A Formal Model for Checking Cryptographic API Usage in JavaScript

Duncan Mitchell[1] and Johannes Kinder[2]

[1] Department of Computer Science, Royal Holloway, University of London, UK
`duncan.mitchell.2015@rhul.ac.uk`
[2] Research Institute CODE, Bundeswehr University Munich, Germany
`johannes.kinder@unibw.de`

Abstract. Modern JavaScript implementations include APIs offering strong cryptography, but it is easy for non-expert developers to misuse them and introduce potentially critical security bugs. In this paper, we formalize a mechanism to rule out such bugs through runtime enforcement of cryptographic API specifications. In particular, we construct a dynamic variant of Security Annotations, which represent security properties of values via type-like information. We formalize Security Annotations within an existing JavaScript semantics and mechanize it to obtain a reference interpreter for JavaScript with embedded Security Annotations. We provide a specification for a fragment of the W3C WebCrypto standard and demonstrate how this specification can reveal security vulnerabilities in JavaScript code with the help of a case study. We define a notion of safety with respect to Security Annotations and extend this to security guarantees for individual programs.

1 Introduction

The standardization of cryptographic APIs in JavaScript through the W3C Web Cryptography API, *WebCrypto* [31], has made strong cryptography available to web developers. In theory, this allows non-experts to implement true end-to-end encryption of confidential data. However, mistakes are easily made when developers use cryptographic APIs. For example, the JavaScript snippet in Listing 1 generates secure keys and then encrypts and signs a message before sending it. Here, the developer made the mistake of appending a signature of the plaintext to the message, allowing an observer to identify retransmissions of the same message. Mistakes like this undermine the security of the overall system, even when the implementation of the cryptographic API itself is correct.

Such mistakes are not exclusive to JavaScript, but in fact common across languages [8,16,20]. Alas, JavaScript exacerbates the problem, due to its dynamic nature and unconventional semantics [22], which thwart traditional analysis techniques and offer plenty of opportunities to violate API specifications. Existing work on JavaScript focuses on the verification of protocol implementations through restriction to small subsets of the language [2,14]. There is currently

```
1   const c = window.crypto.subtle;
2   const hmac = async (msg) => {
3     const alg = { name: "HMAC", hash: {name: "SHA-256"} };
4     const key = await c.generateKey(alg, false, ["sign", "verify"]);
5     return await c.sign({ name: "HMAC" }, key, msg);
6   }
7   const enc = async (msg) => {
8     const iv = crypto.getRandomValues(new Uint8Array(12));
9     const alg = { name: "AES-GCM", iv: iv };
10    const key = c.generateKey({name: "AES-GCM",
11      length: 256}, false, ["encrypt", "decrypt"]);
12    return await { iv: iv, ct: c.encrypt(alg, key, msg) };
13  }
14  let msg = new TextEncoder().encode("my message");
15  send({ct: enc(msg), sig: hmac(msg)});
```

Listing 1: Application code: signing before encrypting.

little support to help non-expert developers avoid introducing critical security bugs into applications built on full JavaScript.

In this work, we introduce a mechanism which rules out misuse of trusted APIs in JavaScript code through runtime enforcement. We extend the concept of Security Annotations [19], type-like tags which represent security properties, such as whether a value is ciphertext or a cryptographic key. Security Annotations are orthogonal to the existing type system and composable to allow for the expression of multiple distinct security properties.

In particular, we make the following contributions:

– We formalize a runtime semantics for Security Annotations in JavaScript by extending an existing formal semantics for a core of JavaScript, S5 [22] (Section 4).
– We mechanize Security Annotations, building upon an existing implementation of S5. We extend this to a reference interpreter for JavaScript programs through extending the JavaScript-to-S5 desugaring relation (Section 5).
– We provide an annotated fragment of the WebCrypto API which defines safe usage of the API through Security Annotations. Developers can replace the WebCrypto API with this annotated copy, which allows our mechanism to report violations of the otherwise implicit API specifications. We demonstrate how this approach can be used to avoid common cryptographic pitfalls by detecting violations of security properties in a case study (Section 5.3).
– We provide safety guarantees for this Security Annotation mechanism, and extend these to describe resulting security guarantees for programs using the annotated WebCrypto fragment (Section 6).

$$
\begin{array}{ll}
\text{[S-Refl]} \quad \dfrac{}{S \prec: S} & \text{[S-Top]} \quad \dfrac{}{S \prec: \mathsf{Top}} \\[2ex]
\text{[S-Trans]} \quad \dfrac{S_1 \prec: S_2 \qquad S_2 \prec: S_3}{S_1 \prec: S_2} & \text{[S-Perm]} \quad \dfrac{S_1 * \ldots * S_n \text{ permutes } R_1 * \ldots * R_n}{S_1 * \ldots * S_n \prec: R_1 * \ldots * R_n} \\[2ex]
\text{[S-Width]} \quad \dfrac{}{*_{i=1}^{n+k} S_i \prec: *_{i=1}^{n} S_i} & \text{[S-Depth]} \quad \dfrac{\text{for each } i,\; S_i' \prec: S_i}{*_{i=1}^{n} S_i' \prec: *_{i=1}^{n} S_i}
\end{array}
$$

Fig. 1: Hierarchical Security Annotation judgments [19].

2 Background

We begin by introducing necessary background on Security Annotations (Section 2.1) and our underlying JavaScript semantics (Section 2.2).

2.1 Security Annotations

Security Annotations represent security properties valid on objects or values within a program [19]. For example, the return value of a trusted key generation API is a valid cryptographic key, so could carry an annotation `CryptKey`. Annotations are composable: if the value has also been generated as a cryptographically secure random value (CSRV), then it can be annotated through composition `CSRV * CryptKey`, via the commutative operator $*$. Security properties are hierarchical: for example, `PrivKey` is more specific than `CryptKey`. Security Annotations therefore have a notion of subannotation judgments, e.g., `PrivKey` $\prec:$ `CryptKey`. Combined with the composition operator, this yields a lattice of security annotations. We include the rules defining this lattice in Figure 1. These judgments follow those given in Mitchell et al. [19]; additional rules governing reflexivity, transitivity and permutations are included for completeness. In these rules, S_i are arbitrary Security Annotations and `Top` is the least specific Security Annotation, representing a lack of security properties.

Mitchell et al. [19] enforce Security Annotations statically within a small lambda calculus. The expression v **as** S adds S as an annotation to v, representing newly valid security properties. Similarly, v **drop** S discards S from v, using a `cut` operator to remove an annotation whilst ensuring super-properties remain valid. We cut the annotation S_2 from S_1 via the following definition: $\mathsf{cut}(S_1, S_2)$ is the annotation R with (i) $S_1 \prec: R$ and (ii) $R \not\prec: S_2$ such that whenever R' also satisfies (i) and (ii) in place of R, then $R \prec: R'$ and $R' \not\prec: R$ [19].

2.2 S5: A Semantics for JavaScript

S5 [22] is a lambda calculus-like language which reflects the semantics of the strict mode of EcmaScript 5.1 (ES5). S5 is accompanied by a desugaring function, which takes native JavaScript source programs and translates them to S5 programs. S5

$$
\begin{array}{ll}
\text{[E-Compat]} \quad \dfrac{e \Longrightarrow e'}{\sigma\Theta; E{<}e{>} \to \sigma\Theta; E{<}e'{>}} &
\text{[E-EnvStore]} \quad \dfrac{\sigma; e \to^{\sigma} \sigma'; e'}{\sigma\Theta; E{<}e{>} \to \sigma'\Theta; E{<}e'{>}} \\[3ex]
\text{[E-Control]} \quad \dfrac{e \to^{\mathsf{e}} e'}{\sigma\Theta; E{<}e{>} \to \sigma\Theta; E{<}e'{>}} &
\text{[E-Objects]} \quad \dfrac{\Theta; e \to^{\Theta} \Theta'; e'}{\sigma\Theta; E{<}e{>} \to \sigma\Theta'; E{<}e'{>}}
\end{array}
$$

Fig. 2: The reduction relations for S5 [22].

itself is described via small-step semantics, incorporating ES5 features such as getters, setters and eval. The language is not a complete reference implementation for the entire standard but is tested against the official ES5 test suite.

Terms in S5 are 3-tuples comprised of an expression, e, a store σ (mapping locations to values) and an object store Θ (mapping references to object literals); the evaluation context is denoted E. The reduction relation $\to$ is split into four parts dependent on which portions of the term are manipulated; their definitions are given in Figure 2. For ease of reference, S5's syntax is given in Appendix A; full details of S5 are contained in the work of Politz et al [22].

3 Overview

We present an overview of our approach. First, we discuss how Security Annotations express properties of cryptography APIs (Section 3.1). We describe, through example, how such properties are enforced without changes to client code (Section 3.2).

3.1 Annotating APIs with Security Annotations

We provide a thin layer of JavaScript code (a *shim*) which adds pre- and postconditions to WebCrypto APIs. Listing 2 gives an example of such a shim for the `encrypt` API, which encodes the runtime specification for the API. This shim is included directly into application code via `require`; line 15 redefines WebCrypto's `encrypt`. This is the only addition an application developer need make to their codebase; the propagation of these annotations is governed by the mechanisms formalized in Section 4. Security Annotations on arguments are checked against the API's preconditions, made explicit through annotation guards. Security Annotations are then attached to return values of functions when these API calls contain specific postconditions.

Lines 1-3 define the annotation lattice for this API; this implicitly contains `Top`, which represents a lack of security properties. The syntax `SecAnn` $S_1 * \ldots * S_n$ defines orthogonal annotations $S_1, \ldots, S_n$. Lines 2-3 use the syntax `SecAnn` $S_1 * \ldots * S_n$ `Extends` S, to define new annotations $S_1, \ldots, S_n$ with $S_i \prec: S$ for each i.

Security Annotations are enforced at function boundaries via `arg` $: S$, which ensures `arg` meets the annotation guard S. For example, on line 8, we enforce that if the symmetric encryption algorithm AES is selected, then the `key` argument

```
1    SecAnn <CSRV * Message * CryptKey>;
2    SecAnn <PrivKey * PubKey * SymKey> Extends <CryptKey>;
3    SecAnn <Plaintext * Ciphertext> Extends <Message>;
4
5    window.oldCrypto = window.crypto;
6    const encShim = async function(alg :S ["iv", <CSRV>], key, data) {
7      if (/AES/.test(alg.name)) {
8        (function(arg : <SymKey>) {})(key);
9      } else if (/RSA/.test(alg.name)) {
10       (function(arg : <PubKey>) {})(key);
11     } else { throw FailedSecurityCheck; }
12     var res = await window.oldCrypto.subtle.encrypt(alg, key, data);
13     return (cpAnn(data, res) drop <Plaintext>) as <Ciphertext>;
14   }
15   Object.defineProperty(window.crypto.subtle, "encrypt", {value: encShim});
```

Listing 2: An annotated shim of WebCrypto's `encrypt` API.

to the function is annotated with `SymKey`. On line 6, `:E` checks that the specified object property meets its guard. In this case, we check that the initialization vector supplied as part of the `alg` object is a properly generated random value.

Postconditions are attached as annotations to return values. Encryption is performed by the original API (line 12); annotations representing newly valid security properties are then attached (line 13). First, **cpAnn** attaches all annotations from the `data` argument to `res`: security properties of the data are not invalidated as a result of encryption. The advantage of **cpAnn** is that we do not need to know the precise annotations of `data`. The exception is that if `data` was annotated with `Plaintext` (e.g., if it had been previously decrypted), we discard this annotation via the **drop** operator. Finally, we attach `Ciphertext` to the return value via the **as** operator.

3.2 Transparent Property Enforcement

The application in Listing 1 sends an encrypted, signed message across a network via the `send` method (line 15). The application developer uses WebCrypto in order to encrypt and sign this message; without our drop-in WebCrypto shim this application will execute and the developer will be unaware of a security flaw. Although individual wrapper functions for signing and encryption are correct, there is a logical error that causes a security bug. In particular, a signature is generated of the plaintext and sent alongside the corresponding ciphertext. This undermines the security of the application: attacks against this signature can reveal details about the underlying plaintext.

At runtime, the application builds the object to send: the `ct` property is constructed by calling the developer's `enc` function. This function is correct: the array stored in `iv` is annotated with `CSRV`, the postcondition of WebCrypto's

$$
\begin{array}{rl}
a := & \textit{Atomic Annotations} \\
S := & a \mid S * S \mid \mathsf{Top} \\
w := & \mathsf{str} \mid \mathsf{num} \mid \mathbf{true} \mid \mathbf{false} \\
w' := & \mathbf{null} \mid \mathbf{undefined} \mid \mathsf{func}(x : S, \ldots)\{e\} \\
r := & \textit{object references} \\
l := & \textit{locations} \\
v := & w\mathord{<}S\mathord{>} \mid r \mid w' \\
e := & \ldots \mid e \ \mathbf{as} \ S \mid e \ \mathbf{drop} \ S \mid \mathsf{cpAnn}(e, e) \\
\cdots & \\
\theta' := & \{[av] \ \mathsf{str} : pv, \ldots\} \\
\theta := & \theta'\mathord{<}S\mathord{>} \\
\sigma := & \cdot \mid \sigma, l : v \\
\Theta := & \cdot \mid \Theta, r : \theta \\
\cdots & \\
E' := & \ldots \mid E' \ \mathbf{as} \ S \mid E' \ \mathbf{drop} \ S \mid \mathsf{cpAnn}(E', e) \mid \mathsf{cpAnn}(v, E')
\end{array}
$$

Fig. 3: Syntax modifications to add Security Annotations to S5.

getRandomValues. The generateKey API, when the algorithm is symmetric (i.e., in the case of both HMAC and AES), returns a valid symmetric key (annotated with SymKey). The call to WebCrypto's encrypt succeeds since each argument satisfies the specification (Listing 2). Next, the developer calls hmac with argument the unencrypted msg. Similarly, the key is correctly generated via the API. However, there is an implicit precondition of sign—data to be signed must be ciphertext to avoid common attacks against the signature revealing information about it. By using our drop-in shim for WebCrypto, this bug is detected: since msg is not annotated with Ciphertext an error is thrown on entry to sign, reporting the violation to the developer.

4 Security Annotations for S5

We formalize Security Annotations within S5 [22], starting with modifications to the syntax (Section 4.1). We describe mechanisms for manipulating annotations (Section 4.2), runtime enforcement (Section 4.3), and their effect on the rest of S5 (Section 4.4).

4.1 Syntax

The additions and modifications to the syntax of S5 (given in Appendix A) to incorporate Security Annotations are contained in Figure 3. We introduce atomic

annotations a, which represent a single security property, and general annotations S, which are either Top, the least specific annotation, an atomic annotation, or the composition of two annotations, given by $*$. Annotations are only attached to certain prevalues w. Prevalues which should not be annotated are given by w', values are then either references r, $w\langle S\rangle$ or w', where $w\langle S\rangle$ is syntactic sugar for the pair of an annotatable value w along with its corresponding Security Annotation S. An additional modification to the syntax reflects the addition of annotations to objects: we consider pre-objects, θ', which form objects when annotated with a Security Annotation S. We annotate objects directly as opposed to their references; properties within objects are annotated in the same manner as values. When an object is modified, previously valid security properties on the object are no longer guaranteed: modifying an object field should alter the annotations associated to the field, and also the annotations of the overall object.

Additional expressions, e, based on manipulating security annotations, cover the `as`, `drop` and `cpAnn` constructs. We add evaluation contexts, E', to cover these cases, where these are built in the same manner as in S5 (see Appendix A). Finally, enforcement of Security Annotations is added to functions via the form $\mathrm{func}(x : S, \ldots)$; this does not require modification of the evaluation contexts.

4.2 Coercing Security Annotations

The evaluation judgments for coercion of annotations on values and objects are given in Figure 4, distinguished by case analysis on values. The expression v `as` S upcasts v to a more specific annotation, achieved by composing the previously valid annotation with S. Dependent on whether we treat $w\langle S\rangle$ (in [E-AsW]), or a reference r ([E-AsR]), we make use of distinct reduction relations (Figure 2). In the former case, [E-Compat] is used to govern the evaluation. In the latter, [E-Objects] is used to modify the object's annotation in the object store. Finally, we throw an error whenever a function, `null` or `undefined` is passed to one of these expressions treating coercion of annotations (e.g., [E-AsW']). The case analysis for `drop` are similar; v `drop` S downcasts v to a less specific annotation. This is accomplished via the cut operator (Section 2.1) to prune the S from the annotation of v. Listing 2 illustrates the use of `cpAnn` to ensure properties of data are still valid after encryption by copying annotations from one value (or object) to another. As with `as`, the addition of newly valid annotations does not render previous annotations invalid, so composition unifies them; the evaluation rules are therefore similar in structure.

4.3 Checking Security Annotations

Figure 5 codifies the enforcement of Security Annotations at function boundaries. [E-App] governs the case when arguments meet their annotation-guards and the function is evaluated. This rule inspects the object store (to look up object annotations when arguments are references) and modifies the variable store (to bind arguments to the corresponding variables); we therefore use the standard reduction relation rather than the split components (Figure 2). To reflect the
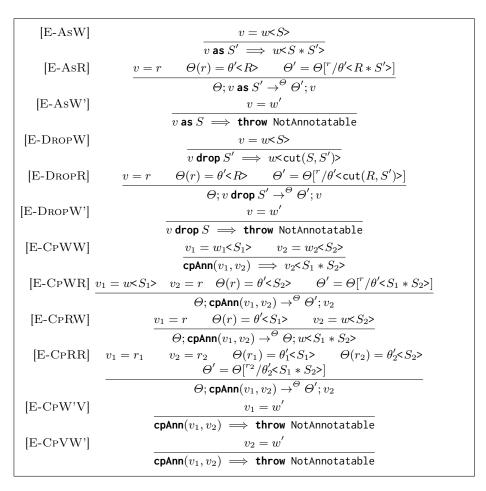
$$[\text{E-AsW}] \quad \frac{v = w\texttt{<}S\texttt{>}}{v \text{ as } S' \implies w\texttt{<}S * S'\texttt{>}}$$

$$[\text{E-AsR}] \quad \frac{v = r \qquad \Theta(r) = \theta'\texttt{<}R\texttt{>} \qquad \Theta' = \Theta[^r/\theta'\texttt{<}R * S'\texttt{>}]}{\Theta; v \text{ as } S' \to^\Theta \Theta'; v}$$

$$[\text{E-AsW'}] \quad \frac{v = w'}{v \text{ as } S \implies \texttt{throw NotAnnotatable}}$$

$$[\text{E-DropW}] \quad \frac{v = w\texttt{<}S\texttt{>}}{v \text{ drop } S' \implies w\texttt{<}\texttt{cut}(S, S')\texttt{>}}$$

$$[\text{E-DropR}] \quad \frac{v = r \qquad \Theta(r) = \theta'\texttt{<}R\texttt{>} \qquad \Theta' = \Theta[^r/\theta'\texttt{<}\texttt{cut}(R, S')\texttt{>}]}{\Theta; v \text{ drop } S' \to^\Theta \Theta'; v}$$

$$[\text{E-DropW'}] \quad \frac{v = w'}{v \text{ drop } S \implies \texttt{throw NotAnnotatable}}$$

$$[\text{E-CpWW}] \quad \frac{v_1 = w_1\texttt{<}S_1\texttt{>} \qquad v_2 = w_2\texttt{<}S_2\texttt{>}}{\texttt{cpAnn}(v_1, v_2) \implies v_2\texttt{<}S_1 * S_2\texttt{>}}$$

$$[\text{E-CpWR}] \quad \frac{v_1 = w\texttt{<}S_1\texttt{>} \quad v_2 = r \quad \Theta(r) = \theta'\texttt{<}S_2\texttt{>} \qquad \Theta' = \Theta[^r/\theta'\texttt{<}S_1 * S_2\texttt{>}]}{\Theta; \texttt{cpAnn}(v_1, v_2) \to^\Theta \Theta'; v_2}$$

$$[\text{E-CpRW}] \quad \frac{v_1 = r \qquad \Theta(r) = \theta'\texttt{<}S_1\texttt{>} \qquad v_2 = w\texttt{<}S_2\texttt{>}}{\Theta; \texttt{cpAnn}(v_1, v_2) \to^\Theta \Theta; w\texttt{<}S_1 * S_2\texttt{>}}$$

$$[\text{E-CpRR}] \quad \frac{v_1 = r_1 \quad v_2 = r_2 \quad \Theta(r_1) = \theta'_1\texttt{<}S_1\texttt{>} \qquad \Theta(r_2) = \theta'_2\texttt{<}S_2\texttt{>} \qquad \Theta' = \Theta[^{r_2}/\theta'_2\texttt{<}S_1 * S_2\texttt{>}]}{\Theta; \texttt{cpAnn}(v_1, v_2) \to^\Theta \Theta'; v_2}$$

$$[\text{E-CpW'V}] \quad \frac{v_1 = w'}{\texttt{cpAnn}(v_1, v_2) \implies \texttt{throw NotAnnotatable}}$$

$$[\text{E-CpVW'}] \quad \frac{v_2 = w'}{\texttt{cpAnn}(v_1, v_2) \implies \texttt{throw NotAnnotatable}}$$

Fig. 4: Judgments for coercing annotations: **as**, **drop** and **cpAnn**.

hierarchy of the annotation lattice, this rule bakes in subsumption, e.g., enforcement of `CryptKey` would accept the more specific `PrivKey`. A common JavaScript paradigm is for non-annotatable values, e.g., functions, to be passed as arguments; we insist the guard for such arguments is `Top`, i.e., no security precondition. For any annotatable values, $w\texttt{<}S\texttt{>}$, we insist S satisfies the guard S'. For references r, we look up the corresponding object and insist the annotation meets the guard. Direct checking of object properties and the **this** argument is achieved via source-to-source rewritings, described in Section 5.2. [E-AppFail] describes what happens when annotation-checking fails, i.e., whenever an argument carries a less precise annotation than its guard. `FailedSecurityCheck` is thrown to report the potential security vulnerability to the user, rather than simply halting evaluation.
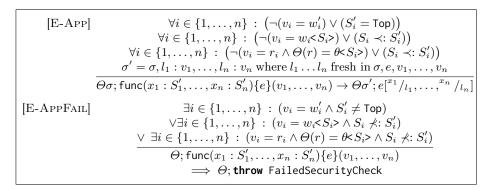
$$[\text{E-App}] \quad \begin{array}{c} \forall i \in \{1, \ldots, n\} \; : \; \big(\neg(v_i = w'_i) \vee (S'_i = \mathsf{Top})\big) \\ \forall i \in \{1, \ldots, n\} \; : \; \big(\neg(v_i = w_i \langle S_i \rangle) \vee (S_i \prec: S'_i)\big) \\ \forall i \in \{1, \ldots, n\} \; : \; \big(\neg(v_i = r_i \wedge \Theta(r) = \theta \langle S_i \rangle) \vee (S_i \prec: S'_i)\big) \\ \sigma' = \sigma, l_1 : v_1, \ldots, l_n : v_n \text{ where } l_1 \ldots l_n \text{ fresh in } \sigma, e, v_1, \ldots, v_n \\ \hline \Theta\sigma; \mathsf{func}(x_1 : S'_1, \ldots, x_n : S'_n)\{e\}(v_1, \ldots, v_n) \to \Theta\sigma'; e[{}^{x_1}/l_1, \ldots, {}^{x_n}/l_n] \end{array}$$

$$[\text{E-AppFail}] \quad \begin{array}{c} \exists i \in \{1, \ldots, n\} \; : \; (v_i = w'_i \wedge S'_i \neq \mathsf{Top}) \\ \vee \exists i \in \{1, \ldots, n\} \; : \; (v_i = w_i \langle S_i \rangle \wedge S_i \not\prec: S'_i) \\ \vee \; \exists i \in \{1, \ldots, n\} \; : \; (v_i = r_i \wedge \Theta(r) = \theta \langle S_i \rangle \wedge S_i \not\prec: S'_i) \\ \hline \Theta; \mathsf{func}(x_1 : S'_1, \ldots, x_n : S'_n)\{e\}(v_1, \ldots, v_n) \\ \implies \Theta; \textbf{throw } \mathtt{FailedSecurityCheck} \end{array}$$

Fig. 5: Function application with Security Annotation enforcement.

4.4 Completing S5 with Security Annotations

The rest of S5 remains largely unchanged. After object fields are manipulated, there is no guarantee the object annotation remains valid. For example, modifying the `keyUsages` field of a key object returned from the `generateKey` API may undermine the security of any future operation involving the key. Any previously valid security properties on the object can no longer be guaranteed; `Top` is therefore associated as the object's annotation. Figure 6 includes judgments for field manipulation, including adding fields which do not exist and writable 'shadow' fields. These semantics are transparent to annotations to allow prevalues to govern control flow, e.g., the `configurable` property must be **true** in [E-DeleteField].

5 Security Annotations for JavaScript

We describe the mechanization of this model (Section 5.1) and a desugaring relation which allows the execution of JavaScript with Security Annotations[3]. We discuss the annotation checking of object internals (Section 5.2) and demonstrate its operation on a case study (Section 5.3).

5.1 Implementing Security Annotations in S5

We mechanize Security Annotations on top of the existing reference implementation of S5 [22]. Alongside object and variable stores, we maintain a third *annotation store*, the lattice of valid annotations in the program. Security Annotations are declared via the **SecAnn** and **Extends** expressions described in Section 3.1. These expressions modify the annotation store to reflect additions to the lattice and evaluate to **undefined**. Using an annotation prior to declaration results in an exception. The lattice is also inspected in function application (Figure 5)

---

[3] An accompanying implementation is available at: https://github.com/duncan-mitchell/SecAnnRefInterpreter.
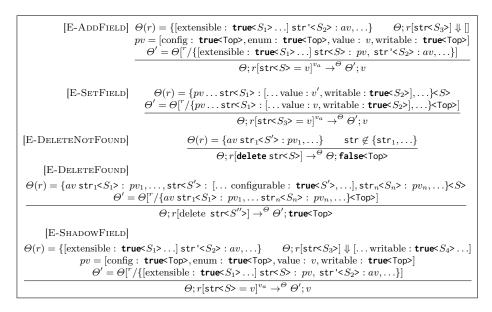
$$[\text{E-AddField}] \quad \Theta(r) = \{[\text{extensible} : \textbf{true}<S_1>\ldots]\, \text{str}'<S_2> : av,\ldots\} \qquad \Theta; r[\text{str}<S_3>] \Downarrow []$$
$$pv = [\text{config} : \textbf{true}<\text{Top}>, \text{enum} : \textbf{true}<\text{Top}>, \text{value} : v, \text{writable} : \textbf{true}<\text{Top}>]$$
$$\Theta' = \Theta[^r/\{[\text{extensible} : \textbf{true}<S_1>\ldots]\, \text{str}<S> : pv,\, \text{str}'<S_2> : av,\ldots\}]$$
$$\overline{\Theta; r[\text{str}<S> = v]^{v_a} \to^\Theta \Theta'; v}$$

$$[\text{E-SetField}] \quad \Theta(r) = \{pv \ldots \text{str}<S_1> : [\ldots \text{value} : v', \text{writable} : \textbf{true}<S_2>],\ldots\}<S>$$
$$\Theta' = \Theta[^r/\{pv \ldots \text{str}<S_1> : [\ldots \text{value} : v, \text{writable} : \textbf{true}<S_2>],\ldots\}<\text{Top}>]$$
$$\overline{\Theta; r[\text{str}<S_3> = v]^{v_a} \to^\Theta \Theta'; v}$$

$$[\text{E-DeleteNotFound}] \quad \frac{\Theta(r) = \{av\, \text{str}_1<S'> : pv_1,\ldots\} \qquad \text{str} \notin \{\text{str}_1,\ldots\}}{\Theta; r[\textbf{delete}\ \text{str}<S>] \to^\Theta \Theta; \textbf{false}<\text{Top}>}$$

$$[\text{E-DeleteFound}]$$
$$\Theta(r) = \{av\, \text{str}_1<S_1> : pv_1,\ldots, \text{str}<S'> : [\ldots \text{configurable} : \textbf{true}<S'>,\ldots], \text{str}_n<S_n> : pv_n,\ldots\}<S>$$
$$\Theta' = \Theta[^r/\{av\, \text{str}_1<S_1> : pv_1,\ldots \text{str}_n<S_n> : pv_n,\ldots\}<\text{Top}>]$$
$$\overline{\Theta; r[\text{delete}\ \text{str}<S''>] \to^\Theta \Theta'; \textbf{true}<\text{Top}>}$$

$$[\text{E-ShadowField}]$$
$$\Theta(r) = \{[\text{extensible} : \textbf{true}<S_1>\ldots]\, \text{str}'<S_2> : av,\ldots\} \qquad \Theta; r[\text{str}<S_3>] \Downarrow [\ldots \text{writable} : \textbf{true}<S_4>\ldots]$$
$$pv = [\text{config} : \textbf{true}<\text{Top}>, \text{enum} : \textbf{true}<\text{Top}>, \text{value} : v, \text{writable} : \textbf{true}<\text{Top}>]$$
$$\Theta' = \Theta[^r/\{[\text{extensible} : \textbf{true}<S_1>\ldots]\, \text{str}<S> : pv,\, \text{str}'<S_2> : av,\ldots\}]$$
$$\overline{\Theta; r[\text{str}<S> = v]^{v_a} \to^\Theta \Theta'; v}$$

Fig. 6: Judgments for setting, deleting and adding fields.

to compare annotations with respect to subsumption. Section 4 describes functions in which each argument is checked against some annotation guard. In implementation, we retain enforcement-free functions and do not insist every argument has an annotation-guard. This allows reuse of existing ES5 environment implementations described in the work of Politz et al. [22].

5.2   A Reference Interpreter for Security Annotations in JavaScript

We execute JavaScript code with Security Annotations by extending the JavaScript-to-S5 desugaring relation. We extend the syntax of JavaScript by adding Security Annotations and function guards, as well as the expressions **as**, **drop**, **cpAnn**, **SecAnn** and **SecAnn Extends**. Our desugaring rewrites these expressions into their S5 equivalents, which are then executed in the reference interpreter.

Checking object properties. Listing 2 demonstrates the need for checking properties of objects. We achieve this via source-to-source rewritings at the JavaScript level; these are simplified by an assert function

```
let assert = function(arg, ann){ (function(x : ann) {})(arg); }.
```

There are three possible cases; first, obj :S [prop, S] checks obj[prop] meets S. We check the specified property exists, and insist it satisfies the guard S:

```
if (typeof obj == 'object' &&
  Object.getOwnPropertyNames(obj).indexOf(prop) >= 0) {
    assert(x[prop], S);
} else { throw 'FailedSecurityCheck'; }
```

Second, `obj :A` S checks all properties meet the guard S; to achieve this we iterate over all object properties:

```
if (typeof obj == 'object') {
  let props = Object.getOwnPropertyNames(obj);
  for (let iter = 0; iter < props.length; iter++) {
    assert(obj[props[iter]], S);
  }
} else { throw 'FailedSecurityCheck'; }.
```

Finally, `obj :E [`N, S`]` checks at least N properties satisfy S. As before, we iterate over object properties, counting the number that meet the guard:

```
if (typeof obj == 'object') {
  let props = Object.getOwnPropertyNames(x), successes = 0;
  for (let iter = 0; iter < props.length; iter++) {
    try { assert(x[props[iter]], S); successes++; } catch (e) { };
  }
  if (successes < N) { throw 'FailedSecurityCheck'; }
} else { throw 'FailedSecurityCheck'; }
```

Checking `this`. Functions have an implicit `this` argument, the context object in which the current code is executing. In the manner of checking object properties, we check this via the syntax `function(:` S, `...) { body(); }` which is rewritten to `function(...){ assert(this,` S`); body(); }`.

5.3   Using the Reference Interpreter

We provide a reference implementation of Security Annotations for the correctness of future implementations in native JavaScript. Our interpreter translates a subset of Node.js programs into S5 programs; we demonstrate the scope of this reference interpreter by describing the modifications to programs necessary for execution. We outline how we envisage Security Annotations being used by developers to detect security vulnerabilities through case study within our interpreter.

A client-server application. We implement a small chat application which takes as argument a confidential message a client wishes to transmit to a server[4]. The server and client negotiate a key exchange, and an encrypted copy of this message is sent to the server, which decrypts it. We omit authentication from this case study for simplicity of presentation. WebCrypto is not implemented in Node.js, so we construct a synchronous mock using the Node.js `crypto` module.

Execution in S5. Library mocks are necessary to execute the case study in S5. S5 does not support asynchronous code, so we construct a synchronous mock of the networking API, `net`. An extension to asynchronous code is possible in principle based on an existing formalization of JavaScript promises [18]. Second,

---

[4] The source code for this application is available alongside the reference interpreter.

cryptographic operations are mocked as stub functions returning objects of the same underlying structure. Finally, S5 programs do not take input, so we declare `process.argv` to simulate this.

Completing the WebCrypto shim. Listing 3 contains an annotated shim of a fragment of WebCrypto for use by developers. These method specifications follow the same structure as Listing 2. `getRandomValues` fills the supplied array with random values, so this array is annotated with `CSRV`. Despite the lack of a return, the annotation on this array persists because the annotation is attached directly to the object. `generateKey` constructs a key (or key pair) object for the supplied algorithm; postconditions of this method are differentiated by case analysis. `deriveKey` is used to compute a shared secret key from the other party's public key and the private key. The contract for `decrypt` is similar to `encrypt`; we do not enforce `Ciphertext` against `data`—or that the IV is randomly generated—to allow decryption of messages received across a network. `importKey` allows public keys received across a network to be formatted for use with other WebCrypto APIs. This API allows the upcasting of arbitrary data; however, without `importKey`, it would be impossible to use WebCrypto across a network.

A security property violation. When constructing the IV, the developer ensures that it can be encoded directly as an ASCII string. Despite correctly generating an IV of the same size as the cipher block size (calling `getRandomValues` on a `Uint8Array` of size 16), they reduce entropy of the IV by zeroing the top bit of each element of this array. This causes the IV to contain only 112 bits of entropy, less than the block size: a potential security flaw which does not visibly affect runtime behavior. To detect such bugs, a developer includes our WebCrypto shim. The IV is initially generated by a WebCrypto API call and annotated with `CSRV`; however, the manipulation of the array drops the annotation (per [E-SETFIELD] in Figure 6). Since the `iv` property of the `alg` object is not annotated with `CSRV`, the call to `encrypt` fails, `FailedSecurityCheck` is thrown and this security flaw is reported to the developer. When the loss of entropy is removed, no error is thrown; the security pre- and postconditions enforced in the shim are respected.

6 Properties of Security Annotations

We discuss safety guarantees for S5 programs with Security Annotations (Section 6.1) and extend this to security guarantees (Section 6.2). Finally, we apply this to prove security of our case study (Section 6.3). Throughout this section, we assume all programs discussed terminate.

6.1 Safety Guarantees

We adopt a relatively modest notion of safety: first, a program is safe if it does not evaluate to an exception as a result of a function argument failing to meet the annotation guard. Second, the program should not coerce the annotation of a non-annotatable value, e.g., **null as** `<CSRV>`. This gives us the definition:

```
1   SecAnn <CSRV * Message * CryptKey>;
2   SecAnn <PrivKey * PubKey * SymKey> Extends <CryptKey>;
3   SecAnn <Plaintext * Ciphertext> Extends <Message>;
4
5   window.oldCrypto = window.crypto;
6   let wc = window.oldCrypto.subtle;
7   const grvShim = function(arr) {
8     window.oldCrypto.getRandomValues(arr);
9     arr as <CSRV>;
10  };
11  const gkShim = async function(alg, extractable, keyUsages) {
12    let key = await wc.generateKey(alg, extractable, keyUsages);
13    if (/RSA|ECD/.test(alg.name)) {
14      key.privateKey = key.privateKey as <PrivKey * CSRV>;
15      key.publicKey = key.publicKey as <PubKey * CSRV>;
16    } else if(/AES|HMAC/.test(alg.name)) {
17      key as <SymKey * CSRV>;
18    } else { throw FailedSecurityCheck; }
19    return key;
20  };
21  const dkShim = async function(alg :S ["public", <PubKey>],
22      masterKey : <PrivKey>, derivedKeyAlg, extractable, keyUsages) {
23    let key = await wc.deriveKey(alg, masterKey, derivedKeyAlg, extractable,
24      keyUsages);
25    return (key as <SymKey>);
26  };
27  const decShim = async function(alg, key, data) {
28    if (/AES/.test(alg.name)) {
29      (function(arg : <SymKey>) {})(key);
30    } else if (/RSA/.test(alg.name)) {
31      (function(arg : <PrivKey>) {})(key);
32    } else { throw FailedSecurityCheck; }
33    var res = await wc.decrypt(alg, key, data);
34    return ((cpAnn(data, res) drop <Ciphertext>) as <Plaintext>);
35  };
36  const ikShim = async function(type, key, alg, extractable, keyUsages) {
37    let pubKey = await wc.importKey(type, key, alg, extractable, keyUsages);
38    return (pubKey as <PubKey>);
39  };
40  const wcShim = { generateKey: {value: gkShim}, deriveKey: {value: dkShim},
41    encrypt: {value: encShim}, decrypt: {value: decShim},
42    importKey: {value: ikShim}};
43  defineProperty(window.crypto, "subtle", { value: wcShim});
44  defineProperty(window.crypto, "getRandomValues", {value: grvShim});
```

Listing 3: An annotated shim for a fragment of the WebCrypto API.

Definition 1 (Annotation Safety)**.** An S5 program is *safe with respect to Security Annotations* (or, *annotation safe*) if the execution of the program does not result in either a `FailedSecurityCheck` or `NotAnnotatable` exception.

Although programs in S5 are deterministic, programs in JavaScript (or any meaningful language) are not: their execution depends on the DOM or user input. Suppose $\mathcal{P}$ is a program expecting input, we extend Definition 1 as follows:

Definition 2 (Annotation Safety for Programs with Input)**.** $\mathcal{P}$, is *annotation safe* if no execution of the program results in either a `FailedSecurityCheck` or `NotAnnotatable` exception.

Consider a family of S5 programs, Π, which are deterministic and simulate input by declaring a global variable `process.argv` assigned to an object containing N fields. For each field, f_i suppose there is an accompanying value v_i. For each v_i, we fix a base type and range over all possible prevalues (and **undefined**, which simulates a lack of input). If v_i is a reference to an object, we range over all possible objects θ. The resulting family of programs represents the space of possible executions for $\mathcal{P}$. We can therefore reformulate Definition 2:

Lemma 3. *Let $\mathcal{P}$ be an S5 program with input and Π the family of deterministic programs p describing all possible inputs for $\mathcal{P}$. Then $\mathcal{P}$ is annotation safe if and only if every program $p \in \Pi$ is annotation safe.*

Proof. By construction, each execution of $\mathcal{P}$ is considered as a separate deterministic program P so the result is immediate. $\square$

Since this family Π is very large, we formalize safety in terms of a subset of these programs. Let π be the set of all $p \in \Pi$ following exactly the same sequence of evaluation judgments. This set of S5 programs corresponds to a single control-flow path of $\mathcal{P}$: so if any p is annotation safe, so are all programs in π. Since the union of all (clearly disjoint) possible paths π is equal to the overall family of programs Π, we can obtain a simpler notion of safety for $\mathcal{P}$:

Theorem 4. *Let Π be the family of deterministic programs describing all possible inputs for $\mathcal{P}$. Consider all disjoint subsets $\pi \subseteq \Pi$ representing single control flow paths of $\mathcal{P}$, and for each, choose a single $p \in \pi$. Then $\mathcal{P}$ is annotation safe if and only if each p is annotation safe.*

Proof. Suppose first that $\mathcal{P}$ is annotation safe. Then by Lemma 3, we know every $P \in \Pi$ is annotation safe. Since each $\pi \subseteq \Pi$, each p must be annotation safe as required. For the other direction, suppose each p is annotation safe. Pick one such p, and the subset of Π to which it belongs, π. Let p' be some other program in π, and suppose that p' is not annotation safe. Then the execution of p' results in either a `FailedSecurityCheck` or `NotAnnotable` exception. This means that the final evaluation judgment applied in the evaluation of p' is either [E-APPFAIL], [E-ASW'], [E-DROPW'], [E-CPW'V] or [E-CPVW']. Since p and p' both belong to π, they follow the same sequence of evaluation judgments. But then p is not annotation safe, which is a contradiction. Thus each p in π is annotation safe,

and extending this across all disjoint subsets π of Π, each program in Π must be annotation safe. Applying Lemma 3 again, we are done. $\square$

This result says that if any set π is not safe, then some control-flow path in $\mathcal{P}$ violates the Security Annotation specification of the program, indicating a possible security vulnerability. This description of safety requires us to find these subsets π to obtain a guarantee. In practice, this is equivalent to enumerating all control flow paths of a program over all types of input values and objects, which makes our mechanism ideally suited for combination with feedback-directed fuzzing or dynamic symbolic execution [17].

6.2 Security Guarantees

Let L be a library and L' an annotated shim of this library; any security guarantees are conditional on the correctness of L, e.g., that WebCrypto itself is a correct implementation of cryptographic primitives. Let P be an S5 program which calls L, and suppose the developer of P in-lines this annotated shim in a program $P' = L'; P$. We assume that P does not contain any expressions which manipulate Security Annotations. We can make the following (overapproximate) claim, which states that the whenever P' is annotation safe, it respects the security properties enforced by the Security Annotation specifications of the methods in L'.

Lemma 5. *Suppose P' is annotation safe. Then the Security Annotation specifications described in L' are respected.*

Proof. Suppose a Security Annotation specification in L' is not respected. Then some function precondition fails, so the judgment [E-AppFail] is evaluated, contradicting our assumption that P' is annotation safe. Since P does not involve the manipulation of Security Annotations, any annotations must be the postconditions of an API call in L'; hence these specifications are respected. $\square$

Analogously to Section 6.1, we extend this result to programs with input:

Theorem 6. *Let $\mathcal{P}$ be a program with input and suppose $\mathcal{P}' = L'; \mathcal{P}$ is annotation safe. Then the Security Annotation specifications described in L' are respected.*

Proof. This is immediate from the combination of Theorem 4 and Lemma 5. $\square$

6.3 Security Guarantees in Practice

We use Theorem 6 to describe concrete security guarantees for the case study outlined in Section 5.3, which are conditional on the correctness of WebCrypto. Recall that after fixing the security vulnerability involving the ASCII-encoded IV, when a message supplied as argument, the program executes without error; if no message is provided the application simply reports this to the user and exits. Both control-flow paths of this program are annotation safe. Referring to the specifications described in our WebCrypto shim (Listing 3), there are two caveats to our claim; the first assumes the developer does not leak keying material and the second relates to the omission of authentication from the case study.

15

Theorem 7. *Suppose that: (i) neither the symmetric key nor either party's secret keys are leaked across the network, and (ii), an attacker impersonates neither party. Then encrypted messages sent by the client can only be read by the server.*

Proof. The application does not manipulate annotations; when executed with a non-annotated copy of the library the program is annotation safe. As described above, both control-flow paths of the program are annotation safe with our annotated library in-lined, we can directly apply Theorem 6. It remains to demonstrate the specification enforced by the annotation library. The encryption—via AES-CBC with a 128-bit key—is secure only when the symmetric key has been securely derived, and the IV is a block-sized CSRV (Listing 2). Our Web-Crypto specification enforces the CSRV portion of the contract directly: calling `getRandomValues` annotates the IV with `CSRV` (lines 7-9 of Listing 3), and this array is not subsequently modified, the annotation check on entry to `encrypt` passes.

Second, the symmetric key used for AES must be shared between the two parties secretly. The key is derived through an ECDH key exchange; both the server and client use `generateKey` (lines 11-20 of Listing 3) to compute a key pair. Public keys are exchanged, and validated it through `importKey` (lines 36-39). The client supplies their private key and the server's public key to `deriveKey` (lines 21-26). Neither key has been tampered with, so the client's key is annotated with `PrivKey` and the server's with `PubKey`. This satisfies the guard of `deriveKey`, and so the key for AES is computed, and annotated `SymKey`. The provenance of the secret key as derived from safe API calls can be confirmed, so the guard against the key in `encrypt` succeeds (line 10 of Listing 2). Therefore, only someone in possession of the private key corresponding to the server's public key can read the message supplied as `data` to this API. □

7 Related Work

Checking cryptographic API Usage. Mitchell et al. [19] introduce Security Annotations within a lambda calculus (discussed in Section 2.1); this paper extends this work to JavaScript. Recent work on cryptographic API use in Android applications shows that the majority of cryptographic bugs are due to misuse of APIs [16]; Egele et al. [8] show that such errors are common. Nadi et al. [20] survey usage of Java cryptographic APIs, and argue that the APIs are too low-level and require implicit understanding of the underlying cryptographic protocols. Krüger et al. [15] present *CrySL*, a domain specific language for the specification of correct usage of cryptographic APIs, focusing on the Java Cryptography Architecture. Our approach of encoding pre- and postconditions via security annotations on values and objects embraces the dynamicity of JavaScript which is notoriously difficult to statically analyze.

JavaScript analysis. While our approach is purely dynamic, various dialects allow for the static checking of JavaScript code [6,23,7,30]. The effective use of such static typing approaches would require modification of APIs and semantics, e.g., prohibiting byte array indexing of Key types. Design-by-contract systems

for JavaScript [13] enforce program properties directly expressible within the language. Our work focuses on security properties which cannot be directly expressed in this manner. Previous work on cryptographic testing for JavaScript focuses on implementations of the underlying cryptographic protocols. This work runs parallel to our own: we assume the correctness of these implementations and check existing usage of these APIs. Taly et al. [29] describe an automatic analysis to ensure security-critical APIs correctly protect resources from untrusted code. Domain-specific languages [3,14,2] have been proposed to enable verification of bespoke implementations by cryptographic experts. Existing programs are not amenable to this approach, since these languages are small subsets of JavaScript without many of the common idioms and advantages of the language.

Type-based approaches for Security. Type systems for F#, such as F7 [1,4,5] and F* [28], allow for the description of security properties of terms via dependent types which are checked statically. Static security type systems [24] to enforce secure information flow offer strong guarantees but have proved impractical in the JavaScript setting. Work on JavaScript monitors for information flow [10,26] provide mechanisms for dynamic enforcement of this in JavaScript; work on information flow monitoring in the presence of libraries [11] extends the applicability of monitor-based approaches. We follow a similar dynamic tag-based approach as such approaches [10], however we adopt a fine-grained system allowing for declassification coupled with precondition checking through annotation guards on functions. COWL [27,12] is an information flow control system for web browsers preventing third-party library code from leaking sensitive information, achieved via the labeling of browser contexts.

Formalizing JavaScript. Various formalizations of JavaScript exist [25,21,9,22]. λ_{JS} [9] and its successor S5 [22] provide a small language modeling the key features of JavaScript and have been extended to provide models for static and dynamic analyses. S5 remains close to the minimal lambda calculus described by Mitchell et al. [19], which allowed for a natural translation of Security Annotations.

8   Conclusions and Future Work

In this paper we described a formal model for Security Annotations in JavaScript, a mechanism to help non-expert developers avoid introducing security-critical bugs. We introduced a runtime semantics for Security Annotations in a core of JavaScript and presented a reference implementation of this system. We specified a partial fragment of the WebCrypto API in terms of Security Annotations, and demonstrated how to use it to detect a potential security vulnerability. Finally, we described the security guarantees offered by Security Annotations.

In future work, we plan to further develop Security Annotations as a runtime analysis for JavaScript by implementing them as an extension for the full language via source code instrumentation. The semantics described in this paper and accompanying implementation serve as a reference to guide the correctness of Security Annotations in full JavaScript.

References

1. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Prog. Lang. Syst.*, 33(2):8:1–8:45, 2011.

2. K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symp. on Security and Privacy (S&P)*, 2017.

3. K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Defensive JavaScript – building and verifying secure web components. In *Found. of Security Analysis and Design (FOSAD)*, 2014.

4. K. Bhargavan, C. Fournet, and N. Guts. Typechecking higher-order security libraries. In *Asian Symp. on Programming Languages and Systems (APLAS)*, 2010.

5. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symp. on Security and Privacy (S&P)*, 2013.

6. A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi. Fast and precise type checking for JavaScript. *Proc. ACM Prog. Lang.*, 1(OOPSLA):48:1–48:30, 2017.

7. R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Sys., Lang., and App. (OOPSLA)*, 2012.

8. M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM SIGSAC Conf. on Comp. and Comm. Security (CCS)*, 2013.

9. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Eur. Conf. on Object-Oriented Prog. (ECOOP)*, 2010.

10. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: Tracking information flow in JavaScript and its apis. In *ACM Symp. on Applied Computing*, 2014.

11. D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld. A principled approach to tracking information flow in the presence of libraries. In *Principles of Security and Trust (POST)*, 2017.

12. S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Principles of Security and Trust (POST)*, 2015.

13. M. Keil and P. Thiemann. Treatjs: Higher-order contracts for javascripts. In *Eur. Conf. on Object-Oriented Prog. (ECOOP)*, 2015.

14. N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symp. on Security and Privacy (EuroS&P)*, 2017.

15. S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. CrySL: Validating correct usage of cryptographic APIs. In *Eur. Conf. on Object-Oriented Prog. (ECOOP)*, 2018.

16. D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Asia-Pacific Workshop on Systems*, 2014.

17. B. Loring, D. Mitchell, and J. Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2019.

18. M. Madsen, O. Lhoták, and F. Tip. A model for reasoning about JavaScript promises. *Proc. ACM Prog. Lang.*, 1(OOPSLA):86:1–86:24, 2017.

19. D. Mitchell, L. T. van Binsbergen, B. Loring, and J. Kinder. Checking cryptographic API usage with composable annotations. In *ACM SIGPLAN Workshop on Partial Evaluation and Prog. Manipulation (PEPM)*, 2018.

20. S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: why do java developers struggle with cryptography APIs? In *Int. Conf. on Software Eng. (ICSE)*, 2016.

21. D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of JavaScript. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2015.

22. J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Symp. on Dynamic Languages (DLS)*, 2012.

23. A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang. (POPL)*, 2015.

24. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

25. J. F. Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner. JaVerT: JavaScript verification toolchain. *Proc. ACM Prog. Lang.*, 2(POPL):50:1–50:33, 2018.

26. J. F. Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of javascript. In *ICT Systems Security and Privacy Protection*, 2014.

27. D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining javascript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

28. N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ACM SIGPLAN Int. Conf. on Func. Prog. (ICFP)*, 2011.

29. A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Symp. on Security and Privacy (S&P)*, 2011.

30. P. Vekris, B. Cosman, and R. Jhala. Refinement types for TypeScript. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2016.

31. M. Watson. Web cryptography API. W3C recommendation, W3C, Jan. 2017.

A Syntax of S5

For convenience we provide the complete syntax of S5 from the work of Politz et al. [22] in Figure 7.

$$
\begin{aligned}
r := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textit{object references}\\
l := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textit{locations}\\
v := &\qquad \texttt{null} \mid \texttt{undefined} \mid \mathsf{str} \mid \mathsf{num} \mid \texttt{true} \mid \texttt{false} \mid r \mid \mathsf{func}(x,...)\{e\}\\
e := &\quad v \mid x \mid l \mid x := e \mid \mathsf{op1}(e) \mid \mathsf{op2}(e,e) \mid e(e,...) \mid e; e \mid \texttt{let}\ (x = e)\ e \mid \texttt{if}\ (e)\ \{e\}\ \texttt{else}\ \{e\}\\
\mid &\quad \texttt{label:}\ x\ e \mid \texttt{break}\ x\ e \mid \texttt{err}\ v \mid \texttt{try}\ e\ \texttt{catch}\ x\ e \mid \texttt{try}\ e\ \texttt{finally}\ e \mid \texttt{throw}\ e \mid \texttt{eval}(e,e)\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{ae\ \mathsf{str} : pe, ...\} \quad\textit{object literals}\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad e[\!\langle o\rangle\!] \mid e[\!\langle o\rangle = e] \quad\textit{object attributes}\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad\qquad e[e\langle a\rangle] \mid e[e\langle a\rangle = e] \quad\textit{property attributes}\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{props}(e) \quad\textit{property names}\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad e[e]^{e} \mid e[e = e]^{e} \mid e[\texttt{delete}\ e] \quad\textit{properties}\\
o := &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{class} \mid \texttt{extensible} \mid \texttt{proto} \mid \texttt{code} \mid \texttt{primval}\\
a := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{writable} \mid \texttt{config} \mid \texttt{value} \mid \texttt{enum}\\
ae := &\qquad\qquad\qquad\qquad [\texttt{class} : e, \texttt{extensible} : e, \texttt{proto} : e, \texttt{code} : e, \texttt{primval} : e]\\
av := &\qquad\qquad\qquad\qquad [\texttt{class} : v, \texttt{extensible} : v, \texttt{proto} : v, \texttt{code} : v, \texttt{primval} : v]\\
pe := &\quad [\texttt{config} : e, \texttt{enum} : e, \texttt{value} : e, \texttt{writable} : e] \mid [\texttt{config} : e, \texttt{enum} : e, \texttt{get} : e, \texttt{set} : e]\\
pv := &\quad [\texttt{config} : v, \texttt{enum} : v, \texttt{value} : v, \texttt{writable} : v] \mid [\texttt{config} : v, \texttt{enum} : v, \texttt{get} : v, \texttt{set} : v]\\
p := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad pv \mid [\,]\\
\mathsf{op1} := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{string\text{-}>num} \mid \mathsf{log} \mid \mathsf{prim\text{-}>bool} \mid ...\\
\mathsf{op2} := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{string\text{-}append} \mid + \mid \div \mid \mathsf{|>|} \mid ...\\
\theta := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{[av]\ \mathsf{str} : pv, ...\}\\
\sigma := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \cdot \mid \sigma, l : v\\
\Theta := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot \mid \Theta, r : \theta\\
E_{ae} := &\quad [\texttt{class} : E', \texttt{extensible} : e, \texttt{proto} : e, \texttt{code} : e, \texttt{primval} : e] \mid [\texttt{class} : v, \texttt{extensible} : E', \texttt{proto} : e, \texttt{code} : e, \texttt{primval} : e]\\
\mid &\quad [\texttt{class} : v, \texttt{extensible} : v, \texttt{proto} : E', \texttt{code} : e, \texttt{primval} : e] \mid [\texttt{class} : v, \texttt{extensible} : v, \texttt{proto} : v, \texttt{code} : E', \texttt{primval} : e]\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad [\texttt{class} : v, \texttt{extensible} : v, \texttt{proto} : v, \texttt{code} : v, \texttt{primval} : E']\\
E_{pe} := &\quad [\texttt{config} : E', \texttt{enum} : e, \texttt{value} : e, \texttt{writable} : e] \mid [\texttt{config} : v, \texttt{enum} : E', \texttt{value} : e, \texttt{writable} : e]\\
\mid &\quad [\texttt{config} : v, \texttt{enum} : v, \texttt{value} : E', \texttt{writable} : e] \mid [\texttt{config} : v, \texttt{enum} : v, \texttt{value} : v, \texttt{writable} : E']\\
\mid &\qquad\qquad\qquad\qquad \texttt{config} : E', \texttt{enum} : e, \texttt{get} : e, \texttt{set} : e] \mid [\texttt{config} : v, \texttt{enum} : E', \texttt{get} : e, \texttt{set} : e]\\
\mid &\qquad\qquad\qquad\qquad [\texttt{config} : v, \texttt{enum} : v, \texttt{get} : E', \texttt{set} : e] \mid [\texttt{config} : v, \texttt{enum} : v, \texttt{get} : v, \texttt{set} : E']\\
E' := &\quad \bullet \mid E' := e \mid v := E' \mid \mathsf{op1}(E') \mid \mathsf{op2}(E', e) \mid \mathsf{op2}(v, E') \mid E'(e, ...) \mid v(v, ..., E', e, ...) \mid E'; e \mid v; E' \mid \texttt{let}\ (x = E')\ e\\
\mid &\quad \texttt{if}\ (E')\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{throw}\ E' \mid \texttt{eval}(E', e) \mid \texttt{eval}(v, E') \mid \{E_{ae}\ \mathsf{str} : pe, ...\} \mid \{av\ \mathsf{str}_1 : pv, ..., \mathsf{str}_x : E_{pe}, \mathsf{str}_n : pe, ...\}\\
\mid &\quad E'[\!\langle o\rangle\!] \mid E'[\!\langle o\rangle = e] \mid v[\!\langle o\rangle = E'] \mid E'[e\langle a\rangle] \mid v[E'\langle a\rangle] \mid E'[e\langle a\rangle = e] \mid v[E'\langle a\rangle = e] \mid v[v\langle a\rangle = E'] \mid \mathsf{props}(E')\\
\mid &\qquad\qquad\qquad\qquad\qquad\qquad E'[e]^{e} \mid v[E']^{e} \mid v[v]^{E'} \mid E'[\texttt{delete}\ e] \mid v[\texttt{delete}\ E']\\
E := &\qquad\qquad\qquad\qquad\qquad\qquad E' \mid \texttt{label} :\ x\ E \mid \texttt{break}\ x\ E \mid \texttt{try}\ E\ \texttt{catch}\ e \mid \texttt{try}\ E\ \texttt{finally}\ e\\
F := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E' \mid \texttt{label} :\ x\ F \mid \texttt{break}\ x\ F \quad\textit{Exception Contexts}\\
G := &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E' \mid \texttt{try}\ G\ \texttt{catch}\ e \quad\textit{Local Jump Contexts}
\end{aligned}
$$

Fig. 7: The syntax of S5 [22].