Formal Methods for Web Security[☆]

Michele Bugliesi, Stefano Calzavara[*], Riccardo Focardi

Università Ca' Foscari Venezia

Abstract

In the last few years, many security researchers proposed to endow the web platform with more rigorous foundations, thus allowing for a precise reasoning on web security issues. Given the complexity of the Web, however, research efforts in the area are scattered around many different topics and problems, and it is not easy to understand the import of formal methods on web security so far. In this survey we collect, classify and review existing proposals in the area of formal methods for web security, spanning many different topics: JavaScript security, browser security, web application security, and web protocol analysis. Based on the existing literature, we discuss recommendations for researchers working in the area to ensure their proposals have the right ingredients to be amenable for a large scale adoption.

Keywords: Formal methods, Web security, Survey.

1. Introduction

The Web is now part of everyone's life and it constitutes the primary means of access to many useful services with strict security requirements. As a result, vulnerabilities on the web platform may enable vicious attacks with catastrophic consequences, ranging from economic losses, e.g., in the case of attacks against payment providers like PayPal, to privacy violations, e.g., in the case of improper disclosure of electronic health records. Security-critical services are more and more supplied online today and this increases the need of effective defenses for the web platform.

Unfortunately, it is well-known that protecting online services is not easy at all, given the intrinsic complexity of the Web. The web ecosystem is variegate and includes a large number of different components and technologies, hence the attack surface against web applications is incredibly large: security flaws in the web browser may expose authentication credentials and sensitive data stored in

[☆]Work partially supported by the MIUR projects CINA, ADAPT and Security Horizons.

[*]Corresponding author.

Email addresses: bugliesi@unive.it (Michele Bugliesi), calzavara@dais.unive.it (Stefano Calzavara), focardi@unive.it (Riccardo Focardi)

web pages; vulnerabilities of web protocols may break the confidentiality and the integrity of the communication session; and errors in the web application code may lead to the inclusion of malicious contents in otherwise trusted web pages. Even experienced web developers and security practitioners have a hard time at taming this complexity, leading to the proliferation of security breaches.

As it normally happens in computer science, when some kind of process is too error-prone, *formal methods* come to the rescue. In the last few years, many security researchers proposed to endow the web platform with more rigorous, analytical foundations. Their goal is designing models which allow for a precise reasoning on web security issues and developing effective tools to make the Web a safer place, relieving at least part of this burden from the shoulders of web developers and browser vendors. Given the complexity of the Web, however, research efforts in the area are quite scattered around many different topics and problems, and it is not easy to understand the import of formal methods on web security so far. One natural question is whether formal methods have been successful in this field or whether they can only be considered a theoretical exercise as of now: practical applications are important to showcase the effectiveness of formal methods at dealing with the problems mentioned above and encourage the web security community to synergise efforts with the formal methods community.

Through this survey, we make the following contributions:

1. we identify the most important, though occasionally underestimated, challenges which must be faced by researchers interested in investigating the application of formal methods to web security (Section 3);
2. we collect, classify and review existing proposals in the area of formal methods for web security, spanning many different topics: JavaScript security, browser security, web application security, and web protocol analysis. We underline the practical applications of the different solutions and we identify several success stories among them (Sections 4-7);
3. we discuss recommendations for researchers working in the area of formal methods for web security to ensure their proposals have the right ingredients to be amenable for a large scale adoption (Section 8).

1.1. Scope of the Survey

In this survey, we review:

- models of common web technologies, like web browsers, and foundational studies on the semantics of scripting languages used by web developers;
- semantics-based tools for the verification and the enforcement of security properties on the web platform;
- alternative, provably sound designs of solutions aimed at replacing existing web technologies to improve their security.

Instead, given our declared goals, we do not review:

- novel security models and abstract proposals for the Web which have not been backed-up by an implementation and an (at least preliminary) on-field evaluation. The Web is a very heterogeneous and complex environment, hence it is impossible to assess the adequacy of new security mechanisms without any practical evaluation;
- tools and solutions which have not been formalized or proved correct with respect to a precise security definition, even if these proposals are loosely inspired by sound principles predicated in the formal methods literature, e.g., on type-safe programming languages or information flow control.

We also exclude from the present survey the rich research line on the verification of the TLS/SSL protocol used for secure communication on the Web. Though formal methods boast many success stories in this area, the topic would better fit a survey on protocol verification.

1.2. Organisation

At a high level, we observe that the proposals we survey can be divided in two main research lines:

(RL1) *security by construction*: some works advocate the usage of better languages and abstractions to make the Web a safer place. They typically recognize severe intrinsic limitations in the design of the current Web and propose a paradigm shift to improve it. These proposals are effective at solving the root cause of a security problem, but they typically require profound changes to existing web technologies and applications;

(RL2) *modelling, verification and enforcement*: some works propose models and algorithms to formalize and reason about the security of current web technologies. They devise solutions to make the Web a more secure place by exploiting the existing frameworks and standards at their best. These proposals may be sometimes sub-optimal in terms of effectiveness, but they do not impact too much on current web technologies.

These two research lines are thus largely complementary and equally important. The presentation in the next sections is based on this classification. When a formal model found successful practical applications we discuss them in a *Security Applications* paragraph.

1.3. Structure of the Survey

Section 2 provides some background information about the web platform and web security in general. Section 3 discusses the main challenges in the application of formal methods to web security. Sections 4 and 5 overview formal methods for web security from the browser perspective: specifically, Section 4 focuses on JavaScript security, while Section 5 discusses other relevant work on browser security. Section 6 presents formal methods for securing web applications. Section 7 discusses formal models for web protocols, aimed at analysing both browsers and web applications, as well as their remote interactions. Section 8 provides a perspective on the current state of the art and details recommendations for future proposals. Section 9 concludes.

2. Web Security in Pills

2.1. The Web Platform

Documents on the Web are provided in form of *web pages*, hypertext files connected to other documents via hyperlinks. The structure of a web page and all the elements included therein are defined by using a *markup language*, typically HTML, which is parsed and rendered by a web browser. Page contents can be dynamically updated by using *JavaScript*, a weakly-typed scripting language executed by the browser. JavaScript code can be included inside a web page and manipulate it by altering the *Document Object Model* (DOM), a tree-like representation of the web page. The ability to change the DOM, possibly as a reaction to user inputs, is useful to develop rich, interactive web applications.

Web pages are requested and served over the *Hyper Text Transfer Protocol* (HTTP), a request-response protocol based on the client-server paradigm. The browser acts as the client and sends HTTP requests for resources hosted at remote servers; the servers, in turn, provide HTTP responses containing the requested resources if available. All the HTTP traffic flows in clear, hence the HTTP protocol does not guarantee the confidentiality and the integrity of the communication. To protect the exchanged data, the *HTTP Secure* (HTTPS) protocol wraps plain HTTP traffic within a TLS/SSL encrypted channel.

Both HTTP and its secure variant HTTPS are *stateless* protocols, i.e., each request is treated by the server as independent from all the other ones. Some web applications, however, need to remember information about previous requests, for instance to track whether a user has already performed the expected steps of a payment procedure. HTTP *cookies* are the most common mechanism to maintain state information about the requesting client and implement *sessions* on the Web. Roughly, a cookie is a key-value pair, which is set by the server into the client and automatically attached by it to all subsequent requests to the server. Cookies may either directly encode state information or, more commonly, just include a unique session identifier allowing the server to identify the requesting client and restore the corresponding session state when processing multiple requests by the same client.

Figure 1 represents the ingredients of the web platform introduced so far.

2.2. Web Threats

Traditionally, web security deals with two main families of attackers: *web attackers* and *network attackers*. A web attacker controls at least one server that responds to any HTTP(S) request sent to it with arbitrary malicious contents chosen by the attacker. Network attackers extend the capabilities of web attackers with the ability of detecting and intercepting all the traffic sent between two network endpoints. These attackers have the possibility of inspecting, forging and corrupting all the HTTP traffic sent on the network, but they cannot break cryptography. Though network attacks are arguably more difficult to carry out than web attacks, they may have catastrophic consequences, since they grant the attacker full control over web pages served over HTTP.

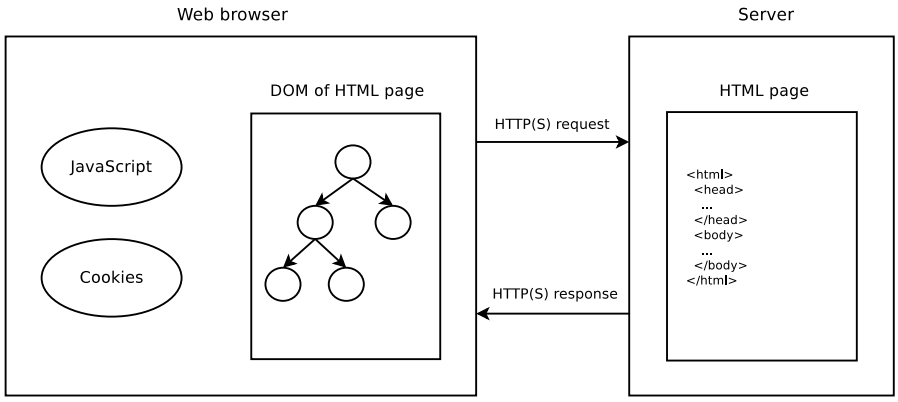

Figure 1: The Web Platform

The baseline defense mechanism offered by web browsers against these attackers is the *same-origin policy* (SOP), an access control policy enforcing a strict separation between contents provided by different web *origins*. An origin is defined as a triple including a protocol (typically HTTP or HTTPS), a host (roughly corresponding to a website) and a port number [9]. As a result of the SOP, for instance, scripts running in a page downloaded from `http://attacker.com` cannot access cookies set by `http://trusted.com`, which is a prerequisite to ensure that web attackers cannot disclose cookies identifying sessions with trusted websites and hijack them.

Unfortunately, the SOP is not enough to prevent many common attacks. For our present endeavours, it is worth mentioning only two notable examples:

1. *code injection*: a missing or flawed sanitization of user inputs in a web application may lead to the inclusion of attacker-controlled contents into benign web pages. Since these injected contents are indistinguishable from legitimate ones and inherit their origin, they may be entitled to access sensitive data provided by the benign pages, e.g., cookies, without violating the SOP. The injection of malicious JavaScript code is one of the most pervasive attacks on the Web, known as *cross-site scripting (XSS)*;
2. *cross-site request forgery (CSRF)*: since the SOP does not constrain cross-site requests, a page from `http://attacker.com` can force the browser into sending HTTP(S) requests to `http://trusted.com`. Since all these requests automatically include cookies previously set by the latter website, they will be considered part of the session between the user browser and `http://trusted.com`. These requests may thus be abused to trigger dangerous side-effects on the website on the user's behalf.

3. Challenges for Formal Methods

There are many reasons why approaching web security with formal methods is hard, we discuss the most important ones based on our experience. The first reason is definitely the inherent *complexity of the web platform*. There is an impressive number of different web standards and technologies nowadays, and most of them are based only on informal RFCs. The HTML5 specification alone spans hundreds of pages [85] and browser vendors often implement the same directives in different ways, since some subtle corner cases are underspecified [80]. This means that it is not obvious to identify which aspects of the web platform are worth modelling and, occasionally, it is not even clear how to model them. Testing on available implementations is sometimes the only way to understand how to correctly model some unclear behaviour.

A second challenge for the area is the *massive user base of the Web*, which has a number of subtle consequences. First, there is a continuous evolution in the specifications, corresponding to limitations and security threats being routinely discovered and fixed; this means that also formal models of the Web should often be updated to be useful, but this process requires both time and expertise. Moreover, the sheer size of the Web implies that the backward compatibility of new security solutions is just as important as their soundness: some security problems of the Web, like the lack of cookie integrity against network attackers [92], are well-known and not hard to fix in a formal model, but they are not fixed in the real Web, since it is unclear how to do it without breaking existing websites. Finally, it is also difficult to put assumptions on what may be expected from web developers, and thus reasonably assumed in a formal model, since large-scale evaluations on the Web often reveal surprises and disrupt widely believed folklore [75, 65, 24].

Besides all these problems, one of the biggest challenges into approaching web security is that *the Web is very peculiar in its own rights*. Though existing methodologies and experiences from other research areas can be ported to the Web, it is not easy to do so, since all the interactions there are mediated by a web browser and make use of the HTTP(S) protocol. For instance, formal methods for protocol verification surely help in analysing web protocols like single sign-on protocols, but they cannot be directly applied to them without missing dangerous attacks [41]. The reason is that the browser is an unusual protocol participant, which acts asynchronously and does not simply follow the protocol specification, but does a number of concurrent operations in the meanwhile. As we anticipated, this also motivates the need for different threat models which naturally take browser-based attacks into account.

4. The Browser View: JavaScript Security

4.1. RL1: Security by Construction

The semantics of JavaScript includes many quirks and surprises. For instance, the + operator performs several type coercions and is heavily overloaded, resulting in a specification based on a 15-step algorithm using meta-functions

spanning three pages: we refer to [44] for an interesting discussion about this and other peculiarities of the language. Moreover, there are often inconsistencies between different browser implementations of the JavaScript engine, which lead to the same JavaScript program behaving slightly differently on different web browsers. For these reasons, several research papers advocate the usage of more disciplined programming languages to develop the client-side part of a web application. To ensure backward compatibility with existing web browsers, these languages are either securely compiled to JavaScript [37, 83, 43] or amount to well-behaved JavaScript subsets enjoying some form of type safety [12].

4.1.1. A Fully Abstract Compilation to JavaScript [37]

Fournet *et al.* proposed a technique to develop secure JavaScript applications by compilation from a more sophisticated programming language amenable to static, type-based verification [37]. Specifically, they presented a compiler from F^* [82], a dependently-typed functional language similar to F#, into JavaScript and they prove a *full abstraction* result, stating that two programs are equivalent in all F^* contexts if and only if their translations are equivalent in all JavaScript contexts. Roughly, a context is defined as an arbitrary piece of code (library) interfacing with the compiled program: the full abstraction result then means that web application developers can reason on the security of their code in terms of the clear, well-established F^* semantics, without the need to understand the inner workings of the compiler or the semantic quirks of JavaScript. For instance, programmers can rely on their knowledge of static scopes and types à la F#, or leverage the type-based verification supported by F^* to prove security invariants of their application code.

4.1.2. Defensive JavaScript [12]

Bhargavan *et al.* defined Defensive JavaScript (DJS), a strict subset of JavaScript for developing security APIs to be embedded in potentially untrusted web pages, in order to support a safe interaction with online service providers like cloud storage services [12]. These APIs typically store sensitive data, like authentication credentials or cryptographic keys, needed to access the service provider: protecting these data despite the compromise of the embedding web page is important for protecting the user account at the provider. DJS supports the development of JavaScript security components which protect their secrets even when loaded into an untrusted page, possibly after other scripts have tampered with the execution environment. These guarantees are achieved by syntactic restrictions on the JavaScript program, enforced by a static type system which provably guarantees a formal notion of defensiveness against the aforementioned attacks.

Given the power of the considered threat model, the syntactic restrictions underlying DJS are harsh: for instance, DJS programs may not access variables or call functions that they do not define themselves, and all the used variables must be lexically scoped. Despite these limitations, the authors successfully applied DJS to develop three non-trivial applications, including a library for browser-side cryptography.

4.1.3. TS: Secure Gradual Typing for JavaScript [83]*

Swamy *et al.* developed *TS**, an experimental programming language supporting many of the dynamic programming idioms of JavaScript, while ensuring type safety even in presence of an untrusted JavaScript environment [83]. A compiler from *TS** into JavaScript allows the deployment of *TS** applications into standard web browsers. The *TS** type system consists of a standard static core and a dynamic fragment, so it is technically a *gradual* type system: dynamically typed code and its interactions with statically typed code are instrumented for safety by the *TS** compiler, by means of type-directed wrapper functions.

The most distinguishing feature of *TS** is the presence of a special type, Un, which is given to any JavaScript program which cannot be typed either statically or dynamically: this represents attacker-controlled code and the type safety theorem of *TS** ensures that Un-typed data cannot break the invariants provided by stronger types. The authors evaluated the effectiveness of their proposal by migrating existing JavaScript code to *TS** and by developing a few utilities from scratch, like a JSON parser.

4.1.4. Verified Security for Browser Extensions [43]

Guha *et al.* proposed a methodology to develop provably secure browser extensions [43]. Their core idea is to develop browser extensions using Fine, a dependently-typed ML dialect, and to use its powerful type system to statically prove that extensions comply with an intended security policy. After verification, extensions are automatically compiled into JavaScript, thus allowing their deployment in multiple browsers; this last step of the development process is not covered by the soundness proof and its correctness is only discussed informally. The security policies supported by the approach are quite general, since they are written in a simple language reminiscent of Datalog, allowing the specification of fine-grained authorization and data flow invariants on web content and browser state accessible by extensions. The authors proved the effectiveness of their proposal by developing 17 browser extensions using Fine.

4.1.5. The Mashic Compiler [54]

Luo *et al.* designed the Mashic compiler to improve mashup security [54]. A *mashup* is a website loading and integrating contents (typically called *gadgets*) from different web sources to create a new service. There are two ways to include gadgets in a mashup:

- using HTML *script tags*: in this case, the gadget is directly embedded in the integrating web page and inherits the origin of the latter. This implies that the gadget runs with the same privileges of the integrator;
- using HTML *iframe tags*: in this case, the gadget is loaded in an isolated environment and preserves its own origin, hence the SOP limits its capabilities on the integrating web page. The interactions between the gadget and the integrator are limited to message passing.

Unfortunately, web developers typically sacrifice security for programming convenience and implement mashup by making use of script tags. The Mashic compiler takes in input an existing mashup and generates a secure mashup based on iframe tags and message passing. The paper presents two formal results: a *correctness* result, proving that the output of the Mashic compiler is equivalent to the original mashup when the embedded gadget is "benign"; and a *security* result, proving confidentiality and integrity properties for the compiled mashup. The Mashic compiler has been implemented and tested on realistic case studies.

4.2. RL2: Modelling, Verification and Enforcement

The main contributions in this research line amount to the definition of rigorous semantics for JavaScript. These semantics may be defined either by providing a direct formal counterpart of the official language specification [56, 84] or by giving a semantics-preserving translation of JavaScript into a simpler core language [44, 70]. Having a rigorous semantics for JavaScript is crucial to retrofit the security of existing web applications.

4.2.1. An Operational Semantics for JavaScript [56]

The first operational semantics for a significant fragment of JavaScript was proposed by Maffeis *et al.* [56]. It is a massive small-step semantics, spanning around 60 pages of rules and definitions in ASCII format. The semantics was developed by closely following the official language specification (the 3rd edition of the ECMAScript standard) and it was complemented by hands-on testing on existing JavaScript engines to iron out the trickiest details. Remarkably, such testing unveiled several points where existing implementations openly deviated from the official specification. The operational semantics was shown to enjoy two main formal results:

- a *progress* theorem, proving that the execution of a program always progresses to an exception or a value of an expected form;
- a *heap reachability* theorem, proving that the behaviour of a program depends only on a specific (computable) portion of the heap.

Security Applications: The JavaScript semantics in [56] found several security applications by the same research group, most notably to reason about sound techniques to isolate untrusted JavaScript and limit its capabilities on a trusted embedding page [57, 59, 58]. These proposals make use of different techniques:

- *filtering* [57, 59]: removing certain language constructs which are hard to tame. For instance, the `eval` construct evaluates and executes a string representation of a JavaScript program. Since strings are values and can be the result of arbitrarily complex JavaScript computations, it may be difficult to identify what is actually executed by an invocation to `eval` and it may be more convenient for security to just drop the construct;

- *rewriting* [57, 59]: wrapping dangerous functions to provide reduced functionalities and renaming sensitive properties (fields) to prevent access to their original version. For instance, an array access $e_1[e_2]$ could be rewritten to insert runtime checks ensuring that e_2 does not evaluate to a blacklisted property name;

- *object-capability models* [58]: ensuring that accesses to sensitive resources are only granted to code which is endowed with a corresponding capability. If this is the case and a script only possesses the capabilities that it is explicitly given by the web developer, then isolation between two scripts may be enforced by granting them disjoint capabilities.

These techniques have not only been proposed in theory, but also successfully applied to find bugs in popular libraries for JavaScript sandboxing, like Facebook FBJS, Google Cajita and Yahoo! AdSafe. We refer to the original papers [57, 59, 58] for full details on the main findings.

4.2.2. λ-JS and S5: Taming JavaScript by Desugaring [44, 70]

Given the daunting complexity of the operational semantics in [56], Guha *et al.* proposed a completely different approach to formally capture the behaviour of JavaScript [44]. Their idea is to reduce JavaScript to a core, relatively standard calculus using a complex *desugaring* process, which translates JavaScript constructs with a sophisticated semantics into (possibly long) sequences of core constructs with a simpler semantics. The core calculus is called λ-JS and it is expressive enough to encode the entire JavaScript specification, with the exception of the `eval` construct: its semantics is compact, spanning only 3 pages of definitions and rules, as opposed to the 60 pages of the semantics in [56].

The definition of λ-JS comes with two software artifacts: a desugarer (compiler) from JavaScript to λ-JS and a λ-JS interpreter. The adequacy of desugaring was empirically assessed by picking a large benchmark of existing JavaScript programs and by comparing the λ-JS interpreter behaviour on the desugared code against the behaviour of available JavaScript engines on the original programs. The λ-JS semantics later evolved thanks to Politz *et al.* into S5 [70], a second core language defined by desugaring the 5th edition of the ECMAScript specification. Most notably, S5 implemented support for getters and setters (not available in JavaScript when λ-JS was defined) and for the `eval` construct (not modelled in λ-JS).

Security Applications: λ-JS was adopted by Lerner *et al.* as a building block for TeJaS [51], a sophisticated framework for developing static type systems for JavaScript. TeJaS found two main applications to security: Politz *et al.* used it to verify the Yahoo! AdSafe library for sandboxing untrusted JavaScript [71], while Lerner *et al.* used it to check the compliance of browser extensions with the incognito mode available in modern web browsers [50]. Both these studies have been highly successful: in particular, the type-based analysis in [71] found some subtle implementation bugs in Yahoo! AdSafe, while the research in [50] identified private-browsing violations in 6 of 12 analysed browser extensions.

A core of λ-JS was also the language chosen by Calzavara *et al.* [23] to design a sound static analysis to detect privilege escalation attacks on browser extensions, where malicious or compromised extension components abuse the message passing interface available to them to unduly gain access to security-sensitive functionalities. The static analysis was implemented in a prototype analyser for Google Chrome extensions, called CHEN.

More recently, a significant fragment of λ-JS was used by Devriese *et al.* [32] to formulate and prove sound a novel approach for reasoning about object-capability languages using logical relations. A key quality of their model is the ability to define *custom* capabilities, which require semantics-based formal tools to reason about their soundness, in contrast to standard syntactic approaches used in the area. The paper discusses a few applications of the model to web security, including the isolation of untrusted advertisement and mashup security.

4.2.3. SESLight [84]

Taly *et al.* defined SESLight, the formalization of a JavaScript fragment inspired to the *strict mode* semantics of the 5th edition of the ECMAScript specification [84]. The strict mode is more amenable to static analysis than plain JavaScript, since it supports static scoping and closure-based encapsulation. Compared to the standard strict mode of JavaScript, SESLight additionally prevents by construction the malicious use of some built-in objects by making them immutable and it only supports a limited usage of the `eval` construct. These choices are important for confining malicious code and to better support static analysis.

Security Applications: The SESLight semantics was used to carry out a foundational study on the effectiveness of the sandboxing mechanisms available in publicly available JavaScript APIs, like those provided in the Yahoo! AdSafe library. The authors presented a sound static analysis based on Datalog clauses, which can be used to analyse security-focused JavaScript libraries developed in SESLight and determine whether they are secure against arbitrary untrusted code written in the same language. A positive analysis result ensures that no sequence of API calls returns a direct reference to a security-critical object, which should not be accessed by untrusted code. Remarkably, the static analysis was able to find a security issue in the code of the Yahoo! AdSafe library, as well as to verify the robustness of a revised implementation.

4.2.4. Additional Related Work

Besides the work above, there are many frameworks for enforcing general security / information flow policies on untrusted JavaScript code, and some of them provide formal soundness guarantees [91, 69, 40, 45]. Information flow policies in particular [40, 45] are very flexible tools and they can be used for a variety of security purposes, ranging from the enforcement of JavaScript sandboxing to the prevention of attacks like XSS and CSRF. Though all these proposals are notable and definitely worth mentioning here, we do not review them in detail, since they are already covered by a recent survey by Bielova [13]. That survey provides a good overview on what has been done in the area of security policies

for JavaScript and it clearly highlights which formal guarantees are provided by each of the reviewed proposals.

Also, it is important to point out that the quest for a trusted, fully accurate JavaScript semantics is still going on. Recent work in the area comprises the specifications of JSCert [17] and KJS [68], two major endeavours at defining mechanised and highly accurate formal semantics for JavaScript. KJS in particular is the most recent effort in the area and its authors identified several errors and inconsistencies in previous proposals, including JSCert and S5. The only security application of KJS as of now is discussed in the original paper presenting the semantics, where a known security vulnerability was rediscovered by using symbolic execution. We believe that both JSCert and KJS will likely find more important security applications in the next future.

5. The Browser View: Beyond JavaScript

5.1. RL1: Security by Construction

This research line spans two different areas. First, there are works exploring novel browser designs, which are more robust than current standards and amenable for formal verification of useful security properties [39, 78]. Then, there are proposals aimed at filling the gap between the formal verification of the security guarantees offered by a browser design and the actual implementation of a web browser, by synthesising the latter from a verified core [48]. This is important, since low-level flaws in the browser code may easily void all the security guarantees granted by formal verification.

5.1.1. The OP(2) Web Browser [39]

OP is an experimental web browser developed by Grier *et al.*, which later evolved into the more sophisticated OP2 [39]. The main differences between OP and OP2 are not relevant to the present discussion, so we just note the browser as OP(2). The OP(2) web browser shares the same design principles of modern micro-kernels: it is made of several distinct and isolated components, and all the interactions between them are mediated by a browser kernel. This kernel enforces an access control policy disciplining the use of security-critical functionalities, so that, even if a component was compromised, it would still be isolated from the other ones and the threats it may pose to the security of the entire browser would be limited to what the kernel allows to the component as part of its normal (non-compromised) behaviour. This design also simplifies formal verification, since all the browser components expose a simple API and most of the formal reasoning can be confined to it.

The authors developed a model of the OP(2) browser in Maude [61], a framework for encoding the semantics and model-checking properties of systems specified in rewriting logic. The Maude model was used to verify two useful security properties of the web browser design:

- *protection against address bar spoofing*: the page content domain is the same as the domain displayed in the address bar. This property can be

broken by subtle attacks which have affected major web browsers in the past, see [62] for a few examples on Internet Explorer 6;

- *correct enforcement of the SOP*: the implementation of the SOP in OP(2) correctly limits the interactions between browser plugins and JavaScript. This should not be taken for granted, see [80] for a discussion about dangerous incoherences of web browser access control policies.

5.1.2. The IBOS Web Browser [78]

The IBOS web browser by Sasse *et al.* aims at pushing the security boundaries advocated by OP(2) even further [78]. In particular, IBOS extended the modularity principles underlying OP(2) also to the operating system, to remove almost all its traditional components and services from the browser trusted computing base. The kernel of IBOS thus includes both browser and operating system functionalities, and it is the only trusted component of the design.

The formal verification of the IBOS kernel was carried out in Maude [61], the same tool used to check the security of OP(2), and the verified security properties were also the same. The main new formal contribution was the presentation of two "small model" theorems, ensuring that the bounded verification performed in Maude actually generalizes to the unbounded case.

5.1.3. The QUARK Web Browser [48]

QUARK is a prototype web browser developed by Jang *et al.* [48]. It builds on the same design principles of OP(2) and IBOS, i.e., a modular architecture where isolated browser components interact through a kernel mediating access to sensitive resources. In contrast to the former proposals, however, formal verification was not carried out on a model of QUARK, but directly on its implementation. The QUARK kernel was developed in the Coq proof assistant and verified using its interactive theorem proving facilities; the kernel code was then compiled into a certified Ocaml program, using the machinery available in the Coq tool chain. The security guarantees provided by verification only depend on the kernel: all the other components are deemed untrusted, which allowed the reuse in QUARK of existing state-of-the-art implementations of complex browser components (e.g., the JavaScript engine).

The security properties verified for QUARK can be summarised as follows:

- *tab non-interference*: no tab can ever affect how the kernel interacts with another tab;
- *cookie confidentiality and integrity*: cookies for a domain can only be accessed by a tab which originally displayed a page on that domain;
- *address bar integrity*: the address bar cannot be modified by a tab without the user being involved, and always displays the correct address.

These properties do not hold for standard web browsers: for instance, a compromised tab in Google Chrome can leak all the cookies of any domain. Providing this additional level of security comes at the cost of compatibility and prevents some web applications from working correctly when accessed using QUARK.

5.2. RL2: Modelling, Verification and Enforcement

The work in this research line complements formal JavaScript semantics to embrace a wider range of web threats. Several of these proposals define web browser models [41, 90, 19]: this is important, since JavaScript is not the only weapon available to web attackers and subtle attacks may arise due to unexpected interactions between different browser components. Given the impressive number of features covered by these models, individual browser components are represented quite abstractly to achieve a tractable level of detail. Other papers thus target more accurate representations of selected browser components, like the DOM [38, 77, 60, 73]. Modelling the DOM is important for security, because many sensitive data like passwords and credit card numbers routinely enter the DOM when they are input in the web browser.

5.2.1. The B Brower Model [41]

The B browser model by Gross *et al.* was the first core browser model proposed in the literature [41]. The goal of the model was providing a foundation for reasoning about the security of browser-based protocols, e.g., identity federation protocols like SAML [66] and Shibboleth [25]. The model only represents the input/output behaviour of the web browser and it does not include cookies, nor any scripting language.[1] The model is based on a formalism reminiscent of I/O automata [55] and it supports a concise graphical representation analogous to UML state diagrams. A peculiar feature of the model is an explicit representation of the user behaviour, which plays an important role in the security analysis of browser-based protocols.

Security Applications: The original paper [41] applied the browser model to establish a security result for password-based authentication, a basic building block of most browser-based protocols. The model was also used to formally prove the security of a specific identity federation protocol called WSFPI [42].

5.2.2. Yoshihama's Browser Model [90]

Yoshihama *et al.* proposed a fairly sophisticated browser model [90]. The model formalizes the browser using a big-step operational semantics, covering the evaluation of client-side scripts, the presence of multiple browser windows, the DOM, cookies and HTTP requests. The model includes several non-trivial features of real web browsers, like document content that may reference external resources (such as `<img>` and `<script>` tags), DOM mutation operations, an `eval` construct for dynamic code evaluation, first-class functions, and event handlers. Unfortunately, the formalism was only explained by a few inference rules demonstrating how one might give a big-step semantics for a web browser, but it is not rigorous or complete enough to be usable in formal proofs.

Security Applications: The browser model in [90] uses information flow labels for fine-grained access control on websites, focusing on the secure integration of contents from different, mutually distrusting websites (mashup security). In

[1] The authors assume these features are turned off, which is unrealistic for the current Web.

this view, websites define sets of labels with access control properties and attach these labels to portions of their HTML documents. The labels are then automatically propagated by the web browser and tracked on individual DOM nodes and script variables to enforce access control checks. Implicit information flows where secrets are leaked by the execution of conditional program branches depending on private data are not covered by this proposal.

5.2.3. Featherweight Firefox [19, 18]

Featherweight Firefox is a core formal model of a web browser, originally developed in Ocaml by Bohannon and Pierce [19] and later rewritten in Coq by Bohannon [18] to allow for the development of mechanized proofs about browser behaviours. The model is a *reactive system*, i.e., an event-driven state machine which consumes inputs and produces outputs in response. Featherweight Firefox is primarily intended to be a "blank slate" for security researchers interested in developing new browser-side security policies and mechanisms amenable for formal verification. The formalization covers the basic aspects of windows, DOM trees, cookies, HTTP requests and responses, user inputs, and a simple scripting language including the most significant features of JavaScript, like first-class functions, dynamic evaluation of strings into code, and AJAX requests.

Security Applications: Featherweight Firefox found quite a number of application as of now. The original Ocaml model was extended by Bielova *et al.* [14] with *secure multi-execution*, a provably sound runtime technique for enforcing non-interference policies, based on multiple executions of the web browser and the application of a specific input/output policy to each execution to ensure that secret inputs do not flow to public outputs [33]. The extended browser model was shown to enforce non-interference for arbitrary information flow policies. The Coq model, instead, was enriched by Bohannon [18] with a number of runtime checks not performed by standard web browsers, so as to dynamically enforce a strong, precise confidentiality policy preventing cross-site requests. The correctness of the enforcement was established by leveraging the interactive theorem proving facilities of Coq.

The Coq model was also used by Bugliesi *et al.* to develop a mechanized proof of non-interference, assessing the effectiveness of the HttpOnly and Secure cookie attributes available in modern web browsers to prevent the unintended disclosure of cookies identifying web sessions [21]. Specifically, [21] proved that the value of the HttpOnly cookies set by trusted web applications has no visible import for a web attacker on a different domain; and, if these cookies are also Secure, their confidentiality is also guaranteed against network attackers. The same research group used a pen-and-paper subset of the Featherweight Firefox model to define a formal notion of *web session integrity* and study its browser-side enforcement [22]. Web session integrity ensures that an attacker can never force the browser into introducing unintended messages in sessions established with trusted websites, or into leaking the authentication credentials (cookies and passwords) associated to these sessions. The paper presents a security-enhanced browser semantics, which provably enforces session integrity for any web session.

5.2.4. Bauer's Browser Model [10]

Bauer *et al.* presented a detailed model of a web browser in terms of a labelled transition system [10]. The model formalises the browser using a small-step operational semantics, including tabs, a core of the DOM, event handling, cookies, bookmarks, history, user actions and an accurate model of browser extensions. The scripting language is quite simplified and it does not cover all the subtleties of JavaScript. The model uses standard mathematical notation and it has not been mechanized.

Security Applications: The browser model in [10] includes a lightweight, coarse-grained form of taint tracking, which is proved to enforce non-interference. The model includes a number of features which are useful for a practical deployment, including declassification and endorsement mechanisms to downgrade confidentiality and integrity respectively. Indeed, the taint tracking mechanism was implemented in Chromium and tested on existing websites, with promising preliminary results. The authors give evidence that their proposal is expressive enough to encompass standard security policies currently implemented in web browsers, including the SOP.

5.2.5. Modelling and Reasoning About the DOM

The first attempt at formalizing a significant portion of the DOM specification is due to Gardner *et al.* [38]. They presented a core imperative language with a simple operational semantics, called Minimal DOM, which allows for representing tree updates in DOM. They also developed a logic to capture structural properties of DOM trees and they used it to verify properties of Minimal DOM programs by means of Hoare triples. Though the paper does not mention security applications, later work used similar formalizations of the DOM to track information flows and protect sensitive data.

Security Applications: Russo *et al.* presented a number of DOM-based attacks, where DOM navigation and updates are abused by a malicious script to leak confidential information to the attacker [77]. They advocated the usage of information flow control as an effective tool to thwart these attacks and they developed a sound runtime monitor for an imperative language extended with primitives for manipulating DOM-like trees. More recently, Almeida Matos *et al.* generalized the approach in [77] to include references and live collections, a special kind of data structure available in the DOM API that automatically reflects the changes occurring in a HTML document [60].

Rajani *et al.* also studied the interplay between DOM updates and information flow control [73]. Their work covers an even larger fraction of the DOM specification and provides a better treatment of live collections, by dropping the programmer-provided annotations (information flow labels) required in [60]. The authors released their DOM model as an Ocaml program, which is of independent interest and may be reused by other security researchers. They also implemented their information flow monitor in WebKit, a popular opensource web browser engine.

6. The Server View

6.1. RL1: Security by Construction

Complex web applications need to interface and integrate many different technologies, e.g., JavaScript code running in the user browser, server-side code implementing the application logic, and a database back-end hosting the application data. This integration process is error-prone and it often introduces security vulnerabilities. To overcome these problems, researchers proposed *multi-tier* programming languages, i.e., high-level languages for web applications which transparently handle the complexity of integrating different technologies by providing a unified layer of abstraction. These languages typically offer strong static typing guarantees, which ensure that many security issues like code injection attacks are prevented by construction; examples of multi-tier languages include Links [29], Hop [20], Ur/Web [28] and ML5 [64]. Security-enhancements for existing multi-tier languages have also been proposed in the literature and we focus on them in this section [30, 6, 27, 79].

6.1.1. SELinks [30]

Corcoran *et al.* proposed SELinks [30], a security-enhanced variant of the Links [29] programming language. Similarly to Links, SELinks provides a uniform server-database programming model, but it also allows programmers to define security meta-data attached to data types, called *labels*, and specify enforcement policy functions that mediate access to labelled data. To ensure that calls to enforcement functions are never forgotten by programmers, SELinks makes values with a labelled type opaque to programs: to use a labelled object, a program is forced to pass it to an enforcement policy function, which performs a label-based security check and strips the label from the object type, in case the check was successful.

Label stripping is performed using a special language construct, `unlabel`, which is confined to enforcement policy functions identified by the `policy` keyword. This allows one to conveniently identify the functions that must be trusted to perform security enforcement, greatly simplifying code review. It is also possible to certify the correct semantics of these functions by using the dependent types available in SELinks. The SELinks language was used to develop two sizeable web applications: SEWiki, a security-oriented blog/wiki, and SESpine, a secure online medical health record management system.

6.1.2. Secure Compilation of Links [6]

Baltopoulos and Gordon observed that one cannot reason on the security of Links programs just by focusing on their abstract, high-level programming model [6]. This worsens the effectiveness of the multi-tier abstraction, since security reviews may not be conducted just by inspecting the Links source code alone. In particular, the authors noticed that Links places too much trust in the browser tier, since it models session state by embedding continuations in HTML pages. Malicious clients can then learn secret data stored in a continuation or

compromise the intended control flow of the web application just by tampering with the HTML page.

The authors proposed to solve this problem by applying authenticated encryption to continuations and provided a formal security proof of their design. Specifically, they introduced TinyLinks, a calculus modelling a core fragment of Links, and they developed a type-and-effect system for it, to statically check correctness properties (e.g., data integrity) based on assertions included in the TinyLinks code. They then proposed a secure translation from TinyLinks into F7 [11], a functional language similar to F# extended with dependent types and amenable for type-based security verification. This translation formalizes in F7 the proposed extension of Links with authenticated encryption. The main formal result in [6] states that well-typed TinyLinks programs are compiled into well-typed F7 programs preserving the code-level assertions, which ensures by type safety that no assertion may ever be falsified in the revised Links design as implemented in F7, despite the best efforts of a malicious client tampering with the HTML pages of the Links program.

6.1.3. UrFlow [27]

Chlipala developed UrFlow [27], an extension of the multi-tier programming language Ur/Web [28] with support for enforcing access control and information flow policies. Since Ur/Web extends a standard functional language with native support for SQL queries, UrFlow advocates the usage of SQL as a natural way to express the desired security policies for the Ur/Web application. For instance, SQL queries can express confidentiality properties by explicitly selecting which information may be disclosed to users; the entitled users are then identified by a predicate `known` embedded in the query syntax, restricting disclosure only to those users who are aware of some information specified in the predicate, e.g., a password.

UrFlow statically verifies that such a kind of policy queries are respected by the Ur/Web application by resorting to symbolic execution, a form of abstract interpretation where unknown input values are modelled symbolically. If the verification fails, UrFlow returns a first-order logic characterization of a program state which may violate the security query. UrFlow was tested on a small set of Ur/Web applications, showing good performance.

6.1.4. SeLINQ [79]

Schoepe *et al.* proposed SeLINQ, a framework to enforce information flow properties preserved across the boundaries between an F# application and a SQL database, thus ensuring end-to-end security [79]. The framework assumes the adoption of LINQ [63], a technology adding native query support to .NET languages, including F#. Since LINQ extends F# with the addition of query expressions, it is possible to revise standard information flow type systems for functional languages to uniformly deal also with database queries.

The authors devised a security type system for a core functional language including LINQ-style constructs for database accesses and proved that their type system enforces non-interference. The paper also describes an implementation

of the type-checker and a translator from the typed core language considered by the authors into executable F# code. The framework was applied to develop a realistic web application interfacing with a movie rental database.

Recently, SELinq evolved into JSLINQ [5]. This extended framework covers the entire end-to-end loop of the web application, not only looking at the server and its SQL database, but also at the browser, in particular by securely generating JavaScript and HTML. The paper presents several case studies assessing the practicality of the framework.

6.2. RL2: Modelling, Verification and Enforcement

While the client-side part of a modern web application is typically developed in JavaScript, a plethora of programming languages can be used to write the server-side logic. General-purposes programming languages like C or Java can be fruitfully applied to the task and program verification techniques for these languages can thus be used for web security. However, given the scope of the present survey, we only focus on research efforts aimed at formalizing and reasoning about *scripting languages* traditionally associated to the Web. Of these languages, only PHP [36] and Python [72] have a formal semantics as of now. Besides these language-specific studies, it is also worth mentioning a few research works which abstract from specific web technologies and rather focus on security problems which affect the large majority of server-side scripting languages, most notably code injection attacks [81, 15, 74].

6.2.1. KPHP: A Formal Semantics for PHP [36]

KPHP is an operational semantics for a substantial core of PHP, defined by Filaretti and Maffeis [36] and mechanised using the popular K framework [76] for expressing programming language semantics. It is a huge formal semantics, providing a very faithful representation of the language it models, and it spans around 8500 lines of code. KPHP was validated by testing it against the official test suite distributed with the Zend engine, the reference implementation of PHP. Though there is still significant room for improvement, especially in the number of supported features, none of the failed tests was due to language constructs being modelled incorrectly by KPHP. The authors gave preliminary example applications of KPHP by model-checking a few expected invariants on some publicly available code snippets.

Security Applications: The original paper on KPHP [36] does not develop any security analysis based on the semantics, but it mentions provably sound static analyses based on abstract interpretation, type systems and taint-checking as an important avenue for future work. Indeed, one of the motivations behind KPHP was exactly the lack of sound support for particularly complicated PHP features in existing static analysers like Pixy [49] and WebSSARI [46].

6.2.2. λ_π: Taming Python by Desugaring [72]

Politz *et al.* proposed to capture all the subtleties of Python by means of a compilation into a simpler core language [72]. The idea behind the approach is

the same of λ-JS and S5, presented in Section 4.2.2: the authors first define the core language λ_π, which is roughly a traditional stateful λ-calculus, and then give the semantics of Python constructs by compiling them into λ_π. Since the semantics of λ_π is standard and well-understood, it is more amenable for formal reasoning than Python. The compilation (desugaring) was defined for a significant core of Python and its effectiveness was established by extensive testing against the standard CPython implementation of the programming language.

Security Applications: None we are aware of at the time of writing.

6.2.3. Defining Code Injection Attacks

The first formal perspective on code injection attacks is due to Su and Wassermann [81]. Their paper presented a definition of code injection attacks in the context of SQL-based web applications: roughly, their definition detects an injection attack on a web application if an input string provided by the user changes the expected syntactic structure (parse tree) of a query performed by the web application. Later work by Bisht *et al.* espoused the main intuition underlying the definition of code injection in [81], but it presented a different proposal which generalizes to more realistic programs [15]. The idea behind the generalisation is to map each possible input v to a corresponding *benign* input $IR(v)$, exercising the same program branches of the web application; then, rather than having a fixed expected syntactic structure for each query, which is restrictive, one can associate different query structures to different program branches, and detect a code injection if the query structure generated from an input v differs from the query structure generated from the benign input $IR(v)$.

Unfortunately, this definition is flawed, since it contains a circularity spotted by Ray and Ligatti [74]. The problem is that attacks are defined by means of the IR function, which in turn requires one to identify benign inputs (which are not attacks). Ray and Ligatti also observed that the idea of checking alterations of the expected syntactic structure of a SQL query is not good enough, since it may both miss attacks and incorrectly rule out legitimate queries. They thus proposed an alternative definition, based on a formal partitioning of inputs into code and non-code: an injection attack is then detected if at least one input entering a query is used as code. Remarkably, parsing is necessary to determine whether an input is code or not: for example, an integer literal is code when used to specify the size of an array, but it is non-code when used as an expression.

Security Applications: The authors of [81] proposed a sound and complete algorithm for detecting code injection attacks based on parsing and implemented it in SQLCHECK, a tool to detect code injection attacks in PHP and JSP applications. The tool in [15], called CANDID, is based on a syntactic transformation of the source program, used to deduce at runtime the query structure intended by the programmer, and a SQL parse tree checker operating on the transformed program to detect any mismatch with respect to the inferred intentions. Luo *et al.* extended the compiler of the HOP multi-tier programming language to automatically prevent code injection attacks [53], following the technique proposed in [81]. Unlike [81], the expected syntactic structure of the dynamically

generated code is not provided by the programmer, but it is apparent from the syntax of the HOP program.

7. Formal Models for the Web Platform

All of the works described in this section belong to the research line RL2. They amount to formal models of the Web as a whole, typically including browsers, servers and communication protocols. These models are particularly well-suited to the security analysis of web protocols, for which all the aforementioned entities play a prominent role.

7.1. A First Formal Foundation for Web Security [2]

The first paper which proposed an abstract model of the web platform is due to Akhawe *et al.* [2]. Motivated by the observation that even security experts are likely to miss simple vulnerabilities of web security mechanisms, given the complexity of the web platform, the authors developed a mechanized model of the Web in Alloy [47]. Alloy is a declarative language based on a relational extension of first-order logic. An Alloy specification can be analysed by Alloy Analyser, a tool to automatically find satisfying models or counter-examples for given assertions, i.e., properties of interest which are expected to hold for the underlying specification. Though the bounded verification performed by Alloy Analyser cannot prove that a given assertion will always hold true for a model of unbounded size, like the one developed in [2], it is still very useful for finding counter-examples (security problems).

The authors included in their model a number of ingredients:

- *web concepts*: the core elements of a web browser, web servers and the most relevant aspects of the HTTP protocol;
- *threat models*: the capabilities available to web attackers and network attackers. Also, the model includes a formalization of the user behaviour, which is often important when reasoning about security;
- *security goals*: several web security invariants, modelling constraints which should be respected to avoid breaking the functionality of existing web applications, and a session integrity property, which ensures protection against CSRF attacks.

Security Applications: In the original paper [2], the model was applied to analyse five case studies and find violations of the security goals in all of them. Interestingly, later research work extended and reused the Alloy model to assess the design of novel web security mechanisms. Chen *et al.* applied the model to validate the effectiveness of App Isolation, an experimental web browser providing stronger isolation guarantees for web applications [26]. De Ryck *et al.*, instead, used the model to formally evaluate the design of CsFire, a browser-side protection mechanism against CSRF attacks [31].

7.2. WebSpi: Using ProVerif for Web Security [8, 7]

WebSpi is a library developed by Bansal *et al.* [8, 7], defining the basic ingredients (users, browsers, web servers, etc.) needed to model web applications, web protocols and their security properties in ProVerif. ProVerif [16] is an automatic state-of-the-art protocol verification tool: protocols are specified using a dialect of the applied pi-calculus [1] and then analysed using an abstract representation of their possible executions in presence of an active adversary (based on Horn clauses). ProVerif is a fairly mature tool nowadays and it can be used to automatically check both secrecy and authenticity properties of security protocols. It can prove security for an unbounded number of protocol executions, though the analysis it implements is not guaranteed to terminate in general.

Since the threat models considered in the web security literature typically depart from the standard Dolev-Yao model used for the analysis of security protocols and implemented in ProVerif, WebSpi also defines a web attacker model in terms of an applied pi-calculus process and provides facilities for fine-tuning the security analysis by enabling or disabling different classes of attacks, e.g., code injection attacks against trusted web applications.

Security Applications: WebSpi was used by its authors to perform a security analysis of the OAuth 2.0 authorization protocol, discovering many previously unknown vulnerabilities in major websites such as Yahoo! and WordPress when they connect to social networks such as Twitter and Facebook [8]. More recently, WebSpi was employed by the same research group to analyse the security of several cloud-based storage services, including popular services like Dropbox, SpiderOak and 1Password [7].

7.3. An Expressive Model for the Web [34]

The most expressive model for the Web to date has been proposed by Fett *et al.* [34]. It is based on a generic process algebra in which processes have addresses and messages are modelled as first-order terms with equational theories defining the behaviour of cryptographic primitives. Though abstract enough to allow for formal reasoning, this pen-and-paper model tries to follow as closely as possible the existing web standards and it spans several pages of the technical report accompanying the original paper. The model includes:

- *basic web concepts*: including web servers, web browsers, DNS servers, HTTP(S) requests and responses, and several HTTP(S) headers;
- *threat models*: defining the capabilities of web attackers and network attackers in terms of the process algebra;
- *a detailed browser representation*: capturing in great detail the concepts of windows, documents, and iframes, as well as new technologies, such as web storage and cross-document messaging. JavaScript is not modelled in its entirety, but the core of a scripting language is included.

The model proposed in [34] is not directly amenable for automation, due to several features which are admittedly useful, but also challenging for automated

tools. For instance, the set of first-order terms (messages) is infinite and the treatment of state information in the model is non-monotonic, e.g., cookies can be deleted from the browser cookie jar and not only added.

Security Applications: The model in [34] was employed by the authors to analyse BrowserID, a complex single sign-on system by Mozilla allowing websites to delegate user authentication to email providers. The security analysis unveiled several attacks, including a critical one which allowed an attacker to hijack the sessions of any user owning a GMail or Yahoo! address. In the same paper, the model was also used to prove the security of a revised variant of the BrowserID system. The analysis of BrowserID was extended to deal with privacy aspects in later work by the same authors [35], revealing additional pitfalls suggesting the need for a major overhaul of the system.

7.4. Model-Checking Web Protocols

We anticipated in Section 3 that formal methods for security protocols cannot be directly applied to web protocols without the risk of missing attacks which are specific to the web platform [41]. Nevertheless, automated tools for protocol verification can be successfully applied for finding attacks against web protocols [4, 3, 86, 87]. These tools are quite appealing to mechanise the attack finding process, because they are relatively easy to use and web protocols often target (at least) a few secrecy and authenticity properties common to standard security protocols.

Notable examples in this area focused on the formal analysis of identity federation protocols such as SAML [66] and OpenID [67]. Armando *et al.* studied the security of the SAML protocol, as well as of a simplified variant of the protocol implemented by Google [4]. They performed a mechanised security analysis of the simplified protocol using the SATMC model-checker, which found an attack breaking the intended authentication goals. Remarkably, even though the original SAML protocol was deemed secure against the identified attack, follow-up work by the same research group used a more refined formal model to find an authentication flaw also in the SAML specification [3]. The same problem also affected the OpenID protocol. Other relevant work by Tobarra *et al.* [86, 87] is in the area of web service security. They used model-checking techniques to find attacks against the WS-Security protocol [89].

8. Perspective and Recommendations

We reviewed many applications of formal methods to web security, including several success stories. We discuss here the most important ingredients and directions we think future researchers in the area should keep in mind to design practical and useful proposals, fostering a large scale adoption of web security solutions backed-up by formal verification.

8.1. Designing the Right Model

In this survey, we mentioned many formal models of existing web technologies, including operational semantics for JavaScript (Section 4.2), browser models (Section 5.2) and models of the web platform (Section 7). Based on existing work, we identify two main desiderata for future proposals:

1. *comprehensiveness*: understanding what is relevant for security and what is not is extremely hard for the Web, given its size and complexity. In particular, while models are useful for attack finding even when incomplete, security proofs of defensive mechanisms need a good understanding and a faithful representation of the entire web platform to be trusted. Formal models should thus provide very accurate descriptions of the web technologies they target: apparently irrelevant aspects should be abstracted by the adoption of suitable formal techniques, e.g., abstract interpretation, rather than just dropped altogether from the model;
2. *mechanization and tool support*: pen-and-papers models are certainly useful as a starting point for the formal analysis of web security mechanisms, but they should eventually be replaced by mechanized models expressed in some programming language, for at least two reasons. First, web standards are dynamic and subject to frequent changes, and it is important to keep formal models in sync with them; structuring these models in a programming language simplifies maintainability. Second, given the size of accurate web models, tool support is crucial to give trust in the correctness of formal proofs and helpful to automatically find attacks.

At the time of writing, it seems these two requirements are particularly followed by researchers working on JavaScript semantics. Browser models and models of the web platform, instead, typically sacrifice at least one of these two points.

8.2. Modular Reasoning

The web platform includes different interacting components, such as browsers and servers, each of which can be further split into smaller sub-components. For instance, browsers include a JavaScript engine, an HTML parser and a cookie jar; servers, instead, include databases and web pages developed using different programming languages. Unfortunately, end-to-end security guarantees for the web platform often require all the sub-components to behave correctly and each sub-component is already a very complex system to verify on its own. Also, end-to-end security requires the presence of secure communication channels between secure communicating components.

It is unrealistic to believe that only one formal model can fully accommodate all the complexity of the web platform and its interactions, while being tractable for formal proofs. Indeed, security researchers have split their efforts along the many research lines discussed in this work to get an in-depth understanding of individual web components and tackle their problems in isolation. Combining these efforts is a major challenge and requires *modular reasoning* techniques. Modular reasoning is common in language-based security, most notably security

type systems, and it allows one to prove the security of a system by building security proofs of its sub-components. Bringing modular reasoning to the Web is an important research direction.

8.3. Proposing the Right Abstractions

There is often a tension between what security researchers propose and what security practitioners desire. For instance, a fully abstract compiler from the ML-like language F^* to JavaScript (Section 4.1.1) is definitely an interesting idea and an impressive research effort. However, follow-up work by the same research group recognized that "this is an attractive design, but the sad truth is that millions of JavaScript programmers are unlikely to switch to ML" [83]. The follow-up paper thus proposes the TS^* language (Section 4.1.3), featuring a programming paradigm much nearer to the customs of JavaScript users.

In general, we observe that even proposals which embrace a "security by construction" approach should be very careful in their design. It is true that these proposals are ultimately targeted at the development of new web applications and technologies, without being necessarily easy to retrofit on the existing Web. However, as a matter of fact, one can change a programming pattern, but not the technological background and the mindset of millions of web developers. To enable a large scale adoption, novel abstractions for secure web programming should speak the same language of web developers (and browser vendors, if any change to current web browsers was needed for their deployment).

8.4. The Importance of Compatibility

Given the massive user base of the Web and the existence of millions of websites, backward compatibility may even be more important than protection (soundness) for new web security solutions. This may be discouraging at first for researchers interested in proposing provably sound web defenses, since it partially limits the design space for clear-cut and innovative solutions tackling the root cause of a security vulnerability. However, the quest for backward compatibility is inherently part of web security and it poses interesting formal problems on its own rights. Indeed, although backward compatibility is typically assessed by testing new proposals against the real Web, we argue that also formal models can be useful to evaluate this important aspect. The point is that, like any form of testing, practical evaluations on the Web are inherently limited and they are better complemented by formal analysis.

For instance, web models can be used to identify sufficient conditions under which a sound enforcement mechanism does not alter the semantics of a web application. Notice that different definitions of "alter" may be plausible here and possibly adapted from other research areas, for instance they could be based on existing observational equivalences defined for process algebras. A good example of a provably sound defense mechanism with a formal compatibility result is the secure multi-execution approach to non-interference (Section 5.2.3). This technique enjoys a *transparency* result, stating that the semantics of already secure programs (web applications) is not affected by secure multi-execution [33].

9. Conclusion

We provided a comprehensive review of research work proposing formal methods as an effective tool to design and validate web security solutions. We observed that formal methods have been successfully applied to many different areas of web security: JavaScript security, browser security, web application security, and web protocol analysis. We discussed the most important challenges which must be tackled when approaching web security from the formal methods perspective and we identified recommendations for researchers interested in investigating this fascinating field.

All in all, we observe that formal methods for web security are quite a well-established reality as of now, but they still represent a very small fraction of the entire literature on web security. Given the critical importance of the web platform in everyone's life, we hope that future web security solutions will only be considered adequate after a careful formal verification. Though this may seem overly ambitious and wishful, we observe that something similar already happened in the protocol community: in that area, formal methods proved extremely effective at detecting attacks even against carefully engineered protocols proposed by security experts and published at prestigious venues [52, 88]. As a result of all these attacks, we are not aware of new protocols published at major security conferences in the last few years without a rigorous security analysis.

Acknowledgements. We would like to thank the anonymous reviewers for their careful reading and precise comments. We would also like to thank Sergio Maffeis, Frank Piessens and Andrei Sabelfeld for their valuable feedback on an early draft of this work.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115, 2001.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 290–304, 2010.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An authentication flaw in browser-based single sign-on protocols: Impact and remediations. *Computers & Security*, 33:41–58, 2013.

[4] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA, October 27, 2008*, pages 1–10, 2008.

[5] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld. JSLINQ: building secure applications across tiers. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 307–318, 2016.

[6] I. G. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 27–38, 2009.

[7] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In *Principles of Security and Trust - Second International Conference, POST 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 126–146, 2013.

[8] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 247–262, 2012.

[9] A. Barth. The web origin concept. http://tools.ietf.org/html/rfc6454, 2011.

[10] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in chromium. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[11] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8, 2011.

[12] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 653–670, 2013.

[13] N. Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.*, 82(8):243–262, 2013.

[14] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive noninterference for a browser model. In *5th International Conference on Network and System Security, NSS 2011, Milan, Italy, September 6-8, 2011*, pages 97–104, 2011.

[15] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2), 2010.

[16] B. Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, pages 54–87, 2013.

[17] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014.

[18] A. Bohannon. *Foundations of webscript security*. PhD thesis, University of Pennsylvania, 2012.

[19] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010.

[20] G. Boudol, Z. Luo, T. Rezk, and M. Serrano. Reasoning about web applications: An operational semantics for HOP. *ACM Trans. Program. Lang. Syst.*, 34(2):10, 2012.

[21] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[22] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta. Provably sound browser-based enforcement of web session integrity. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 366–380, 2014.

[23] S. Calzavara, M. Bugliesi, S. Crafa, and E. Steffinlongo. Fine-grained detection of privilege escalation attacks on browser extensions. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 510–534, 2015.

[24] S. Calzavara, G. Tolomei, A. Casini, M. Bugliesi, and S. Orlando. A supervised learning approach to protect client authentication on the web. *TWEB*, 9(3):15, 2015.

[25] S. Cantor and M. Erdos. Shibboleth specification, 2015. Available at https://shibboleth.net/.

[26] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 227–238, 2011.

[27] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 105–118, 2010.

[28] A. Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 153–165, 2015.

[29] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, pages 266–296, 2006.

[30] B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 269–282, 2009.

[31] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pages 100–116, 2011.

[32] D. Devriese, L. Birkedal, and F. Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 147–162, 2016.

[33] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 109–124, 2010.

[34] D. Fett, R. Küsters, and G. Schmitz. An expressive model for the web infrastructure: Definition and application to the BrowserID SSO system. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 673–688, 2014.

[35] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, pages 43–65, 2015.

[36] D. Filaretti and S. Maffeis. An executable formal semantics of PHP. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 567–592, 2014.

[37] C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 371–384, 2013.

[38] P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. DOM: towards a formal specification. In *PLAN-X 2008, Programming Language Technologies for XML, An ACM SIGPLAN Workshop colocated with POPL 2008, San Francisco, California, USA, January 9, 2008*, 2008.

[39] C. Grier, S. Tang, and S. T. King. Designing and implementing the OP and OP2 web browsers. *TWEB*, 5(2):11, 2011.

[40] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 748–759, 2012.

[41] T. Groß, B. Pfitzmann, and A. Sadeghi. Browser model for security analysis of browser-based protocols. In *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, pages 489–508, 2005.

[42] T. Groß, B. Pfitzmann, and A. Sadeghi. Proving a ws-federation passive requestor profile with a browser model. In *Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005, Fairfax, VA, USA, November 11, 2005*, pages 54–64, 2005.

[43] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 115–130, 2011.

[44] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 126–150, 2010.

[45] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 3–18, 2012.

[46] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 40–52, 2004.

[47] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[48] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 113–128, 2012.

[49] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 258–263, 2006.

[50] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 57–74, 2013.

[51] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: retrofitting type systems for JavaScript. In *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 1–16, 2013.

[52] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.

[53] Z. Luo, T. Rezk, and M. Serrano. Automated code injection prevention for web applications. In *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, pages 186–204, 2011.

[54] Z. Luo, J. F. Santos, A. A. Matos, and T. Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. *Journal of Computer Security*, 24(1):91–136, 2016.

[55] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151, 1987.

[56] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, pages 307–325, 2008.

[57] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 505–522, 2009.

[58] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 125–140, 2010.

[59] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91, 2009.

[60] A. G. A. Matos, J. F. Santos, and T. Rezk. An information flow monitor for a core of DOM - introducing references and live primitives. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, pages 1–16, 2014.

[61] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[62] J. Meseguer, R. Sasse, H. J. Wang, and Y. Wang. A systematic approach to uncover security flaws in GUI logic. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 71–85, 2007.

[63] Microsoft. LINQ (language-integrated query), 2015. Available at https://msdn.microsoft.com/it-it/library/bb397926.aspx.

[64] T. Murphy VII, K. Crary, and R. Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, pages 108–123, 2007.

[65] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 736–747, 2012.

[66] OASIS. SAML specification, 2005. Available at http://saml.xml.org/saml-specifications.

[67] OpenID Working Groups. OpenID specification, 2014. Available at http://openid.net/developers/specs/.

[68] D. Park, A. Stefanescu, and G. Rosu. KJS: a complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 346–356, 2015.

[69] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, pages 47–60, 2009.

[70] 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 *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, pages 1–16, 2012.

[71] J. G. Politz, A. Guha, and S. Krishnamurthi. Typed-based verification of web sandboxes. *Journal of Computer Security*, 22(4):511–565, 2014.

[72] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 217–232, 2013.

[73] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 366–379, 2015.

[74] D. Ray and J. Ligatti. Defining code-injection attacks. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 179–190, 2012.

[75] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 52–78, 2011.

[76] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[77] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 86–103, 2009.

[78] R. Sasse, S. T. King, J. Meseguer, and S. Tang. IBOS: A correct-by-construction modular browser. In *Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers*, pages 224–241, 2012.

[79] D. Schoepe, D. Hedin, and A. Sabelfeld. SeLINQ: tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 25–38, 2014.

[80] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 463–478, 2010.

[81] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 372–382, 2006.

[82] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

[83] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman. Gradual typing embedded securely in JavaScript. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 425–438, 2014.

[84] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 363–378, 2011.

[85] The World Wide Web Consortium. HTML5 specification, 2015. Available at http://www.w3.org/TR/html5/.

[86] M. L. Tobarra, D. Cazorla, F. Cuartero, and G. Díaz. Application of formal methods to the analysis of web services security. In *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, pages 215–229, 2005.

[87] M. L. Tobarra, D. Cazorla, F. Cuartero, and G. Díaz. Analysis of web services secure conversation with formal methods. In *International Conference on Internet and Web Applications and Services (ICIW 2007), May 13-19, 2007, Le Morne, Mauritius*, page 27, 2007.

[88] L. Viganò. Automated security protocol analysis with the AVISPA tool. *Electr. Notes Theor. Comput. Sci.*, 155:61–86, 2006.

[89] WSS Technical Committee. WS-Security specification, 2006. Available at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[90] S. Yoshihama, T. Tateishi, N. Tabuchi, and T. Matsumoto. Information-flow-based access control for web browsers. *IEICE Transactions*, 92-D(5):836–850, 2009.

[91] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 237–249, 2007.

[92] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies lack integrity: Real-world implications. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 707–721, 2015.