Checking Cryptographic API Usage with Composable Annotations (Short Paper)

Duncan Mitchell
Department of Computer Science
Royal Holloway, University of London
United Kingdom

L. Thomas van Binsbergen
Department of Computer Science
Royal Holloway, University of London
United Kingdom

Blake Loring
Information Security Group
Royal Holloway, University of London
United Kingdom

Johannes Kinder
Department of Computer Science
Royal Holloway, University of London
United Kingdom

Abstract

Developers of applications relying on cryptographic libraries can easily make mistakes in their use. Popular dynamic languages such as JavaScript make testing or verifying such applications particularly challenging. In this paper, we present our ongoing work toward a methodology for automatically checking security properties in JavaScript code. Our main idea is to attach security annotations to values that encode properties of interest. We illustrate our idea using examples and, as an initial step in our line of work, we present a formalization of security annotations in a statically typed lambda calculus. As next steps, we will translate our annotations to a dynamically typed formalization of JavaScript such as λ_{JS} and implement a runtime checked type extension using code instrumentation for full JavaScript.

CCS Concepts • **Security and privacy** → *Software and application security*; • **Software and its engineering** → *Software verification and validation*;

Keywords JavaScript, type systems

ACM Reference Format:
Duncan Mitchell, L. Thomas van Binsbergen, Blake Loring, and Johannes Kinder. 2018. Checking Cryptographic API Usage with Composable Annotations (Short Paper). In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3162071

1 Introduction

Increased public awareness of privacy concerns on the Internet has led to software moving toward implementing strong cryptography by default, in a trend dubbed "ubiquitous encryption". For instance, web applications for messaging platforms now routinely implement client-side cryptography in JavaScript for true end-to-end encryption. Where TLS/SSL terminates security at the HTTP server, application-specific client-side cryptography can help secure modern distributed environments consisting of cloud-hosted servers and third-party content-distribution networks. Despite initial skepticism of browser-based cryptography [1, 16], a diverse range of cryptographic libraries and APIs for JavaScript has evolved over the last ten years. To avoid the pitfalls of purely JavaScript-based cryptography, the W3C has standardized the Web Cryptography API (*WebCrypto*), which is implemented directly in the browser [24].

The clash between the agile mindset of JavaScript developers and the requirements of secure software engineering leads to lingering problems: cryptographic APIs are often hard to use correctly, and security problems do not lead to failing test cases or visible errors. Even for a constant key, a cursory inspection of the encrypted messages will not raise suspicion. For example, the open source browser extension Cryptocat for end-to-end encrypted messaging suffered from critically weak security for over a year due to a subtle type coercion bug in the use of private keys [22].

Despite several initiatives to verify not just cryptographic protocols but reference implementations in software [2, 6], it is not yet feasible to prove correctness of mainstream implementations. Dynamic languages like JavaScript are notoriously difficult to formally reason about: the dynamic type system and use of reflection thwart the application of expressive type systems or theorem provers without significant manual intervention by cryptographic experts [3, 14].

In this paper, we describe our work in progress toward a new approach for checking the use of cryptographic APIs in client code that is compatible with the dynamic type system and common design patterns of JavaScript. Our goal is to arrive at an automatic testing methodology that does not require manual analysis by cryptographic experts. Our main idea is to use a system of security annotations that is

PEPM'18, January 8–9, 2018, Los Angeles, CA, USA

orthogonal to the existing type system. Security annotations express security properties of values, e.g., whether a value is a key of a certain length, or a cryptographically secure random value. They are propagated alongside regular type tags but follow their own semantics. Security annotations should be transparent to client code until they encounter an error, in which case the test program should fail.

We break down our research program into three phases: (i) formalization of security annotations and validation of our idea in a statically typed lambda calculus; (ii) adaptation to the dynamic type system of a formal description of JavaScript semantics such as λ_{JS} [11]; (iii) implementation for full JavaScript using code instrumentation [19].

We motivate our work using examples of JavaScript code that show how the full system of annotations—once implemented—could help to detect bugs in cryptographic API usage (§2). Security type annotations are added to the signature of an API function to specify required security properties on arguments (preconditions) and ensure security properties specified as annotation tags on return values or referenced objects (postconditions). We formally define composable security annotations as a lattice model and provide its notion of subtyping as well as the intricacies required to allow adding or removing annotations (§3). We then present step (i) of the research program outlined above, a formal definition and prototype implementation of a statically typed lambda calculus with security annotations (§4). We discuss the implications of type safety in our lambda calculus and required steps to continue our work (§5) before we compare to related work (§6) and conclude (§7).

2 Introducing Security Annotations

We begin by illustrating our idea of security annotations through two examples of security-critical JavaScript.

2.1 Example 1: Key Truncation

Listing 1 uses the WebCrypto API to create a random key and encrypt a message, but a security bug in line 7 causes eight bits of the key to be overwritten. To follow the control flow of the example, note that it relies on promises, a mechanism for asynchronous code execution heavily used in WebCrypto: API methods such as `importKey` immediately return a `Promise` object, whose method `then` takes a function to be executed once the operation is completed. The result of the operation is passed to this function as an argument.

We introduce two security annotations to illustrate how our approach would detect the bug at runtime: `CryptKey` denotes that a value is a valid key, and `CSRV` that a value is a cryptographically secure random value. The `importKey` API ensures that the resulting key has the `CryptKey` annotation attached; `getRandomValues` ensures that it returns the array passed as an argument with the `CSRV` annotation attached. The `encrypt` API requires its `key` argument to be annotated

```
1  let c = crypto, cs = c.subtle;
2  let pt = new TextEncoder().encode("my message");
3  let iv = c.getRandomValues(new Uint8Array(12));
4  let alg = { name: "AES-GCM", iv: iv };
5  let rnd = c.getRandomValues(new Uint8Array(32));
6
7  if (...) { rnd[3] = 3; }
8  cs.importKey("raw", rnd, alg, false, ["encrypt"])
9    .then(function(key) {
10     cs.encrypt(alg, key, pt).then(function(ct) {
11       console.log(new Uint8Array(ct));
12     });
13 });
```

Listing 1. A simple example of WebCrypto usage in which the security of the encryption is undermined.

with both `CryptKey` and `CSRV` (enforcing random session keys). The pre- and postconditions can be checked during testing; concretely, postconditions cause security annotations as additional type tags; preconditions check whether required annotations are present and throw an exception otherwise.

Security annotation tags are attached to values during execution. They are introduced and dropped through annotated postconditions of API functions or the dynamic semantics of operations. At runtime, the `CSRV` annotation is attached to the `Uint8Array` object passed to `getRandomValues`. However, the semantics for assignment will be defined such that modifying one of the array elements causes the array to lose the `CSRV` annotation, as it is no longer a secure random value. As a result the precondition check to fail upon entering the encrypt function on line 10.

2.2 Example 2: Establishing a Session in Signal

The library `libsignal-js` is used by the JavaScript-based desktop application for the Signal messaging platform. It provides cryptographic functionality through frontends for WebCrypto and primitives compiled from trusted C code to JavaScript. We discuss through example how annotating these primitives and the WebCrypto API can help test software using `libsignal-js` for security property violations.

Listing 2 demonstrates the flexibility of placing precondition enforcement burdens on trusted API calls rather than developer code. This function acts as a wrapper for a trusted elliptic curve library—in principle, we would annotate each API of this library in the same manner as proposed for WebCrypto. This function takes as argument a private key, asynchronously constructs a corresponding public key, and returns the public-private key pairing. The function `getRandomBytes` acts as a wrapper for WebCrypto's `getRandomValues`, which we annotate in the same manner as §2.1. A central advantage of our approach is that it allows to bypass the inherent ad-hoc polymorphism of this function: annotating

```
createKeyPair: function(privKey) {
  if (privKey === undefined) {
    privKey = Internal.crypto.getRandomBytes(32);
  }
  return
    Internal.Curve.async.createKeyPair(privKey);
}
```

Listing 2. The `Internal.Curve.createKeyPair` function in `src/Curve.js` of `libsignal-js`.[1]

`createKeyPair` by hand, one would have to account for a variable signature of the function leading to a complex and unintuitive description. By annotating only trusted APIs, we need not worry about about the polymorphism: instead of checking preconditions at the boundaries of the developer's code, we simply check preconditions at each API call.

3 A Lattice of Security Annotations

This section details the usage of security annotations within our approach. Annotations are declared locally in a library such that they induce a lattice embedding a hierarchy only on those annotations used by the library. We also introduce useful operators for manipulating annotations.

Declaration and composition. Cryptographic APIs have arguments and return values with several distinct security properties. For example, recall WebCrypto's `encrypt` API from §2.1, which enforces two security properties on the key argument, expressed via the annotations `CSRV` and `CryptKey`; these annotations are distinct—some cryptographic keys are not directly computed as random values (e.g., public keys). Since distinct security annotations are required for different trusted libraries, we propose that security annotations are declared on a per library basis. Thus, program testing only involves those annotations declared in the libraries used by the program. The composition of two or more annotations is expressed via the symmetric and associative operator $*$. For example, a value annotated with `CryptKey * CSRV` has both the `CryptKey` and `CSRV` security properties.

Hierarchies of annotations. Security properties have degrees of specificity, e.g., the property that a value is a cryptographic key is less specific than that of a private key. Annotations should thus not just be compositional but also have natural subtyping-like structures. We further propose that annotations may be declared as extensions of previously declared annotations, indicating that an annotation is more specific than any annotation it extends. For example, we write **`SecAnn`** `PrivKey` **`extends`** `CryptKey` to say that `PrivKey` represents a more specific property than `CryptKey`. A collection of declarations induces a subannotation relation $<:$, i.e., the declaration above determines that `PrivKey` $<:$ `CryptKey`.

The annotation lattice. We can naturally extend subannotations to annotations composed by $*$ as well. Formally, this is achieved by describing the extension of subannotations in two rules, [S-Width] and [S-Depth] (see Figure 3), inspired by record typing [15]. [S-Width] describes the intuition that composing a number of annotations is more specific than composing a subset of them. [S-Depth] encodes that individual subannotation relationships also extend over composition. These judgments, together with those defined by the **`extends`** declarations, induce a partial ordering of the program annotations and in turn a lattice of security annotations for each program under test. `Top`—the empty element—intuitively represents a lack of valid security properties.

Operators on annotations We define two additional operators of interest, the definitions of which are dependent on the individual program's annotation lattice. The first is the natural least upper bound operator, $\sqcup$, which finds the most specific annotation in the lattice such that both operands are a subannotation of it. The second operator is `cut`, which is motivated by the idea that security properties should be discarded when they are no longer valid. There is a subtlety here: superannotations of a removed annotation should still be valid when they are explicitly removed. For example the result of `cut(PrivKey*CSRV, PrivKey)` is `CryptKey * CSRV`, the most specific annotation such that `PrivKey` is no longer valid. We define $\texttt{cut}(S_1, S_2)$ formally as the unique annotation R with $S_1 <: R$ and $R \not<: S_2$ such that if there is some other S also satisfying these properties, then $R <: S$ and $S \not<: R$.

4 Formalizing Security Annotations

We present a minimal typed lambda calculus[2] with security annotations. We construct this small calculus to ensure the correctness and feasibility of the approach; a discussion of the translation of this formalization to dynamic languages is contained in §5. The calculus adds security annotations, which are defined for individual programs and manipulated through specific keywords. We outline the syntax of this calculus (§4.1), describe its dynamics (§4.2) and statics (§4.3), and discuss specific notions of type safety (§4.4).

4.1 Syntax

The syntax for the lambda calculus is given in Figure 1, which differs from that of a standard lambda calculus in several ways. Firstly, values in this calculus comprise a prevalue, corresponding to the values of a traditional lambda calculus, and a security annotation S. This allows us to represent how security properties on individual values change over

[1] https://github.com/WhisperSystems/libsignal-protocol-javascript, retrieved September 2017.

[2] A prototype implementation in Haskell of our calculus is available at https://github.com/ltbinsbe/lambda-security-tags.

P ::=		*programs*:
	$D \ldots D\ [t]^i$	annotations and program term
D ::=		*Annotation Declarations*:
	`SecAnn` a	new annotation
	`SecAnn` a `extends` a	annotation inheritance
t ::=		*terms*:
	x	variable
	v	annotated value
	$\lambda x : T\langle S\rangle.t$	abstraction
	$t\ t$	application
	if t then t **else** t	conditional
	let $x = t$ **in** t	let binding
	t as S	annotation introduction
	t drop S	annotation removal
w ::=		*prevalues*:
	$\lambda x : T\langle S\rangle.t$	abstraction prevalue
	true	true prevalue
	false	false prevalue
v ::=		*annotated values*:
	$w\langle S\rangle$	annotated prevalue
S ::=		*security annotations*:
	a	security annotation
	$S * S$	annotation composition
	`Top`	empty annotation
T ::=		*pretypes*:
	`Bool`	pretype of Booleans
	$A \to A$	pretype of functions
A ::=		*annotated types*:
	$T\langle S\rangle$	annotated type
Γ ::=		*contexts*:
	$\varnothing$	empty context
	$\Gamma, x : T\langle S\rangle$	term variable binding

Figure 1. Syntax of the lambda calculus.

time. Annotations are modified via the **as** and **drop** keywords. An **as** adds annotations, representing newly valid security properties, while **drop** removes the annotations, representing security properties that are no longer valid. Secondly, programs are prepended by a series of *annotation declarations*, as described in §3, which define the annotations available in the program together with the lattice they induce.

4.2 Dynamic Semantics

The evaluation rules, presented in Figure 2, follow those of a standard lambda calculus [15], extended with the **as** and **drop** constructs. As standard, $[x \mapsto v]\ t$ denotes the substitution of the value v in place of the free occurrences of variable x in term t. The rules for the **if** construct explicitly refer to prevalues in order to decide which branch to take. The rules for **as** and **drop** state that the result of evaluating `t as S` or

$$[\text{E-App1}]\ \frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad [\text{E-App2}]\ \frac{t_2 \to t_2'}{t_1\ t_2 \to t_1\ t_2'}$$

$$[\text{E-AppAbs}]\ \ (\lambda x : T\langle S\rangle.t)\langle S'\rangle\ v \to [x \mapsto v]\ t$$

$$[\text{E-LetV}]\ \ \textbf{let}\ x = v\ \textbf{in}\ t \to [x \mapsto v]\ t$$

$$[\text{E-Let}]\ \frac{t_1 \to t_1'}{\textbf{let}\ x = t_1\ \textbf{in}\ t_2 \to \textbf{let}\ x = t_1'\ \textbf{in}\ t_2}$$

$$[\text{E-IfTrue}]\ \ \textbf{if true}\langle S\rangle\ \text{then}\ t_2\ \textbf{else}\ t_3 \to t_2$$

$$[\text{E-IfFalse}]\ \ \textbf{if false}\langle S\rangle\ \text{then}\ t_2\ \textbf{else}\ t_3 \to t_3$$

$$[\text{E-If}]\ \frac{t_1 \to t_1'}{\textbf{if}\ t_1\ \text{then}\ t_2\ \textbf{else}\ t_3 \to \textbf{if}\ t_1'\ \text{then}\ t_2\ \textbf{else}\ t_3}$$

$$[\text{E-AsV}]\ \ w\langle S_1\rangle\ \text{as}\ S_2 \to w\langle S_1 * S_2\rangle$$

$$[\text{E-DropV}]\ \ w\langle S_1\rangle\ \text{drop}\ S_2 \to w\langle \text{cut}(S_1, S_2)\rangle$$

$$[\text{E-As}]\ \frac{t \to t'}{t\ \text{as}\ S \to t'\ \text{as}\ S} \qquad [\text{E-Drop}]\ \frac{t \to t'}{t\ \text{drop}\ S \to t'\ \text{drop}\ S}$$

Figure 2. Dynamic semantics of the lambda calculus.

$$[\text{S-Sub}]\ \frac{\Gamma \vdash t : T\langle S_1\rangle \qquad S_1 <: S_2}{\Gamma \vdash t : T\langle S_2\rangle}$$

$$[\text{S-ArrowSub}]\ \frac{S_2 <: S_1 \qquad S_3 <: S_4 \qquad \Gamma \vdash t : (T_1\langle S_1\rangle \to T_2\langle S_3\rangle)\langle S\rangle}{\Gamma \vdash t : (T_1\langle S_2\rangle \to T_2\langle S_4\rangle)\langle S\rangle}$$

$$[\text{S-Width}]\ \frac{}{*_{i=1}^{n+k} S_i <: *_{i=1}^{n} S_i} \qquad [\text{S-Depth}]\ \frac{\text{for each } i,\ S_i' <: S_i}{*_{i=1}^{n} S_i' <: *_{i=1}^{n} S_i}$$

Figure 3. Hierarchical security annotation judgments.

`t drop S` is the result of evaluating t with possibly updated annotations. In the case of **as**, new security properties are added (or existing ones strengthened) by composing the original annotation with S; for **drop**, annotations are removed by cutting S from the original annotation (see §3).

4.3 Static Semantics

Figures 3 and 4 present inference rules defining the static type system. We describe first rules relating to subannotations and then discuss other rules of particular interest.

Judgments for hierarchical security annotations. Recalling the discussion in §3, we embed this lattice of annotations into the type system. The first two rules of Figure 3 are standard subsumption rules enriched with annotations; they ensure that a term will type-check as any subannotation of the one attached: if a term is valid for many security properties, then it must still be valid for each individually. The lattice is encoded by the [S-Width] and [S-Depth], described in §3.

$$[\text{T-Var}]\ \frac{x : T\langle S\rangle \in \Gamma}{\Gamma \vdash x : T\langle S\rangle}$$

$$[\text{T-True}]\ \texttt{true}\langle S\rangle : \texttt{Bool}\langle S\rangle \qquad [\text{T-False}]\ \texttt{false}\langle S\rangle : \texttt{Bool}\langle S\rangle$$

$$[\text{T-Abs}]\ \frac{\Gamma, x : T_1\langle S_1\rangle \vdash t : T_2\langle S_2\rangle}{\Gamma \vdash (\lambda x : T_1\langle S_1\rangle.t)\langle S_3\rangle : (T_1\langle S_1\rangle \to T_2\langle S_2\rangle)\langle S_3\rangle}$$

$$[\text{T-App}]\ \frac{\Gamma \vdash t_1 : (T_1\langle S_1\rangle \to T_2\langle S_2\rangle)\langle S_3\rangle \qquad \Gamma \vdash t_2 : T_1\langle S_1\rangle}{\Gamma \vdash t_1\ t_2 : T_2\langle S_2\rangle}$$

$$[\text{T-Let}]\ \frac{\Gamma \vdash t_1 : T_1\langle S_1\rangle \qquad \Gamma, x : T_1\langle S_1\rangle \vdash t : T_2\langle S_2\rangle}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : T_2\langle S_2\rangle}$$

$$[\text{T-If}]\ \frac{\Gamma \vdash t_1 : \mathit{Bool}\langle\rangle \qquad \Gamma \vdash t_2 : T_2\langle S_2\rangle \qquad \Gamma \vdash t_3 : T_3\langle S_3\rangle}{\Gamma \vdash \textbf{if}\ t_1\ \texttt{then}\ t_2\ \textbf{else} t_3 : T_2\langle S_2 \sqcup S_3\rangle}$$

$$[\text{T-As}]\ \frac{\Gamma \vdash t : T\langle S_1\rangle}{\Gamma \vdash t\ \texttt{as}\ S_2 : T\langle S_1 * S_2\rangle}$$

$$[\text{T-Drop}]\ \frac{\Gamma \vdash t : T\langle S_1\rangle}{\Gamma \vdash t\ \texttt{drop}\ S_2 : T\langle \texttt{cut}(S_1, S_2)\rangle}$$

Figure 4. Typing judgments for the lambda calculus.

```
1 SecAnn A
2 SecAnn B
3 SecAnn C extends A
4 SecAnn D extends C
5
6 if true<> then true<D*B> else false<A*B>
```

Listing 3. Typing the **if** construct.

The* if *statement. In typing the **if** construct we want to retain the maximal amount of information possible without over-complicating our annotation lattice (e.g., via sum-like annotations). It is intuitive to consider the most expressive annotation valid along both branches (i.e. the $\sqcup$ operator); consider the trivial example in Listing 3. Since both `C` and `D` are not valid along the **else** branch, they cannot be subannotations of the resulting annotation. We therefore type this as `Bool<A*B> = Bool<D*B ⊔ A*B>`. The type after evaluating the program will be a subtype of this; this consequence of allowing a more expressive construct is discussed in §5.

Adding and removing security annotations. The typing rules for **as** and **drop** reflect that `t as S` and `t drop S` evaluate to the same prevalue as t with (possibly) modified annotations. The modified annotation corresponds to the prevalue's annotation at runtime, as determined by $*$ and `cut`.

4.4 Annotated Type Safety

The classical notion of type safety within this calculus merits discussion: type safety in this calculus is important to ensure that the flexible system of annotations introduced does not result in the ability of invalid programs to pass type checks. We formulate a notion of type safety based on the traditional notions of (i) *progress* and (ii) *preservation* [15], which we adapt to incorporate security annotations:

(i) A well-typed term is either a value, or can take a step of evaluation according to the rules in Figure 2.

(ii) If a well-typed term takes a step of evaluation, then the resulting term is also well-typed, i.e. if $t : T\langle S\rangle$ and $t \to t'$, then there is some $S' <: S$ such that $t' : T\langle S'\rangle$.

Here, a term t is *well-typed* if there is some type T and security annotation S such that $t : T\langle S\rangle$. A proof of these twin statements is straightforward and is therefore omitted. The differing notion of (ii) reflects our **if** construct, however we are unconcerned by this: programs cannot type as valid when they do not possess the necessary security properties.

5 Discussion

We discuss the lambda calculus and the formalization of security annotations in full dynamic languages, e.g., JavaScript.

On annotated type safety. Since security annotations can be arbitrarily modified inline by the programmer, it is possible to circumvent the notion of type safety by assigning annotations to terms to arbitrarily pass any type check; the developer would simply lose the benefits of using security annotations to enforce correct preconditions of secure APIs.

Dynamic enforcement of annotations. We intend this notion of annotations as a first step toward dynamic enforcement of security properties in JavaScript: we have constructed a kernel representation of this within a formal setting and demonstrated the guarantees which accompany this presentation. The transition to dynamic checking of security annotations within the calculus presented in §4 is straightforward: values comprise prevalues and annotations, so we can directly alter the evaluation rules to introduce enforcement mechanisms, i.e., the [E-AppAbs] rule would become

$$[\text{E-AppAbs'}]\ \frac{v = w\langle S_1\rangle \qquad S_1 <: S_2}{(\lambda x : T\langle S_2\rangle.t)\langle S_3\rangle\ v \to [x \mapsto v]\ t}.$$

This dynamically ensures the annotation is valid; a second evaluation rule would throw a security exception in the event that $S_1 \not<: S_2$. This is the first step; next, we plan to translate this model into an existing formal semantics for JavaScript, such as λ_{JS} [11].

Annotation-polymorphic functions. There is one further challenge within the central concept of checking security properties via attaching annotations as postconditions to functions. Namely, many functions will have distinct valid postconditions dependent on the security properties of the supplied value. For example, consider the JavaScript function `window.btoa` which takes a string and returns a Base64 encoded string. Clearly, any security properties valid on the argument should be still valid on the return value. Therefore, if the annotation passed to the function is S, we would expect

```
1 SecAnn Encoding
2 SecAnn Base64 extends Encoding
3
4 (let myBtoa = λ x : String<S>.
5   (((cpAnn x (window.btoa x)) drop Encoding) as
        Base64)
6 in myBtoa "SecretKey"<S>
```

Listing 4. Enforcing the security properties of `btoa`.

the annotation of the returned value to be $\texttt{cut}(S, \texttt{encoding}) * \texttt{Base64}$. In a purely static setting, annotation variables would be required to achieve this typing. However, in the dynamic setting we can introduce an operator **cpAnn** which copies the annotations from one term to another, e.g., via

$$[\text{E-cpAnnV}]\frac{v_1 = w_1\langle S_1\rangle \qquad v_2 = w_2\langle S_2\rangle}{\texttt{cpAnn}\ v_1\ v_2 \longrightarrow w_2\langle S_1 * S_2\rangle}.$$

Intuitively, this suffices in the dynamic setting: functions at most destroy every annotation on entry, or they manipulate the annotations present on a value by adding some fixed annotations or removing fixed annotations, representing the security properties of the function. This allows the propagation of security annotations through our dynamic system. As a simple example of this construct, we consider a wrapper `myBtoa` around a library function `window.btoa` in a language much like the one described in §4 with strings. In order to describe the security properties of such a function, we consider the example function in Listing 4. Here, the program will evaluate to a value with the desired annotation.

6 Related Work

Type systems for security property enforcement have been advocated by Bhargavan et al. [2, 5, 6, 9]. The dependently typed languages F7 and F* [20] seek to statically verify security properties in F# code. Security type systems [17] augment types with annotations specifying policies for secure information flow. Our approach is similar to this, but heavily relies on annotations changing over time, which in security type systems only occurs during declassification. Type qualifiers [8] are another related concept. They provide an extension of the type system via composable qualifiers that represent properties of the program terms; however, type qualifiers are ill-suited to untyped scripting languages [12].

Work extending the expressiveness of JavaScript's type system often builds distinct dialects [7, 23]. Such approaches, including industry tools Flow or TypeScript, allow for over-approximate static checking of programs that are written specifically in these dialects. TreatJS [13] is a higher-order design-by-contract system for run-time checking code contracts with native JavaScript: however, properties such as valid key generation cannot be encoded as native assertions without further instrumentation.

The work of Taly et al. [21] describes an automated analysis for security-critical JavaScript APIs that complements ours. We assume that the APIs themselves are secure but construct a mechanism to ensure correct API usage. ProScript [3, 14] is a domain-specific language for cryptographic protocols that builds on Defensive JavaScript [4] to allow developers and cryptography experts to work together to incorporate verified protocol implementations into JavaScript applications. We propose a methodology enabling developers to gain feedback on their implementation without requiring help by cryptography experts.

Recent work has demonstrated the tractability of logic-based symbolic verification techniques for JavaScript [18], building upon previous work [10] toward the semi-automatic verification of JavaScript programs. To verify individual programs, one must write precise logical specifications for each function: this can provide strong guarantees at the cost of requiring significant manual intervention by developers.

7 Conclusions and Future Work

In this paper we have presented the first step toward a new approach for the checking of cryptographic libraries usage within JavaScript. Our methodology places at a premium the need for any solution to be usable by non-expert developers. The annotation of trusted cryptographic libraries with pre- and postconditions in the form of security annotations promises to enable fully automatic testing for violations of the assumptions upon which security proofs rely.

We described the first stage of our research program by validating the concept within a statically typed lambda calculus, and presented a formal model of composable security annotations. We discussed the challenges for the next step of this approach in transforming our current construction to the dynamic setting, describing additional constructs required to handle real-world JavaScript code.

We aim to continue the research program by adapting our formalization to the dynamic setting using existing formalizations of JavaScript semantics such as λ_{JS} [11]. Finally, we will implement security annotations as a language extension to JavaScript using source code instrumentation. This allows to check API usage in a manner transparent to the code under test. Finally, we aim to develop a thorough set of pre- and postconditions for widespread cryptographic libraries, in particular the WebCrypto API.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, and James Patrick-Evans and Claudio Rizzo for their comments on an early draft of this paper. This work has been supported by the Centre for Doctoral Training in Cyber Security (EP/K035584/1) and the UK Research Institute in Verified Trustworthy Software Systems.

References

[1] Tony Arcieri. 2013. What's wrong with in-browser cryptography? https://tonyarcieri.com/whats-wrong-with-webcrypto (Last accessed: 22 November 2017).

[2] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (2011), 8:1–8:45.

[3] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *IEEE Symp. on Security and Privacy (S&P)*.

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Defensive JavaScript – Building and Verifying Secure Web Components. In *Foundations of Security Analysis and Design VII (FOSAD)*.

[5] Karthikeyan Bhargavan, Cédric Fournet, and Nataliya Guts. 2010. Typechecking Higher-Order Security Libraries. In *Asian Symp. on Programming Languages and Systems (APLAS)*.

[6] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with Verified Cryptographic Security. In *IEEE Symp. on Security and Privacy (S&P)*.

[7] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent Types for JavaScript. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[8] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[9] Cédric Fournet, Karthikeyan Bhargavan, and Andrew D. Gordon. 2011. Cryptographic Verification by Typing for a Sample Protocol Implementation. In *Foundations of Security Analysis and Design VI (FOSAD)*.

[10] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a Program Logic for JavaScript. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *European Conf. on Object-Oriented Programming (ECOOP)*.

[12] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symp. on Security and Privacy (S&P)*.

[13] Matthias Keil and Peter Thiemann. 2015. TreatJS: Higher-Order Contracts for JavaScripts. In *European Conf. on Object-Oriented Programming (ECOOP)*.

[14] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. 2017. Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach. In *IEEE European Symp. on Security and Privacy (EuroS&P)*.

[15] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

[16] Thomas Ptacek. 2011. JavaScript Cryptography Considered Harmful. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2011/august/javascript-cryptography-considered-harmful/ (Last accessed: 22 November 2017).

[17] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[18] José Fragoso Santos, Philippa Gardner, Petar Maksimovic, and Daiva Naudziuniene. 2017. Towards Logic-Based Verification of JavaScript Programs. In *Int. Conf. on Automated Deduction (CADE)*.

[19] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering, (ESEC/FSE)*.

[20] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*.

[21] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symp. on Security and Privacy (S&P)*.

[22] Steve Thomas. 2013. Decryptocat. https://tobtu.com/decryptocat.php (Last accessed: 22 November 2017).

[23] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[24] Mark Watson. 2017. *Web Cryptography API*. W3C Recommendation. W3C. https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/.