypted Text	Attacking Wrapped Keys
PKCS1-v1.5	**Potential Attack** – PKCS1-v1.5 padding is susceptible to known oracle attacks when an attacker can discern that decryption failed due to incorrect padding. The API specifies that failure to decrypt should result in a `OperationError`. Causes of this failure are incorrect padding (either incorrect leading bytes or not enough padding) and a cipher text that is out of range of the RSA modulus. (The latter can be prevented in the attack.) These are the only possible causes of the `OperationError` from PKCS1-v1.5 decryption, leading to the possibility that a decryption oracle is exposed to the attacker	**Potential Attack** – Similarly to the attack against encryption, the error given when unwrapping an incorrectly padded key is an `OperationError`. However, the error that results from a correctly padded but incorrectly formatted key (which would be used in the attack) is a `DataError`. If the difference in errors in not concealed from attackers, an attack would be able to recover wrapped keys
AES-CBC	**Potential Attack** – AES-CBC is known to be susceptible to padding oracle attacks when an attacker can discern that a particular cipher text cannot be decrypted due to a padding error. The API specifies that this error is a `DataError`. The only other source of this error during the *decrypt* operation is an incorrect initialization vector length, which the attacker could check given access to the IV	**No Obvious Attack** – A successful attack requires the ability to differentiate between keys that cannot be unwrapped due to 1) incorrect padding and 2) incorrect key length or structure that cannot be parsed. In both cases, the error specified by the API is the same and no other test is apparent to distinguish between the two

messages and wrapped keys in WebCrypto. Table 7 explains how these vulnerabilities manifest themselves in the Webcrypto API. After these attacks were discussed with the W3C Web Cryptography Working Group due to the analysis presented in this paper, *RSAES-PKCS1-v1_5* had its support removed from the W3C Web Cryptography specification. Also, errors that could lead to attacks on AES-CBC wrapped keys, such as *DataError*, were removed from the spec where necessary and replaced with *OperationError* that could not distinguish between a key and padding operation. This should be considered a good example of a standards-based Working Group working well with knowledge from the cryptographic community.

5.1 AES-CBC Wrapped Keys

It is worth noting that despite the API's resistance to padding attacks against AES-CBC wrapped keys, this vulnerability could easily emerge through implementation errors or misuse of the API. To guard against implementation errors, we recommend the following checks:

- All errors caused by improper padding or incorrect key length/formatting are indistinguishable. (Padding errors will be returned from a different subroutine than the other errors and be discovered first, so any information about the *source* of the error is potentially a distinguishing factor.)
- Lengths of unwrapped keys are verified to match one of the predefined key lengths.
- All bytes of padding are checked for conformance.

Of these three recommendations, the first was accepted in to the specification. Additionally, the specific key lengths reduce the search space of a brute force attack against 192 and 256 bit keys. Unwrapping a 256 bit key as if it was 192 bits requires guessing only the 64 bits that need to be (wrongly) interpreted as padding for unwrapping to be successful. Thus the problem is reduced to finding a 192 bit key. These, in turn, require guessing another 64 bits in order to be unwrapped as if they were 128 bit keys. From there, the problem is equivalent finding a 128 bit key. Thus, brute forcing 192 and 256 bit keys takes at most $2^{128} + 2^{64}$ and $2^{128} + 2^{65}$ guesses respectively, which is less than the traditional brute force attack. Lastly, it should be mentioned that if the attacker is given an oracle that uses the *decrypt* operation instead of the *unwrap* operation with the same key used for wrapping, a standard padding attack may be able to recover wrapped keys.

5.2 High-Level API Recommendations

Although the API does not provide "safe" defaults, the IRTF CFRG (CryptoForum Research Group) created a document to track known security flaws, attacks, and the status of formal security proofs for each algorithm in the API.[17] From

[17] https://www.w3.org/2012/webcrypto/draft-irtf-cfrg-webcrypto-algorithms-01.html.

our analysis, it is quite clear what the recommend modes should be in general for a developer-friendly "high-level" API that also automatically took care of IV vector initialization and other parameters. For RSA-based algorithms, *RSA-PSS* should be used for signing and verification while *RSA-OAEP* should be used for encryption and decryption. It is likely that Curve 25519 support should be added. Standardised by NIST, *AES-GCM* is gaining traction in standards such as IPsec, MACSec, P1619.1, and TLS [35]. Regarding *DH*, more protocols are now favoring *ECDH* as attacks against "weak" standard Diffie-Hellman groups are not as powerful against elliptic curves due to a loss of a clear precomputation-based advantage [1]. HMAC has well-studied security proofs, even if the underlying hash function is not (weak) collision resistant [7]. In terms of hashing functions, of course *SHA-2* is to be preferred due to the amount of increased feasibility of practical methods of obtaining collisions for SHA-1.[18] As regard key size, in-line with NIST and ENISA [37], larger key sizes should be preferred such as RSA keys of at least 2048 bits and 256 bits for symmetric keys and elliptic curve cryptography.

6 Conclusions

6.1 Fixing the Web Cryptography API

In summary, the Web Cryptography API had three attacks, of which only one still stands. The attack that is still present is that the usages of keys are not preserved upon export that can be exploited in numerous ways to reveal not only wrapped secret key material sent from between the client and server but also disrupt authenticated key exchange. A number of simple mitigations would prevent this attack. The most general solution would be to prevent usages from being changed, but this binds key usages to a key throughout its lifespan. A more limited mitigation that would address only the unique case of wrapping would be to have key wrapping require that the properties of a wrapped and then unwrapped key be preserved, and not require the export of the wrapped key before wrapping. Wrapping could be done outside the general Javascript environment and only the wrapped key material exposed. One way to implement this option would be to inherit the property of being unextractable from the wrapping key to the wrapped key by default. Another more restrictive option would be to prevent wrapping and unwrapping. Earlier errors involving padding attacks being made possible due to error types were corrected, and the $RSAES-PKCS1-v1\_5$ algorithm was correctly removed from the specification due to the analysis presented in this paper. However, the API does not suffer from the fatal errors in its key management that can be detected via formal modeling, such as PKCS#11 [17] or the Yubikey [27].

In detail, the handling of key attributes in the API does not create a clear intuition about their actual effect as the *usages* may not always be supported, and so will confuse developers about key management across the boundary

[18] https://sites.google.com/site/itstheshappening/.

between client and server. For any key transported between either client and server or server and client, the *usages* array may be changed arbitrarily. In other words, the originating host has no control over the usages a key has once imported onto another host. Another limitation is that keys are either extractable or not, and must be extractable in order to be wrapped. As demonstrated, extractable keys are easily attacked and can be retrieved (including maliciously) from a client with a single API call. Although seemingly harmless insofar as we would assume a correctly designed Web application would only allow keys to be extractable on purpose, this produces counter-intuitive results when mixed with wrapping, as restricting keys to be wrapped to be extractable forces the aforementioned vulnerabilities. This wrapping attack was verified in all conformant Web Crypto implementations, including Chrome, Edge, and FireFox. Furthermore, it prevents WebCrypto for being used for use-cases such as those proposed by Netflix to ensure secure delivery of key material to clients. This attack also prevents users from sharing long-term private keys that are unknown to the server between sessions by virtue of wrapping and sending to the server and then downloading the wrapped keys into private local storage when a successful authentication is completed. This is a widely requested feature for those wanting some ability to authenticate without the server being able to easily impersonate a user by having access to all the user's secrets.

The lack of a long-term key storage model combined with a lack of persistent key usages may be detrimental to the usage of Web Cryptography. Without guidance, developers may make poor choices that do not meet expectations when storing key material, as the lifetime of these keys is tied to the execution environment. While this provides many positive security and privacy benefits, to retain a key for use in later sessions developers will need to make use of a persistent key storage service on the server using the previously described problematic key exporting and wrapping routines. As it would be expected then that key wrapping in order to send keys from the client to the server (and back again upon revisiting the page) will be used to preserve long-term keys, the key wrapping attacks mentioned earlier are particularly dangerous. One suggestion is that future versions of the specification should likely tie private keys and wrapping operations with special processing outside of the normal Javascript environment, or even more ambitiously try to use a trusted environment to secure keys and cryptographic operations. This may require some kind of tie between hardware tokens for keys and their operations. Recently, the W3C has been exploring adding hardware token access to the Web Cryptography API in their "Web Cryptography v.Next" workshop, and so the next version of the API may support both secure multi-session key storage and cryptographic operations on those keys via some form of a trusted execution environment[19] as well as access via

[19] Such as ARM TrustZone.

next-generation authentication APIs such as FIDO[20] to origin-bound platform-held keys via call-response requests that do not reveal the secret key material.[21]

Standards to assure the end user of the integrity of Javascript code would prevent many of these attacks. Only recently has the W3C begun to develop standards to secure Javascript code, and these tend to be quite simple such as the Sub-resource Integrity W3C standard that allows the hash of Javascript to be checked before running [15] or Content Security Policy [6] that restricts the domain of Javascript being run. In detail, Sub-resource Integrity requires Javascript linked or imported as a script to match a particular hash before execution and so could prevent some of these cross-scripting attacks or where a third-party library has been exploited in order to gain access to the origin. There does not yet exist for Javascript a way to securely install code, such as has been done via signed code in Linux-based operating systems, much less the more comprehensive necessary precautions taken into account by The Update Framework.[22] While signed Javascript may seem difficult, many other systems such as native applications have moved to such a model and so it should not be surprising if the Web itself may need to adopt signed code. In fact, the hashes of popular Javascript code could even be imagined to be stored in a Merkle-tree based append-only log such as those being designed in Certificate Transparency [28]. Also, there does not exist a standard way to defend the entry in cleartext of data in locally-running Javascript from the server.

These kinds of attacks could also be countered by creating higher-level libraries that make it easier to use the Web Cryptography API and avoid having developers make decisions of key usages and key exporting. 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 and for attacks such as those detailed in this paper.

6.2 Next Steps for Standards Research

More formal research is needed on the larger framework of the Web Cryptography API and the Web security model, with a focus on the possible interactions between Web Cryptography and other APIs that are part of HTML5. Ideally, 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 as most applications will use multiple APIs with possibly contradictory security policies. 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 when the WebCrypto

[20] http://www.fidoalliance.org.

[21] For details of the W3C Web Cryptography v.Next workshop that dealt with hardware tokens, FIDO, and trusted execution environments, see http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/.

[22] http://theupdateframework.com/.

API is used in combination with other APIs that allow low-level access to a browser's localstorage.

The process of formal modeling would be helpful if integrated into the standardization process to understand the security properties of APIs and their complex interactions with other APIs. One approach would be to include it at the early stages of the design of the standard. If it were, it could both correct early flaws, but would require considerable investment in updating the model. Another option would be do the formal model as part of the security review, although such a security review is currently optional at the W3C. Another option would be to include the formal modeling as part of the test-suite necessary to reach standardiation, where the test-suite must demonstrate security properties. One possible incentive structure is that just as currently W3C specifications require conformance testing via a test-suite to be done manually, the automatic generation of a test-suite using formal methods would both save the developers time and lead to a more thorough test-suite. The formally-generated test-suite could then be tested against real-world implementations in order to prove interoperability and conformance. The use of formal methods for testing is currently under development for the new Web Authentication API (formerly the "FIDO 2.0" API) that attempts to supplement passwords with one-factor cryptographic authentication.[23] In general, we hope that formal analysis of Web APIs will lead to a more secure Web that is better understood and easier to use for developers.

References

1. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., et al.: Imperfect forward secrecy: how Diffie-Hellman fails in practice. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 5–17. ACM (2015)
2. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium, CSF 2010, pp. 290–304. IEEE Computer Society, Washington, DC, USA (2010)
3. Sleevi, R., Watson, M.: Web Cryptography API. Candidate recommendation, IETF (2014). http://www.w3.org/TR/WebCryptoAPI/
4. Bansal, C., Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Keys to the cloud: formal analysis and concrete attacks on encrypted web storage. In: Basin, D., Mitchell, J.C. (eds.) POST 2013. LNCS, vol. 7796, pp. 126–146. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36830-1_7
5. Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., Tsay, J.-K.: Efficient padding oracle attacks on cryptographic hardware. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 608–625. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32009-5_36
6. Barth, A., Veditz, D., West, M.: Content Security Policy level 1.1. Working draft, W3C (2012). http://www.w3.org/TR/2014/WD-CSpp.11-20140211/

[23] https://www.w3.org/TR/webauthn.

7. Bellare, M.: New proofs for NMAC and HMAC: security without collision-resistance. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 602–619. Springer, Heidelberg (2006). doi:10.1007/11818175_36
8. Bellare, M., Rogaway, P.: The exact security of digital signatures-how to sign with RSA and Rabin. In: Maurer, U. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 399–416. Springer, Heidelberg (1996). doi:10.1007/3-540-68339-9_34
9. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., Zinzindohoue, J.K.: A messy state of the union: taming the composite state machines of TLS. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 535–552. IEEE (2015)
10. Bhargavan, K., Lavaud, A.D., Fournet, C., Pironti, A., Strub, P.-Y.: Triple handshakes and cookie cutters: breaking and fixing authentication over TLS. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 98–113. IEEE (2014)
11. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, CSFW 2001, pp. 82–96. IEEE Computer Society, Washington, DC, USA (2001)
12. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998). doi:10.1007/BFb0055716
13. Bond, M., Anderson, R.: API-level attacks on embedded systems. Computer **34**(10), 67–75 (2001)
14. Boneh, D., Shparlinski, I.E.: On the unpredictability of bits of the elliptic curve Diffie-Hellman scheme. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 201–212. Springer, Heidelberg (2001). doi:10.1007/3-540-44647-8_12
15. Braun, F., Akhawe, D., Weinberger, J., West, M.: Subresource Integrity. Working draft, W3C (2014). http://www.w3.org/TR/SRI/
16. Cremers, C.J.F.: The Scyther tool: verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_38
17. Delaune, S., Kremer, S., Steel, G.: Formal security analysis of PKCS#11 and proprietary extensions. J. Comput. Secur. **18**(6), 1211–1245 (2010)
18. Dennis, G., Chang, F.S.-H., Jackson, D.: Modular verification of code with SAT. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, 17–20 July 2006, Portland, Maine, USA, pp. 109–120 (2006)
19. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**(2), 198–208 (1983)
20. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: 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, CCS 2012, pp. 38–49. ACM, New York (2012)
21. Halpin, H.: The W3C web cryptography API: motivation and overview. In: Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion, WWW Companion 2014, pp. 959–964, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee (2014)
22. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)

23. Jager, T., Schinzel, S., Somorovsky, J.: Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML encryption. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 752–769. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33167-1_43
24. Kaliski, B.: PKCS #7: Cryptographic Message Syntax. RSA Security Inc., v1.5. https://www.ietf.org/rfc/rfc2315.txt
25. Kaminsky, A., Kurdziel, M., Radziszowski, S.: An overview of cryptanalysis research for the advanced encryption standard. In: 2010 Military Communications Conference - MILCOM 2010 (2010)
26. Krawczyk, H.: Cryptographic extraction and key derivation: the HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14623-7_34
27. Künnemann, R., Steel, G.: YubiSecure? Formal security analysis results for the Yubikey and YubiHSM. In: Jøsang, A., Samarati, P., Petrocchi, M. (eds.) STM 2012. LNCS, vol. 7783, pp. 257–272. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38004-4_17
28. Laurie, B., Langley, A., Kasper, E.: RFC 6962 Certificate Transparency. Experimental, IETF (2013). https://tools.ietf.org/html/rfc6962
29. Mitchell, C.J.: Error Oracle attacks on CBC mode: is there a future for CBC mode encryption? In: Zhou, J., Lopez, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 244–258. Springer, Heidelberg (2005). doi:10.1007/11556992_18
30. Near, J.P., Jackson, D.: Derailer: interactive security analysis for web applications. In: Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 587–598. IEEE/ACM (2014)
31. Paterson, K.G., Yau, A.: Padding Oracle attacks on the ISO CBC mode encryption standard. In: Okamoto, T. (ed.) CT-RSA 2004. LNCS, vol. 2964, pp. 305–323. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24660-2_24
32. Perrin, T.: Web Cryptography API. Editor's draft, W3C (2014). http://github.com/trevp/curve25519_webcrypto
33. Rizzo, J.: Duong., T.: Practical padding Oracle attacks. In: Proceedings of the 4th USENIX Conference on Offensive Technologies, WOOT 2010, pp. 1–8. USENIX Association, Berkeley, CA, USA (2010)
34. Rogaway, P., Shrimpton, T.: A provable-security treatment of the key-wrap problem. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 373–390. Springer, Heidelberg (2006). doi:10.1007/11761679_23
35. Rogaway, P.: Evaluation of some blockcipher modes of operation. Technical report, University of California, Davis, Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan, February 2011
36. Schmidt, B., Sasse, R., Cremers, C., Basin, D.: Automated verification of group key agreement protocols. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 179–194. IEEE (2014)
37. Smart, N.P., Rijmen, V., Warinschi, B., Watson, G., Patterson, K., Stam, M.: Algorithms, key sizes and parameters report: 2014 recommendations. Technical report, November 2014. ENISA Report. Version 1.0
38. Stark, E., Hamburg, M., Boneh, D.: Symmetric cryptography in Javascript. In: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC 2009, pp. 373–381. IEEE Computer Society, Washington, DC, USA (2009)

39. Taly, A., Erlingsson, Ú., Mitchell, J.C., Miller, M.S., Nagra, J.: Automated analysis of security-critical Javascript APIs. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP 2011, pp. 363–378. IEEE Computer Society, Washington, DC, USA (2011)
40. Torlak, E., Taghdiri, M., Dennis, G., Near, J.P.: Applications and extensions of alloy: past, present and future. Math. Struct. Comput. Sci. **23**(4), 915–933 (2013)
41. Vaudenay, S.: Security flaws induced by CBC padding — applications to SSL, IPSEC, WTLS. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 534–545. Springer, Heidelberg (2002). doi:10.1007/3-540-46035-7_35
42. Wen, C.C., Dawson, E., González Nieto, J.M., Simpson, L.: A framework for security analysis of key derivation functions. In: Ryan, M.D., Smyth, B., Wang, G. (eds.) ISPEC 2012. LNCS, vol. 7232, pp. 199–216. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29101-2_14
43. Yao, F.F., Yin, Y.L.: Design and analysis of password-based key derivation functions. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 245–261. Springer, Heidelberg (2005). doi:10.1007/978-3-540-30574-3_17