Accountable JavaScript Code Delivery

Ilkan Esiyok*, Pascal Berrang†, Katriel Cohn-Gordon‡, Robert Künnemann*
*CISPA Helmholtz Center for Information Security {ilkan.esiyok, robert.kuennemann}@cispa.de
†University of Birmingham and Nimiq {p.p.berrang@bham.ac.uk}
‡Meta {me@katriel.co.uk}

***Abstract*—The Internet is a major distribution platform for web applications, but there are no effective transparency and audit mechanisms in place for the web. Due to the ephemeral nature of web applications, a client visiting a website has no guarantee that the code it receives today is the same as yesterday, or the same as other visitors receive. Despite advances in web security, it is thus challenging to audit web applications before they are rendered in the browser. We propose Accountable JS, a browser extension and opt-in protocol for accountable delivery of active content on a web page. We prototype our protocol, formally model its security properties with the TAMARIN Prover, and evaluate its compatibility and performance impact with case studies including WhatsApp Web, AdSense and Nimiq.**

Accountability is beginning to be deployed at scale, with Meta's recent announcement of Code Verify available to all 2 billion WhatsApp users, but there has been little formal analysis of such protocols. We formally model Code Verify using the TAMARIN Prover and compare its properties to our Accountable JS protocol. We also compare Code Verify's and Accountable JS extension's performance impacts on WhatsApp Web.

I. INTRODUCTION

Over the years, the web has transformed from an information system into a decentralised software distribution platform. Websites are programs that are freshly fetched whenever accessed and the web browsers are runtime environments. This design implies that when a user opens a website, they have no reason to trust it will run the same program that it did yesterday or the same program that other users receive. Instead, the application loaded may vary over time, and different users may receive different codes.

The majority of web pages, and even web applications, have neither specified security goals nor the need to establish them. Nevertheless, for some websites, maintaining trust between developers and users is part of the business model:

- a private email provider might wish to reassure users that it will always encrypt their messages,
- a cryptocurrency wallet might wish to guarantee that it has no access to users' funds, or
- a tracking pixel might wish to prove that it only receives data that is explicitly sent to it.

Some academic proposals for secure protocols implemented for browsers include TrollThrottle [1] and JavaScript Zero [2]; industry proposals include payment platforms such as Stripe and Square, chat protocols such as WhatsApp Web, Facebook Messenger and Matrix's Hydrogen client, encrypted cloud storage such as MEGA or SpiderOak. A concrete example is Nimiq, an entirely web-based digital currency managing private keys in the browser. It is challenging for such websites to make verifiable guarantees to their users: a compromised or malicious web server can precisely target classes of users: the email provider might disable encryption on a specific IP range, the cryptocurrency wallet might redirect payments made in some countries, or the tracking pixel might exfiltrate data only for certain users.

Auditing: A common risk mitigation strategy is *auditing*: a developer who wishes to build trust appoints external auditors to inspect the client code. This can include both vulnerability research (e.g. via bug bounties) or commissioned security audits. Audits work well where it is possible for a user to verify that the code they are running is the same code that was audited, for example when binaries are received via third party package repositories or app stores that control the distribution and targeting. App stores do not usually permit developers to deliver different codes to different users for the same app, except in a restricted set of circumstances such as for beta testing new features.

However, auditing does not work for web applications: a compromised or malicious web server can simply choose at load time to deliver unaudited code to a user. No matter how careful the audit or even verification of the web application, users cannot know that they are receiving the audited code. Large parts of modern web security thus depend on techniques like sandboxing or access control to critical resources like cameras, but fail to capture properties defined in the context of the application (e.g. authorisation of transactions in a payment system).

Accountability: A second risk mitigation strategy is *accountability*, where developers can be held accountable for applications which they publish. In curated software repositories such as Debian GNU/Linux or the Apple App Store, developers' code is reviewed and malicious or compromised code is linked to their identities. Developers who repeatedly publish malicious code may face consequences such as loss of user trust or banning from the repositories. For example, a package mirror which publishes malicious code may be removed from future lists of mirrors, or a developer who takes over a browser extension and publishes a malicious version [3] may be blocked from publishing future code updates.

Again, web applications fail to have accountability. A malicious or compromised web server may publish malicious code to certain users, but there is no public record of the code

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24096
www.ndss-symposium.org

which it serves, and thus no way for users to hold the server accountable.

Summarising, it is difficult to establish trust in the web as a software distribution mechanism because it lacks *auditability* (the means for anyone to inspect the code being distributed to others) and *accountability* (the means to hold a developer accountable for the code they publish).

In this paper, we propose an opt-in transparency protocol that aims to establish more rigorous trust relations between browsers and web applications, and provide the foundation for a more secure web. Using our standard for accountable delivery of active content, efficient and easy-to-use code-signing technique, and public transparency logs; websites can convince the users that they are trustworthy in an economical way. At a high level, we propose that web application developers, who choose to opt-in, provide a signed manifest enumerating all the active content in their applications.

The manifest files in our proposal are stored in publicly readable transparency logs. When a browser requests a URL and downloads the resulting HTML document from the web server, the web server also provides the corresponding manifest for this URL. The browser checks that the active content provided by the server matches the manifest entry, that the manifest is correctly signed, and that the provided manifest is consistent with the transparency logs.

Moreover, our proposal aims to reinforce the communication between the browser and the web server by adding non-repudiation to the HTTP request-response procedure. By itself, Transport Layer Security (TLS) does not provide evidence that what was delivered actually originated from the web server. Using digital signatures, we show how HTTP requests can be extended to provide a proof of origin.

From the signed manifest, the transparency logs, and the non-repudiation mechanism, the protocol establishes that:

- The code a user executes is the same for the users of the plugin within a certain timeframe depending on the validity of the manifest and a new manifest is signed.
- On the client side, the code is bound to interact with third party code according to how the developer declared in the manifest. This includes the order of execution, the trust relation to third party code, and the use of sandboxing.
- If the code's execution is inconsistent with the manifest, the browser can provide a claim that can be verified by the public.

Our proposal can be implemented by changes in the server configuration only, without the need to modify the served web content (assuming that the web page already makes use of Subresource Integrity hashes) and without changes to the HTML standard.

To sum up, our contributions are as follows:

1) We propose Accountable JS, a protocol to enable auditability and accountability for web apps.
2) We formally model Accountable JS with the TAMARIN Prover and prove desired properties in the presence of active adversary.
3) We implement Accountable JS in a browser extension that obtains the signed manifest, verifies its signature, and both statically and dynamically ensures that the active content on a web page agrees with the manifest. We also provide a code-signing mechanism for the developers.
4) We evaluate the deployment of this technology and the performance overhead for the client in six case studies, including real-world applications: Google AdSense, Nimiq and WhatsApp.
5) We model Meta's Code Verify protocol and compare its properties with Accountable JS.

Relationship to Meta's Code Verify protocol: In [4], Meta (formerly Facebook) proposed Code Verify, likewise implementing a mechanism to enforce accountability via transparency for active content in the web. Our present proposal goes beyond Code Verify and provides a superset of its functionality, most notably the ability to delegate trust to third parties. On the other hand, our browser extension is an academic prototype and thus not ready for productive use. The protocol has the same message flow, but chooses a different signature scheme and encodings. We elaborate on these differences in Section X-A. An initial draft of the present proposal was shared with Meta's WhatsApp team in 2022. The protocol, manifest file format and browser extension we present in this work are academic developments by the authors and not endorsed by Meta in any way.

II. BACKGROUND

Web pages are delivered via HTTP or HTTPS. In the latter case, a secure and authenticated TLS channel tunnels the HTTP protocol. Typically, the initiator of the TLS connection, i.e. the web browser, is not authenticated[1], whereas the responder, i.e. the web server, is identified with their public key and a certificate linking the public key to the domain.

The authentication guarantees of TLS exclude non-repudiation of origin, i.e. a communication party cannot prove to a third party that they received a certain message. This property is an important building block for accountability and can be achieved, e.g. using digital signatures. After the shared keys are established in TLS, any messages exchanged could be produced by either party. Roughly speaking, the party providing the evidence has enough information to forge it. Ritzdorf et al. [5] proposed a TLS extension that provides non-repudiation, but it has not been deployed in the wild.

Browsers typically parse the HTML document describing the web page into a tree of HTML elements called Document Object Model (DOM). Some HTML elements have *active content*, which includes Flash or Silverlight, but we will focus on JavaScript (JS) in this work. Active content can be *inline*, i.e. hard-coded in `<script>`-tags or event handlers, *external*, i.e. referring to an external JS file by URL, or *via iframe*, i.e. the web page contains an iframe that refers to an HTML file which, again, contains active content.

Like in the case of app stores, we distinguish the roles of the *website*, which is distributing the web application, and the *developer*, which is the author of the web application. This

[1] At the communication layer. Authentication may be implemented at the application layer.

allows us to view the website as a distribution mechanism that is necessarily online and publicly visible, as opposed to the developer, who can be offline most of the time. We distinguish the following roles:

- The web application developer (short: *developer*) creates the active content and has a secure connection to the web server. It is not active all the time.
- The web server (short: *server*) delivers code provided by the developer to the *client*. The website and the developer are associated with a domain, but the client is anonymous.
- The web browser (short: *client*) requests a URL from the website.

A transparency log (short: *ledger*) provides a publicly accessible database. It typically has the property of being append-only (for consistency), auditable, verifiable, and it hinders equivocation. Hence, for the data in the logs, all parties are convinced that it is a public record and that everyone sees the same version of it. We are using the ledger to store manifest files for each URL. Having public records of the manifest files allows us to reason about accountability.

A. Threat Model

Dolev-Yao attacker: We consider a Dolev-Yao style adversary, i.e. cryptography is assumed perfect (i.e. cryptographic operations do not leak any information unless their secret keys are exposed), but the attacker has full control over the network. This is formalised in our SAPiC [6] model in Appendix A. Informally, we assume hash function to behave like random oracles, signature schemes to be unforgeable and TLS to implement an authentic and confidential communication channel. We also rely on an intact public-key infrastructure.

Corruption scenarios: We assume honest parties follow the protocol specification and dishonest parties are controlled by the attacker. The parties which considered honest are determined by the property of interest:

- *Accountability and Authentication of Origin:* An honest client wants to be sure that code is executed only if it was made public and transparent i.e. inserted into logs by the developer; here developer and web server are assumed dishonest.
- *Non-repudiation of Reception* A dishonest client may want to present false evidence for having received some JS code. Here we assume the public is trusted and run a specified procedure[2] to check the evidence, and the web server to behave honestly, i.e. not to help the client provide false claims of reception, which are against the web server's interest.
- *Accountability of Latest Version* An honest client that receives a version of the code and wants to ensure it is the latest version. We assume an honest global clock that helps comparing the time of the code reception and the latest version at that time, and consider a dishonest developer and web server.

Target websites: We target developers that *aim at establishing user trust* or pretend to do so. Hence we assume, for honest developers, that active content changes infrequently, e.g. multiple times per day, and that their code facilitates the audit. Dishonest developers may counteract, but, due to accountability and authentication of origin, it is publicly recorded.

Therefore, while our formal security arguments make no assumption on how often the code changes are or how obfuscated it is, we assume that, from accountability of authentication of origin, code obfuscation attacks or microtargeting are practically disincentivised.

Browser features & Transparency log : We assume the current browser security features, specifically the `sandbox` attribute of the iframe tag, to be implemented correctly. Furthermore, the transparency log is trusted, efficient, available, append-only and provides non-equivocation (i.e. the same information is served to everyone). Many strategies are available to implement such a log. For example, Trillian [7] and CONIKS [8] use data structures that can be distributed over multiple parties and allow to prove append operations efficiently. Misbehaviour can thus be detected by trusted public auditors or by honest logs distributing such proofs (called *gossiping*). See [9] for a survey over different mechanisms.

III. Use Cases

We introduce several types of web applications that will benefit from our protocol. We will revisit these examples later and show how our approach can be applied to them.

A. Self-Contained Application

Perhaps the simplest possible web application is a one-page HTML document with active content that simply prints 'Hello World' into the developer console. Upon loading this website, a user can manually check that its sole behaviour was to print 'Hello World', but they have no guarantees about subsequent page loads: a server could easily decide to provide different behaviour to certain users, or to insert malware based on IP address or browser fingerprint. For this simple example, the consequences of a malicious or compromised server are relatively limited, although we remark that cryptojacking[3] is a growing trend [10].

WhatsApp Web is a large real-world self-contained web application: its source code is bundled using WebPack and served to all users; personalisation is implemented through local storage and dynamic data fetching. We will show how our protocol can be applied.

B. Trusted Third-Party Code

Many websites rely not just on their own content but on resources served by a third party. This may be a Content Delivery Network (CDN) serving common JS libraries, embedded content such as photos or videos, analytics and measurement libraries, tracking pixels, fraud detection libraries, or many other options. For example, the following code loads the jQuery JS library from a CDN, and uses it to display a 'Hello World' message.

[2]Detailed in Appendix B.

[3]Malicious JS which secretly mines cryptocurrencies in unsuspecting users' browsers.

```
<html><head>
  <script src="https://googleapis../jquery-3.6.1.min
  .js" integrity="sha384-i6..."/></head><body>
  <script>$("body").html("Hello World")</script>
</body></html>
```

Listing 1: Trusted third party code

As before, users are supposed to always receive the same code from the server. This time, there is an additional avenue for compromise, though: even if the first-party server is honest, it is possible for the CDN to perform targeted attacks. The developer, however, wants to pin the third party code to the precise version that they inspected or trust.

C. Delegate Trust to Third Parties

The application uses third party code that its developer cannot vouch for. This can be the case if the code is too complex to inspect or if the application developer wants to always use the latest version. The third party developer, however, is willing to vouch for their code. An example of this is Nimiq's Wallet, a web application for easy payment with Nimiq's crypto currency. This application can be embedded by first-party applications that provide, e.g. a web shop, who are willing to trust Nimiq, but only given that they make themselves accountable for the code they deliver.

```
<html><body>
 <script type="text/javascript">
  function addTransaction () {
    window.postMessage({'id': '123', 'amount': '10n
    ', 'from':'abc'}, 'https://wallet.nimiq.com/');}
 </script>
 <iframe src="https://wallet.nimiq.com/" onload="
 addTransaction()"></iframe>
</body></html>
```

Listing 2: Delegate trust to third party

D. Untrusted Third-Party Code

For web technologies, consecutive deployability is a must. Hence, in this use case, the application developer cannot audit the code, but the third party does not use Accountable JS. The application developer needs to blindly trust the third party, but using sandboxing techniques, it can restrict the access that the possibly malicious script provided by the third party can have.

A particularly important instance of this problem is ad bidding. The third party is an ad provider that decides online which ad is actually served. Because they cannot review the ads that they distribute, which may contain active content, they are not willing to vouch for the code they distribute. This is the case for Google AdSense, used by over 38.3 million websites. Cases where ads were misused to distribute malicious code are well documented [11].

E. Code Compartmentalisation

The application that the developer provides can be compartmentalised so that the most sensitive information is guarded by a component that is easy to review and changes rarely. The other components that are user-facing and changing more often are separated from this component using sandboxing. The developer wishes to reflect this structure and make themselves accountable for the whole code, but also separately commit on keeping the secure core component small and auditable. For example, Nimiq's Wallet components follow a similar structure.

IV. Approach: Accountable JS

We propose a cryptographic protocol between the client, the server, the developer, and a distributed network of public transparency logs. The protocol's objective is to hold the developer accountable for the code executed by the browser. The protocol provides four main functionalities:

- The server provides a manifest declaring the active content and trust relationships of the web application, which the client compares with a published version on the transparency logs.
- The client measures and compares the active content received by collecting active elements, e.g. JS, in the HTML document delivered by the web server.
- Developers and clients submit manifests to a public append-only log to verify that everybody receives the same active content.
- The server signs a nonce as non-repudiable proof of origin for the JS that the client receives.

Website Manifests: Website developers may provide a signed manifest for each publicly accessible URL in their website (excluding the query string). The signed manifest comprises a manifest and a signature block over it. A manifest describes the webpage, including, besides the active content, its URL and a version number. The active content is described in a custom format. We elaborate on the manifest directives in the supplementary material [12]. The developer's identity is distinct from the server's, but their certificates must share the same Common Name(CN) to restrain from unauthorised manifest deployments. The browser validates the authenticity of the developer's public key in the same way, using the existing Public Key Infrastucture (PKI) and its built-in root Certificate Authority (CA) certificates.

Accountable JS is an opt-in mechanism. The website declares the signed manifest using an experimental HTTP response header field called `x-acc-js-link`. Henceforth, the client, however, expects the website to provide a valid manifest for this URL in any case.

Client Measurement: The client measures the active content inside the HTML document delivered in the response body, collecting information about each active element in the document and validating it with the corresponding manifest block in a manifest file. Elements that cannot be matched trigger an error and the user is warned about this error. The current extension is not preventive, but in the future with pervasive developer support, browsers may choose to halt the execution if delivered code is inconsistent with the boundaries drawn by manifest. The active content is measured with a so-called *mutation observer*, starting with the first request. The measurement procedure that we developed listens to the observer's collected mutations that regard active elements in a list. In Section VII, we explain the process in more detail.

Manifest Logs: While a signed manifest may prove the integrity and authenticity of the manifest, it cannot prevent equivocation, i.e. it cannot prove the same signed manifest

is delivered to every request by the web server. To this end, we propose to use transparency logs. A manifest file declares a version number and the version number is unique per manifest file. The developer publishes their signed manifest in a publicly accessible, auditable, append-only log like the Certificate Transparency (CT) protocol [13], which provides logs for TLS certificates. Clients may verify that a version they receive is the latest online, or use a mechanism like Online Certificate Status Protocol (OCSP)-Stapling [14] to check that a version they receive was the latest version a short time ago. Any client that encounters a signed manifest that is not yet in the log can submit it to the log. We discuss the transparency log considerations in more detail in Section XI.

Non-Repudiation of Origin: We propose a simple non-repudiation mechanism for the client's web requests, so that in case a developer distributes damaging active content, a client can prove that they have received that content from a web server. The client transmits a nonce via a request header and the server signs this nonce along with the signed manifest (c.f. Section IX).

V. Manifest File

In the manifest, the developer declares the active elements a web application is bound to execute during its run time. The run time starts from the web request and ends with the window's close or a new web request. For Single Page Applications (SPA) (e.g. Nimiq), the run time for the web page ends when page is refreshed, its URL is changed or the window is closed.

The manifest file represents the active elements and their relevant metadata as a collection of attribute-value pairs in the JSON format. The metadata expresses the trust relations w.r.t. third party content and settings for sandboxing. The top-level properties in the manifest, also called manifest header, contain descriptive information about the web page: its URL, its version number, and optional metadata, e.g. the developer's email address. The domain within the URL determines which keys can be used to sign the manifest, namely, the common name of the signature key's certificate has to match that domain.[4] The developer can decide for any numbering scheme for the version, but they must be strictly increasing with each new manifest published.

A manifest file is accepted if it is *syntactically correct*, i.e. follows the schema (see manifest manual in the supplementary material [12] for details), *complete*, i.e. it contains enough information about the web application and its active elements to enable evaluation, and, most importantly, *consistent with the delivered resource*, i.e. that evaluation succeeds.

A. Execution Order

An active content is considered dynamic if it is added after the window's load event; otherwise, it is static. The manifest specifies elements as either static or dynamic using the *dynamic* attribute. SPAs in particular download or preload resources during navigation, rewriting the DOM on the fly depending on how the user navigates.

[4]The query component of the URL can be excluded, since the browser extension discards that part in the measurement.

TABLE I: Trust Relationships by Type of Active Element

	trust			
type	*assert*	*blind-trust*	*delegate*	*sandbox*
inline	●	○	○	○
event_handler	●	○	○	○
external	●	●	●	○
iframe with …				
src_type =external	●	●	●	●
src_type =srcdoc	●	○	○	●
src_type =script	●	○	○	●

For static elements, the sequence number *seq* specifies in which order they must appear after browser renders the delivered HTML. It starts from 0 and repetitions are not allowed. Dynamic content is only measured if they are present in the web page, i.e. it is allowed to be injected, but not required to. This mechanism can also be used to declare region-specific active content. The order is ignored for dynamic content. The measurement procedure will check if the list of the elements in the manifest is in the same order except for elements that will be dynamically added to the DOM. Elements may be removed dynamically, but only if the attribute *persistent* is set to false.

B. Trust and Delegation

With the manifest, the developer provides assurance for the active content in their application. Third-party components, e.g. JS libraries, bootstrappers, advertisements or ad-analytics tools play a significant role in most modern web applications, which are thus a mixture of first-party code and code from multiple third parties. In the manifest, we enable the developers to decide the trust level on each active element imported to their web applications. For instance, they can take the responsibility and provide assurance (i.e. with a cryptographic hash) on first party elements while for the external elements, they may declare a valid source and delegate the trust on the developers of those resources.

We thus require each block in the manifest to have a trust declaration. There are three options to declare the trust level:

- *assert* : The developer provides the hash of the expected active content and asserts it is behaving as intended. It is computed using the standard Subresource Integrity hash generation method [15], i.e. comprises the hash algorithm used, followed by a dash and the base64-encoded hash value.
- *delegate*: The developer refers the trust to the third party providing this element. Now the third party is taking responsibility for this code and provides a manifest whose location is either declared in the first-party manifest, or delivered in the headers of the third party's response. The third party manifest can likewise delegate trust, thereby constructing a chain of trust delegations.
- *blind-trust*: The developer blindly trusts the third party, without identifying the code they trust. This should only be used with the *sandbox* attribute.

C. Types of Active Elements

The developer describes the manifest blocks for each active element by their resource type *type* (e.g. *javascript*, *iframe*), trust policy *trust* (e.g. *assert*, *delegate*, *blind-trust*), whether

they are dynamic or static and, in case they are static, their sequence number *seq*. There are mandatory and optional directives for writing a manifest and these directives may depend on the resource type. If the developer declared a manifest section including an optional directive, that does not mean this directive is ignored in the evaluation; this directive still is part of the evaluation. For instance, the *crossorigin* directive is optional for *external* resource type, but if the developer declares a *crossorigin* attribute, then it has to match with the active content information. Not all resource types support all trust policies (see Table I). We will discuss them one by one:

- *inline*: Inline scripts are `script` elements without the *src* attribute, i.e. the JS code is included in the HTML document. Therefore, *trust* can only be *assert* and may be omitted. The cryptographic hash covers the included JS code, i.e. the textContent value of the script element.
- *event_handler*: Event handlers are active content included in attributes such as `onClick` that are executed on HTML events. Like inline scripts, *trust* must be *assert* and can be omitted.
- *external* : A `script` element can be outsourced by specifying its URL in the *src* attribute. An *external* script can originate from a different origin (cross-origin) or from the same origin. Trust can be set to *assert* and *delegate* – as sandboxing is not supported for external scripts, *blind-trust* would give little assurance.
- *iframe* : An *iframe* embeds another document within the current document. There are three ways this can happen, which the manifest file represents using the attribute *src_type*. The most common is to specify a URL (*src_type* = *external*). The other ways (*src_type* = *srcdoc* and *src_type* = *script*) are explained in the full version [16]. This type of content can be declared with any *trust* value.

D. Sandboxing

Besides, iframes permit the use of sandboxing via the attribute with the same name [17]. A sandboxed iframe is considered a cross-origin resource, even if its URL points to the same-origin website. Hence, because of the browser's same-origin-policy, the parent window and the iframe are isolated, and they cannot access the DOM of each other. Furthermore, sandboxing blocks the execution of JS and the submission of forms and more. These restrictions can, however, be lifted using an allow list in the HTML tag.

As we will see in the next section, security-critical websites need to use sandboxing to protect data from other browsing contexts; hence we reflect the *sandbox* feature in the manifest file. The measurement procedure ensures that the active element has an equally strict or stricter sandboxing policy than described in the manifest. An allow list is stricter if it is a subset of the other.

VI. Use Cases, Revisited

We come back to the use cases from Section III to illustrate how Accountable JS applies to real-world web applications with different trust assumptions.

A. 'Hello World' Application

We begin with the basic 'Hello World' website example, and add a reference to the manifest in its meta tags.

```
<html><head>
  <meta charset="utf-8" name="x-acc-js-link"
  content="http://www.helloworld.com/manifest.sxg">
</head><body>
  <script>console.log("Hello World")</script>
</body></html>
```

Listing 3: First example: Hello World.

Alternatively, the manifest can be provided as an HTTP response header. The manifest file provides the URL and version of the website and lists the base64-encoded SHA-256 hash of the inline script.

```
{ "url": "http://www.helloworld.com/",
  "manifest_version": "v0",
  "contents": [
    { "seq": 0,
      "type": "inline",
      "load": "sync",
      "trust": "assert",
      "hash": "sha256-AfuyZ600rk..."}]}
```

Listing 4: Manifest for first example.

B. Self-Contained Web Applications

Web applications can be completely self-contained. This may be for security or because they follow the recent serverless computing paradigm (e.g. Amazon Lambda). In serverless computing, a web application developer may only write static user-side code and delegate all the server-side logic to a cloud service provider.

The application of Accountable JS is straightforward in this case: as part of our prototype, we developed our deployment tool `generate_manifest`, which computes the hash values of all active contents in the browser and produces a manifest file that asserts their trustworthiness. The developer can then sign this manifest file.

We tested this methodology on a popular example, the WhatsApp Web client, and provide the manifest file in the supplementary material [12]. It lists nine external and four inline scripts.

C. Trusted Third-Party Code

The developer can use the manifest file to identify the included third party code by hash and set the order of execution. This expresses that the developer vouches for the third party code. We add the following attribute to the header of 'Hello World' example from Section III-B and we declare it in the manifest file with *trust* = *assert*.

```
<script src="https://googleapis../jquery-3.6.1.min
.js" integrity="sha384-i6..."></script>
```

D. Delegate Trust to Third Parties

The first party can delegate trust to a third party by embedding their code in an iframe (or linking their JS) and setting *trust* to *delegate*. The extension will verify the third party code based on a manifest file signed by its developer.

This expresses that the main developer vouches for the third party to be trustworthy, but demands that the third party itself can be held to account. This is in contrast with trusting a concrete piece of code provided by the third party.

We tested this technique using Nimiq's Wallet, which can be embedded in third party web pages. These can now combine the code that they control (e.g. for setting up a shopping cart) with the code that Nimiq provides for signing transactions.

The website's manifest below (Listing 5) specifies some inline scripts with $trust = assert$ (omitted) and an iframe with $trust = delegate$. The browser now expects the response to the query for the iframe's content (https://wallet.nimiq.com) to point to a URL with a signed manifest.

```
{ "url": "https://www.example-shop.com/",
  "manifest_version": "v2",
  "contents": [
    [inline script manifests omitted]
    { "seq": 2,
      "type": "iframe",
      "src_type": "link",
      "src": "https://wallet.nimiq.com/",
      "sandbox": "allow-scripts",
      "dynamic": false,
      "trust": "delegate" }]}
```

Listing 5: Manifest is delegated to a trusted third party

E. Untrusted Third-Party Code

High-security applications may want to rely on third party code they cannot vouch for, e.g. when including ads that are dynamically chosen by an ad-bidding process. We developed a small web application that uses Google AdSense and sandboxed this code, but noticed that AdSense and many other ad providers require access to the top-level window [18] for fraud detection, e.g. to detect invalid clicks.

We therefore needed to turn the relationship between the secure code and the untrusted code around. We sandboxed the secure code with *trust* set to *assert*, protecting it from the potentially unsecure AdSense code, which is not sandboxed and declared *blind-trust*. Now the AdSense code cannot access the secure document in the iframe. The manifest file is shown in List. 6. It includes thirteen active elements (six *external*, seven *iframe*) related to AdSense, along with Nimiq's Wallet (seq='6'), for which trust is delegated.

```
{"url": "https://www.helloworld.com/",
  "manifest_version": "v3",
  "contents": [
    [six external scripts for AdSense with trust=
    blindtrust]
    { "seq": 6,
      "type": "iframe",
      "src_type": "link",
      "src": "https://wallet.nimiq.com/",
      "sandbox": "allow-same-origin allow-scripts",
      "dynamic": false,
      "trust": "delegate" // See Listing 7
      },
    [six more iframes for AdSense with blindtrust]]}
```

Listing 6: Untrusted AdSense and the Delegated Nimiq wallet at manifest section sequence number '6'.

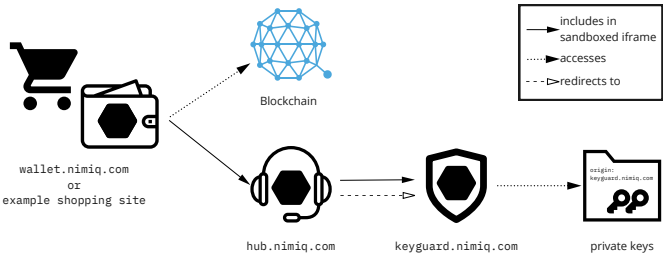

Fig. 1: Structure of Nimiq Ecosystem.

F. Compartmentalisation of Code and Development process

We further expand on Nimiq's Wallet application, this time as an example for compartmentalising the code and the signing process. Nimiq's Wallet application at no point has direct access to the users' private keys. It is treated the same way as any other third party application interacting with the Nimiq ecosystem (see Fig 1). It embeds the *Hub* which acts as an interface to the user's addresses and can trigger actions on the private keys. Access to the users' private keys is only possible through the Hub and pre-specified APIs. The Hub will then forward any request that needs to access the private keys to the *KeyGuard* component, which upon user input can decrypt the locally stored keys, perform the requested action, and return the result to the Hub.

The procedure `generate_manifest` produces the following manifest for Nimiq's Wallet. Observe that it heavily employs sandboxing. Both included iframes have the *sandbox* attribute set empty, meaning no exceptions defined.

```
{"url": "https://wallet.nimiq.com/",
  "manifest_version": "v0",
  "contents": [
    [five external scripts]
    { "seq": 3,
      "type": "iframe",
      "src_type": "link",
      "src": "https://hub.nimiq.com/iframe.html",
      "sandbox": "",
      "dynamic": true,
      "trust": "assert",
      "manifest": [[seven external scripts],
        { "seq": 7,
          "type": "iframe",
          "src_type": "link",
          "src": "https://keyguard.nimiq.com/",
          "sandbox": "",
          "dynamic": true,
          "trust": "delegate"}]}]}
```

Listing 7: Delegated content Nimiq Wallet's manifest.

The Wallet's manifest includes hub.nimiq.com in an iframe, containing, among other elements, the KeyGuard, which has a separate origin and thus exclusive access to the user's keys. For transactions, the Hub redirects to the KeyGuard. The KeyGuard is trusted, easy to audit, does not depend on any third party code and changes rarely. The KeyGuard manifest is as follows.

```
{"url": "https://keyguard.nimiq.com/",
  "manifest_version": "v0",
  "contents": [
    { "seq": 0,
      "type": "external",
      "link": "https://keyguard../web-offline.js",
```

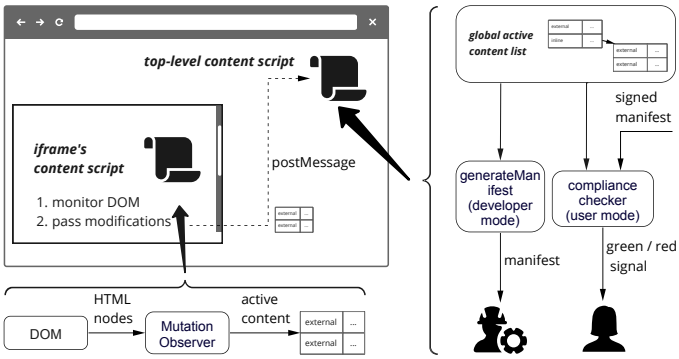

Fig. 2: Manifest file generation and metadata collection .

```
    "hash": "sha256-L8NMxOGkIW...",
    "load": "defer",
    "dynamic": false,
    "trust": "assert"
  },
  [two external scripts w/ same dynamic/trust.]]}
```

Listing 8: Nimiq Keyguard depends on its own content.

The Wallet manifest file reflects the web applications compartmentalisation: every component – Wallet, Hub and KeyGuard – runs on a different domain, hence locally stored information like the wallet key is inaccessible to the Hub or Wallet due to the same-origin policy.

With this setup, it is easy to compartmentalise the development process, too. A separate developer key could be used for the KeyGuard code given that it is already bound to a second domain. New KeyGuard releases would need to be signed by that key, which, internally, can be assigned additional oversight requirements. Without requesting a new key from the PKI, any bypassing of this procedure would either end up with code that cannot access the user's key or be provable with the signed manifest for the Wallet.

VII. Measurement Procedure

We present a practical active content measurement procedure that can be used to identify active elements and collect their metadata, allowing the client to check whether the web application follows the provided manifest. In development mode, the same procedure can be used to automatically generate a manifest file from an HTML document.

The measurement procedure is depicted in Fig. 2. The browser's rendering engine parses the raw HTML document and creates the DOM, observing the DOM for mutations, e.g. elements that are added at run time. Whenever an active element is appended, edited or removed from the DOM, the metadata agent will be triggered, which keeps a list of the active elements and their metadata.

The extension obtains access to the DOM by defining a *content script*, a script that runs in the context of the current page. This includes all pages loaded in top-level browser windows (e.g. Tabs), but also iframes within those. Content scripts running at the top level are responsible for collecting metadata on all active elements in their context. For nested iframes, they can only collect the metadata *about* the iframe like the attributes src_type, src and $sandbox$, but not inspect the document *inside* this iframe. The same-origin policy forbids this in many cases. We therefore use the iframe's content script to gather information: if the content script recognises that it is not at the top-level, it runs statelessly, collecting the metadata as usual, but reporting it to the parent window's content script via postMessage.

The metadata agent distinguishes script and iframe elements by their HTML tags. A script that has src attribute is $external$ otherwise it is $inline$. For external scripts Subresource Integrity (SRI) hashes, crossorigin and load attributes are collected. For inline scripts, hash is computed on the script and the load attribute is collected. Event handlers are searched inside all DOM elements checking if their attributes contain any of the global event attributes e.g. onclick in a list [19]. For event handlers, the hash is computed on the value of the event attribute. For iframes with $src\_type = external$, the metadata agent in the parent window collects the crossorigin and sandbox attributes and gathers the metadata about the document inside the iframe from its content script. Also, for each active element boolean $dynamic$ and $persistent$ scores are assigned by the metadata agent. An active content is considered dynamic if it is added after the window's load event; otherwise, it is static. Elements that get removed from the DOM are marked to be non-persistent, but still kept in the active content list for evaluation.

If the web page opted in, i.e. it has sent the `x-acc-js-link` header in the past and provided a valid manifest, then the metadata collector compares the metadata list with the list of active elements in the manifest. If the web page violates the protocol, the extension reports this to the user.

In developer mode, a failure to comply triggers the manifest generator to collect and generate metadata for the active elements. The `generate_manifest` procedure then produces a manifest file with $trust = assert$ for each active element based on the collected information, which can be easily adapted to other trust settings. This manifest represents the most restrictive manifest functional for this web application.

VIII. Signing and Delivering a Manifest

A valid signature on the manifest proves that the manifest was created by a known origin, i.e. a developer publicly associated with the website, and that it was not tampered with in transit. To sign manifests, we adopt the Signed HTTP Exchanges (SXG) standard. SXG is an emerging technology that makes websites portable. With SXG, a website can be served from others, by default untrusted, intermediaries (e.g. a CDN or a cache server), whereas the browser can still assure that its content was not tampered with and it originated from the website that the client requested. This allows decoupling the web developer from the web host and nicely fits our view of websites as software distribution mechanisms. The SXG scheme allows signing this exchange with an X.509 certificate that is basically a TLS certificate with the 'CanSignHttpExchanges' extension. Browsers will reject certificates with this extension if they are used in a TLS exchange, ensuring key separation. SXG certificates are validated using the PKI, allowing Accountable JS to be used with the existing infrastructure, although, currently, Digicert is the only CA that

provides SXG certificates. The lifespan of an SXG certificate is at most 90 days, limiting the impact of key leaks.

An SXG signature includes the HTTP request, as well as the corresponding response headers and body from the server. The signature is thus bound to the requested URL, in our case, the manifest file's URL. It also includes signature validation parameters like the start and end of the validity period and the certificate URL. If the current time is outside the validity period, SXG permits fetching a new signature from a URL. This URL is also contained in the (old) signature's validation parameters. These features provide a solid foundation for Accountable JS's signed manifests, allowing manifests to be cached during the validity period and enabling dynamic re-fetching and safe key renewals.

A web application in compliance with Accountable JS must deliver the signed manifest. If it is small enough, it can be transmitted directly via the HTTP response header (using the directive `x-acc-js-man`). Alternatively, the response includes the URL of the SXG file, using the HTML meta-tag or HTTP-response header `x-acc-js-link`. The signature in this file includes the manifest file (as the HTTP response body) and the manifest URL (part of the HTTP request). Also, the browser needs to check that the URL value in the manifest corresponds to the web application's URL (excluding the query part of the URL).

Providing a signed manifest indicates the website (i.e. the URL) opted into the protocol. From now on, the extension will expect an accountability manifest until the users explicitly choses to opt out.

Apart from the manifest generation, the signing operation and uploading the signature to the ledger can also be automated thanks to existing tool support for SRI and SXG. We stress that the signatures need only be computed if the JS code changes. Techniques like microtargeting are disincentivised by accountability (see Section II-A), hence the performance of the signature generation is of secondary concern.

IX. Protocol

In this section, we present the Accountable JS protocol. The end-to-end goal is to hold the developer accountable for the active content the client receives. Clients can compare this code with the manifest, hence, for honest clients, we can reformulate this task as follows:

- Clients should only run active content that follows the manifest. This is a setup assumption.
- Any manifest the client accepts needs to originate from the developer, even if the developer or server is dishonest. This follows from the non-repudiation of origin property of the signature scheme. A signed manifest was either signed by the developer, or the developer leaked their key.
- Whenever two clients accept a manifest with the same version number, that manifest must be the same, or they can provide non-repudiable proof that this was not the case. This is achieved by including a transparency log that gathers all manifest files with valid signatures.
- Whenever a client accepts a manifest with some version number, this version was the latest version in some client-defined time frame. This is achieved by a timestamping mechanism like OCSP-Stapling [20].
- A client can provide non-repudiable proof that they received a manifest from the web server. This is achieved by signing a client-provided nonce.

The developer of the website generates a manifest file for the web page that is identified with a URL, signs the manifest and publishes it in one or more public transparency logs (see Fig. 3 before t). The signature proves to the client that the developer takes responsibility of the manifest.

The $CodeStapling$ protocol ensures that, whenever the client accepted a manifest, the developer can be held accountable for publishing it. Nevertheless, the developer cannot be held accountable for delivering it to a client, as there is no proof for that. We thus define the $CodeDelivery$ protocol for non-repudiable code delivery (in Fig. 3 after t). With the HTTP GET request, the client C sends a nonce n signed with its signing key sk_C. The web server W responds with a signature on the HTTP response HTML, the client nonce n, and signed log timestamp sig_L. The client validates the log's signature and the developer's signature within. Should one of these checks fail, the client aborts and displays an error message. Then, the client compares the active content in HTML with the manifest; if they are consistent, the browser decides the web page adheres to the protocol.

X. Protocol Verification

We analysed Accountable JS with Tamarin [21], considering the protocol's security w.r.t. a Dolev-Yao adversary that can manipulate messages in the network and corrupt other processes to impersonate them. Using Tamarin's built-in stateful applied-π calculus [6], we could model a global state such as represented by the transparency log.

The protocol comprises five processes running in parallel:

$$!P_{Developer} \,|!P_{Webserver} \,|!P_{Client} \,|!P_{Log} \mid P_{Pub}$$

The first three processes model the role of the developer, web server and client, outputting and accepting messages as specified in Figure 3. The developer, web server and the client are under replication to account for an unbounded number of parties acting in each role. Any party except the log and the public process can become dishonest. This is modelled by giving control to the adversary, but only after emitting a $Corrupted$ event, which can be used to distinguish the party's corruption status in the security property. A corrupted party remains dishonest for the rest of the protocol execution.

The process P_{Log} models an idealised append-only log using insert and lookup operations to a global store [6]. Moreover, the built-in lock and unlock commands are used to ensure atomicity of the operations. Finally, the process P_{Pub} makes the public's ability to validate a client's claim explicit. Upon obtaining a claim (from the client), this process : (1) reads, from the log, the information that concerns the URL mentioned in the claim, (2) verifies the signatures in the claim and (3) matches the signed values with those in the log.

Using Tamarin, we prove the following properties which are detailed in Appendix A.

- **Authentication of origin**: The client executes active content only if the corresponding manifest was generated by the

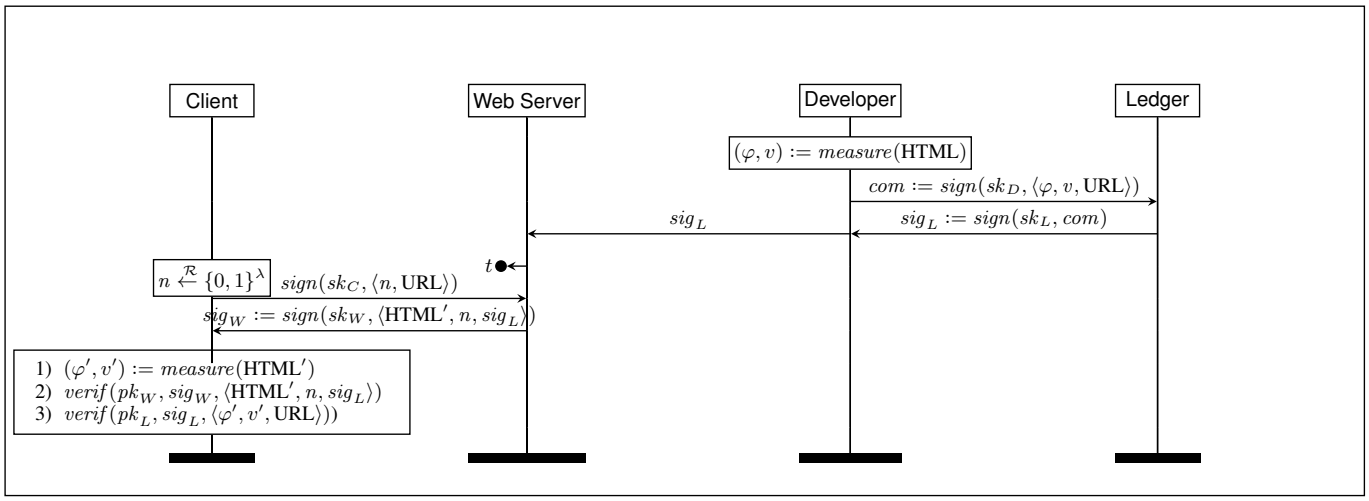

Fig. 3: Protocol flow: *CodeStapling* (before t) and *CodeDelivery* (after t).

.

honest developer unless the adversary corrupts the developer.
- **Transparency**: If the client executes code then its manifest is present in a transparency log in a sufficiently recent entry.
- **Accountability**: When the public accepts a claim, then even if the client was corrupted, the code must exist in the logs and the server must have sent that data (either honestly or dishonestly via the adversary).
- **End-to-end guarantee** : Only by corrupting the developer it is possible to distribute malicious code.

A. Code Verify Protocol

Meta's Code Verify [4] was published in March 2022 and made available as an extension. As of now, it is deployed only by WhatsApp Web. Intuitively, WhatsApp Web (the developer) submits a hash of their JavaScript along with a version number to Cloudflare, which Cloudflare then publishes to the end user. The end user's browser extension computes a hash on the JavaScript delivered from WhatsApp Web and compares it against the hash published by the Cloudflare. Given that the manifest is hashed instead of signed, Cloudflare is trusted for authenticity and thus constitutes a trusted third party, replacing the log. Moreover, users' IP addresses are sent to Cloudflare instead of to WhatsApp Web.

We likewise modelled Code Verify in Tamarin, considering the following five processes:

$$!P_{Developer} \; |!P_{Webserver} \; |!P_{Client} \; |!P_{Cloudflare} \mid P_{Pub}$$

Again, we assume the developer is separate from the web server. The protocol does not have a public log and does not include independent auditors. Instead, Cloudflare records the hashes for each version. To our knowledge, Cloudflare does not provide information about the history of submitted versions or which is most recent. As the public cannot inspect how often versions have changed, it relies on Cloudflare to implement countermeasures against microtargeting. Publicly available information [4] did not give information about such measures in Meta's deployment.

Under these considerations, we analysed the same properties, except for transparency, which, due to the lack of a public log, could not apply. We highlight the differences to our original properties below.

- **Authentication of origin**: The client executes active content only if the corresponding manifest was generated by the honest developer unless the developer or Cloudflare is corrupted.
- **Non-Accountability:** The data provided to the client is not sufficient to prove they received certain content from the web server, even if web server and Cloudflare are honest.
- **End-to-end guarantee**: Only by corrupting the developer or Cloudflare it is possible to distribute malicious code. In a separate lemma we show that, the developer by itself can indeed distribute malicious content.

The latter property indicates that Cloudflare's role as trusted party is not fully exploited yet. At least as far as we know [4], Cloudflare neither promises to ensure the code is harmless, nor does it guarantee to collect information to provide transparency or accountability. Nevertheless, the current message flow can be extended to provide such guarantees by having Cloudflare acts as a transparency log. Accountability can likewise be achieved by simply deploying signatures instead of a hashing scheme.

XI. Logging Mechanism

We would like clients to verify they received the latest and same version of the code as any other user. To this end, we assume a public append-only log to provide a public record of the software published and prevent equivocation attacks. The log does not determine which JS is considered malicious, but it provides proof of receipt and origin, and allows identifying malicious versions.

Such a public log is realistic to deploy: CT Logs [13] are used in the modern internet infrastructure. These logs store certificates, which are signed by CAs. In contrast, our logs need to store manifests signed by the developers. It is thus impossible to reuse the existing CT infrastructure, but we can closely follow the structure and properties of CT.

Websites that offer security-conscious services have an incentive to retain their reputation. Similar to how CT logs

operate, our log can be run by a party that wants to support such webpages. Third party monitors can keep the monitor honest and we allow third parties to submit signed manifests they observe in the wild.

When implemented naively, a logging mechanism can bring significant privacy implications: To confirm that other clients receive the same manifest, the client would need to consult the log on each request and reveal the URL to the log. We can mitigate these privacy issues by allowing the web server, which learns each request anyway, to include a signed and timestamped inclusion statement from the log instead. This is similar to the OCSP-Stapling for certificate revocation status requests [14]. While it mitigates the privacy issues of consulting the log, it instead requires the user to trust the specific log selected by the web server. We outline other approaches to solve the trade-off between trust and privacy in Section XIV.

Overall, our transparency log needs to provide interfaces to at a minimum:

- store the signed manifest file (including its version number) bound to a URL,
- query the latest signed manifest file for a URL in the logs,
- form a signed response for a query that can be pre-fetched by the web server to staple it to each request from the clients.

A possible implementation of this functionality could be based on Verifiable Log-Based Maps [22]. An implementation of this structure for Trillian [7], the software running Google's CT server, is currently in progress [23], with the goal of supporting transparency in certificate revocation.

Availability, scalability and the size of the transparency logs are other implications. Be it submitting a new manifest to the log or collecting the latest version of manifest for a URL, low latency to access the network of transparency logs can be achieved by eliminating the single point of failure by adding multiple logs that will provide load balancing. The mechanism proposed for query privacy will also decrease the number of requests to the logs since the web server will provide the stapled result in most cases.

Websites that frequently update their active contents can create significant burden on the log size. We calculate approximately how many times each log can be updated for a limited time and space. We assume a non-leaf node overhead is approximately 100 bytes and for the leaf nodes it is 700 bytes(signature 600 bytes + 100 bytes). If a log provider has 100 TB of space for 5 years, it can contain 137 billion signatures in total. To make sense of this number, take the following example. We start with a log of 10M URLs with eight updates per month on average. The number of URLs also increases exponentially at a rate of 1% with each update (i.e. also eight times per month). [5] This number would be well below 137 billion signatures.

XII. Evaluation

We implemented Accountable JS in a Chrome extension [12, folder `accjs-extension`] for demonstration and prototyping. Ideally, the measurement procedure should be part of the browser's rendering engine, since it can access the response body and observe mutations to elements first-hand. Our measurements here can thus be (promising) upper bounds. We elaborate on the technical limitation imposed by the extension SDK in Section XIII.

We come back to the use cases from Section VI and measure how the extension affects the following metrics: 1) number of additional requests, 2) size of additional traffic, 3) time until the browser paints the first pixel / the largest visible image or text block[6] / until the web page is fully responsive. 4) total blocking time, i.e. time during which web page cannot process user input. We consider differences below 100 ms to be imperceptible to the users, differences of 100-300 ms barely noticeable and differences above 300 ms noticeable.[7]

Evaluation environment: Measurement took place on a MacBook Pro with 2 GHz Intel Quad-Core i5, 16 GB RAM and macOS Monterey 12.5.1 with Google Chrome 107.0.5304.121. The results are compiled in Table II. We measured the number of additional requests and traffic using Chrome's developer tools and the rendering metrics using Lighthouse (set to 'desktop simulated throttling'). Unfortunately, WhatsApp Web is incompatible with Lighthouse, so we instead computed the combined duration of all tasks performed by the browser using Puppeteer Page metrics [27]. We automated this process using Puppeteer and NodeJS and perform $n = 200$ trials per website and configuration to minimise the impact of network latency on page loads.

Configurations: For performance evaluation, we compare the CSP built into the browser with the Code Verify and Accountable JS extensions as follows:

1) **Baseline:** disabled CSP and extensions.
2) **CSP:** CSP active, no extension.
3) **Accountable JS:** CSP inactive, only Accountable JS extension active.
4) **Code Verify:** CSP inactive, only Code Verify extension active. This configuration only applies to WhatsApp Web, as Code Verify currently only supports Meta websites.

Experiments: We consider the examples from Section VI: Hello World, WhatsApp Web, Trusted Third-Party, Delegate Trust to Third Parties (Nimiq A), Untrusted Third Party (Google AdSense and Nimiq B). For the compartmentalisation experiment on Nimiq's Wallet, we use a different baseline that we will discuss below. For the CSP measurement, we defined CSP headers for each website that listed all active content in the Accountable JS manifest files. We collected all valid sources of external scripts and hashes for the external and inline scripts in CSP's `script-src` directive, hashes for event handlers in `script-src-attr` and sources for iframes in `child-src`. For the Accountable JS experiment, we first navigate to the target website and wait for ten seconds for the page to load. Thereafter, using the `generate_manifest`

[5] e.g. after the first update, 10M updates along with 100k new URLs are appended to the existing 10M, resulting in a total of 20.1M.

[6] More precisely: the 'largest contentful paint'.

[7] We derive these performance categories from the RAIL model [24]. According to RAIL, users feel the result is immediate if < 100 ms and feel they are freely navigating between 100-1000 ms (see also [25]). However, we found this gap is too wide to ignore, and split the category at 300 ms for an unusually common delay in web apps due to the 'double tap to zoom' feature on iPhone Safari [26].

TABLE II: Evaluation results on case studies: The second and third columns show the number and total size of additional requests made by the extension, i.e. the number of signed manifest and certificate. Each subsequent block provides Lighthouse performance metrics for rendering time and the total time that the browser spends unresponsive. For each metric, we compare the baseline (no Content Security Policy (CSP), no Accountable JS) with the overhead incurred by enabling CSP and enabling the Accountable JS extension (leaving CSP disabled). For compartmentalisation, the baseline is with the extension activated but the same signing key for all Nimiq components. All the time values are averages over $n = 200$ runs and given in milliseconds. The additional traffic(kB) value is affected by the size of the signature and SXG certificate. Signatures are generated on uncompressed manifest JSON files.

	additional network ...		time to ... baseline + CSP overhead + Accountable JS overhead											
case study	requests	traffic (kB)	first pixel			largest element			reactive			blocking time		
Hello World	2	2.06	196	+1	+20	197	+0	+23	196	+1	+24	0	+0	+0
Trusted Third-Party	2	2.46	462	+0	+21	462	+0	+21	462	+0	+21	0	+0	+0
Delegate Trust (Nimiq A)	3	9.93	262	+3	-10	262	+3	-10	5591	-29	-144	172	+4	+87
AdSense + Nimiq B	3	15.62	747	+2	+91	901	+5	+68	6034	+1	-82	159	+3	+77
Compartmentalisation	2 + 2	8.66 +1.10	2200		-17	4675		+20	5321		+115	212		+7

in the extension, we download the manifest file and self-sign it using the *gen-signedexchange* command line tool [28]. For Nimiq A+B and AdSense, we changed the *trust* attribute for the external element(s) to *delegate* before signing. We publish this signed manifest via a local web server and configure the web server to provide a response header pointing to a URL. We also ensure the website provides SRI tags for *external* scripts. Evaluation procedures of each case study are elaborated in the full version [16, Appendix C].

Results: The CSP configurations show an imperceptible overhead in all case studies. This is hardly surprising, as CSP is built into the browser built-in and can validate resources during rendering. The Accountable JS configurations likewise have an imperceptible overhead in all case studies. Moreover, the traffic requirements are modest and incur only modest blocking time. For Nimiq A, the traffic requirements are about 9.9 kB for the additional signature. In terms of performance, CSP and Accountable JS' overhead are comparable, except the total blocking time is slightly higher than CSP. Besides, the time to interactive value unexpectedly decreases more with Accountable JS than CSP. However, the difference is minimal and could possibly be explained by a) network latency, (b) side effects of the browser's just-in-time compilation or scheduling or (c) a side effect of the former two on how Lighthouse evaluates the reactive metric. Nimiq is a complex web application heavily dependent on external data, in particular the remote blockchain it connects to.

Discussion: The Accountable JS configurations have an imperceptible overhead which is slightly higher than the CSP configurations. Recall that the CSP is built in the rendering engine whereas Accountable JS runs as a browser extension. Accountable JS has to perform signature validation, meta data collection and a final compliance check. The prototype achieves good performance overheads by measuring all elements simultaneously and combining their results. The browser extension panel displays the results instantaneously, while the evaluation is in progress, although the evaluation is usually too quick for the user to notice. Moreover, the traffic requirements are modest and incur little blocking time.

For AdSense + Nimiq B, the network overhead is slightly higher than Nimiq A. This is due to the larger size of the manifest, which now also includes AdSense. We again observe an imperceptible impact on performance with Accountable JS.

The difference between Code Verify (220ms) and Accountable JS (244ms) on WhatsApp Web is small. This is remarkable, because Code Verify only applies SRI checks on external scripts but not event handlers or iframes. In contrast to Accountable JS, the order of active elements is ignored, attributes are not checked (e.g. `load='async'` for scripts) and a short hash value is downloaded from Cloudflare, rather than a signature.

Compartmentalisation: For compartmentalisation, we evaluate the impact of the additional signing key. We signed Nimiq Keyguard, which is embedded in Nimiq Wallet, with a different signing key and set the Keyguard's *trust* attribute to *delegate* in the Wallet's manifest. The baseline therefore also has the Accountable JS extension activated, but uses the same signing key on all Nimiq components. The Wallet's manifest includes the Hub's manifest inside and the Hub's manifest declares the Keyguard with $trust = delegate$ in its manifest section. Thus a separate manifest is required for the KeyGuard. Also, this time a separate signing key is used for the KeyGuard manifest. For the baseline performance, we inline the KeyGuard's manifest as an entry for its iframe in the Wallet's manifest, thus having one manifest and one signing key, and activate the extension.

In the compartmentalisation evaluation, we observe that there are two more round trips and slightly higher traffic overhead (about the overhead of Accountable JS, w.r.t. the overall page traffic of 4.6 MB). This is due to downloading the extra SXG certificate and manifest for Keyguard. The effect on the rendering metrics is small; the barely noticeable increase for time-to-reactive value can again be explained with network latency and side effects described above. This is because the delegated manifest can be validated in parallel to rendering, while it is inlined in the baseline scenario and thus validated in sequence.

Due to stapling, the overhead for clients to verify that they received the latest version of the code (and thus the same as any other user), is negligible. The web server staples a query result, i.e. the log's signature on the signed manifest, to each request. The signatures use 2048-bit RSA keys and are 256 Byte long.

XIII. Limitations of Prototype

The browser extension is a prototype to evaluate performance and applicability of the approach. The advantage of an extension (as opposed to modifying the browser's source or writing a developer plugin) is that users can easily experiment with its code. On the other hand, extensions cannot interrupt the browser's rendering engine. Thus we inject a content script that can apply the client-side operations of the protocol to the browser window. The content script runs in the same context as the web page; hence it can observe changes to the DOM via the Mutation Observer. Since the extension cannot access to the browser's rendering engine, some active elements can be added within a small time frame before the Mutation Observer is registered. This race condition is a limitation of using the extension and fixable by closer integration into the browser.

Another limitation is that other browser extensions may interfere with the measurement by injecting active content to the web page. Since extensions cannot distinguish website code from the code that other extensions injected to the web page, this can break the measurement. This is the correct behaviour, as the website developer cannot attest to every possible modification of the active content by other extensions, however, there are various client-side solutions: (a) closer integration into the browser could distinguish active content injected by websites, (b) the extension could provide an API for third party extensions to register modifications or (c) an allowlisting for the most common extensions that gives a warning to the user.

XIV. Related Work

We first discuss how Accountable JS relates to other (proposed) web standards with seemingly similar goals, before discussing related academic proposals.

CSP was introduced to counter Cross-Site Scripting (XSS) attacks. They specify runtime restrictions for the browser, typically the set of allowed sources for scripts, iframes, stylesheets, etc., including eventual requirements for sandboxing. Like accountability manifests, CSPs can specify which sources are allowed and, combined with SRI, fix their content. This is comparable to a manifest file that includes types with *trust* set to either *assert* (if SRI is employed) or *blind-trust* (otherwise). By contrast, CSPs do neither cover the order nor possibly nested active contents (e.g. iframe within iframe). Mixed ordering of active content may create malicious activity, a site loading script A before script B may mean something different from loading B before A. A site that only uses CSP cannot catch that behaviour, whereas in Accountable JS, we take the order into account. Most importantly, in CSP, there is no means of delegating trust and no distinction between web server and developer. Steffens et al. [29] show that outsourced content is one of CSP's major deployment obstacles. Instability in third party inclusions (e.g. adbidding code that delivers code from different resources) forces first parties to continuously update the CSP. Techniques like in Section VI-E allow developers to delegate trust to the third party. Moreover, CSP tries to mitigate XSS attacks throughout the web, whereas Accountable JS targets websites willing to allow for an audit. The ability to identify the code that is run is a key requirement for that. Overall, the goals of CSP and Accountable JS are orthogonal and can be combined. It is possible to generate a CSP from a manifest file.

The Web Package proposal (currently in draft status [30], see Fig. 4) aims at packaging web applications for offline use. Web packages provide a declaration of the web application's metadata via *Web App Manifests* [31], a serialisation of its content via *Web Bundles* [32], and authenticity via SXG [33]. We likewise employ SXG to provide authenticity of origin via signatures. SXG, like Accountable JS, decouples web developer from website hoster. Web App Manifests, despite their name, are only superficially related. They contain startup parameters like language settings, entry points and application icon, e.g. for 'installable web application' displayed in a smartphone's launcher. Web Bundles are a serialisation format for pairs of URLs[8] and HTTP responses. They represent a web application as a whole, but a signature on a web bundle would change with every modification of a web pages' markup. Web Packages are thus not competing with Accountable JS. Instead, both standards are compatible. A web bundle can contain `x-acc-js-link` in the header of its entry point's HTTP response, triggering the browser to validate the manifest. The manifest is specified via a URL that also included in the web bundle. This URL maps to an HTTP response that contains the manifest in its content part.

Signature-based SRI [35] proposes easier maintainable SRI tags to protect against script injections, by including signature *keys* instead of hashes. These enable validating the provider of the third party script, instead of their content, like the trust relationships expressed with $trust = delegate$. The tags are part of the HTML code, instead of the manifest file. Signing the HTML files is impractical, as they are frequently changing.

Service Workers [36] are Network proxies programmable via JavaScript, often used to perform URL response caching, separate from the browser cache. Theoretically, a compliance check like our measurement could be implemented in a service worker, but (a) the service worker would need to be delivered correctly and (b) service workers lack access to the DOM and thus information about how active elements used.

We will now discuss related academic work. Accountability in the web requires non-repudiable proof. For static assets, this can, in principle, be provided by digital signatures (e.g. via SXG and web bundles, see above), but recreating the signature for each exchange is costly. We solve this via

[8] More precisely, HTTP *representations* [34].

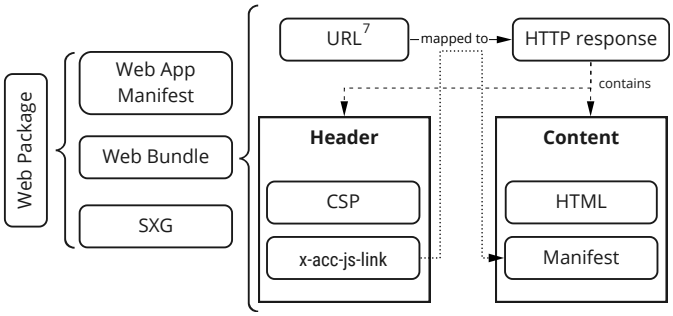

Fig. 4: Accountable JS in the context of other web technologies.

a simple challenge-response mechanism. Ritzdorf et al. [5] provide a full-fledged solution, giving non-repudiation for the entire communication, optionally hiding sensitive data. The statement we prove is that the client has *obtained* certain active content, not that they *execute* it. Ensuring a remote partner runs certain software is the goal of remote code attestation (e.g. [37]). Outside embedded systems, this is typically based on a trusted execution environment (e.g. TPM, SGX). While the browser (and for that matter, our extension) could provide a trusted execution, establishing trust in the correctness of the browser is the crux.

Our work relies on a transparency log. As mentioned before, Trillian's [7] verifiable log-based maps would fit the bill, but there are many ways to implement such a store. The most interesting aspect is privacy. We propose an approach based on stapling, an established method for revocation management [20], but other techniques promise privacy, too. CONIKS [8] provides a log mapping user identities to keys, keeping the list of all user identities in the system private. This is not useful in our case, as the URLs (the domain of our mapping) are not secret, but which URL a user accesses. Multiparty protocols for Private Information Retrieval [38], Private Set Intersection (e.g. [39]) or ORAM [40] lack efficient database updates, mechanisms to efficiently update precomputation steps, or only preserve k-anonymity for URLs. K-anonymity is often not enough if we consider that an attacker, e.g. a censor, tries to punish access to a few critical URLs, each of which may end up in a bucket with uncritical, but also not frequently visited URLs. Finally, Accountable JS may be an enabler for formal verification of web applications, as users are potentially able to link the code they receive to code to published verification results. Various static and dynamic analyses target JavaScript already [41], [42].

Although we showcased only a single approach to code compartmentalisation (as it is being deployed by our real-world example), other approaches are also compatible with Accountable JS. Language-based isolation methods like BrowserShield [43]) rewrite JavaScript into a safer version preventing or mediating access to critical operations like `createElement` or `eval`. If the code is rewritten on the client (typically using JavaScript), the developer declares the wrapper that fetches the code and deals with the code rewriting in the manifest file. If the code is rewritten on the server, the developer declares the transformed JavaScript code that will be delivered to the user. Frame based isolation methods (e.g. AdJail [44]) that isolate the third party code inside iframe are also compatible with our proposal, see the use case for untrusted third party code in Section VI-E.

XV. Discussion

We provide a solution that allows users to detect if they are microtargeted by developers and to prove this to the public if it is the case. Sending different codes to classes of users might not be outlawed in many countries, but sending malicious code is. Our solution neither provides a code audit tool nor does it propose a framework, legal or otherwise, for the punishment of malicious code distribution. It provides, however, verifiable data that authorities can use to evaluate which code was published and whether that code was delivered to a user. Moreover, the protocol provides users with a claim that includes the delivered code and the identity of the developer.

The transparency logs can be used as a point of reference for the public code for auditing and evaluating. Honest developers aim to make their code easy to audit; dishonest developers thus risk loss of reputation if they microtarget users (as frequent updates are visible on the transparency logs), silently opt out of the system (as this will be caught by users that received a previous opt-in), or provide obfuscated code (due to the log).

Honest developer will benefit from a good reputation and their ability to provide proofs for any efforts they make toward independent audit or formal verification. Clients, who often debate a website's reputation in a public forum (e.g. the case of ProtonMail or Lavabit) obtain data to substantiate positive and negative claims.

We stress that accountable code delivery is necessary to connect auditing results to the code users actually run, but does not by itself guarantee the safety of this code. Realistically, it will take some time until software analyses are mature enough to handle this at scale. Assuming, however, that such analyses may not necessarily run at each browser independently, authentic code delivery appears to be a necessary first step.

Moreover, Accountable JS only authenticates the active content, thereby exposing the active content to data-only attacks, e.g. modified button labels or form URLs. A signature on the content of a web application could be achieved by building on Web packages/Web bundles (which we discussed in Section XIV), however, this approach would be too static and inflexible for the requirements of the current web ecosystem. Thanks to accountability, the developer would take responsibility for the active content that they published, in this case, for code that is vulnerable to data-only attacks. Realistically, there would not be consequences, because they can plausibly point to the dire state of verification of JavaScript—which is at least partially because users could thus far not be sure to receive the verified code anyway. Accountable JS choice to validate the active content only is a compromise and possible starting point for future work, as we discuss in the next section.

XVI. Conclusion

With Accountable JS, we provide a basis for the accountable delivery of web applications, and thus a first step towards re-establishing the trust between a user and the web application code they run on their computers. How to achieve security – via audit, code analysis or formal verification – is a question that we left open intentionally. Accountable delivery is, nevertheless, a requirement for any non-instantaneous analysis.

A key question for verification and audit is how to relate the web page's user interface to the active content. As some desirable security properties concern user input, we would like to give guarantees about, e.g. form fields. We can account for the JavaScript code that address them by ID, but those are invisible to the user. Future work may investigate how to establish stronger ties between the manifest and the user interface.

Acknowledgements: This project was partially funded by the ERC Synergy Grant IMPACT (with grant agreement number 610150) and a research award for privacy-preserving technologies from Meta research, specifically for the "Transparency.js, transparency for active content" initiative.

References

[1] I. Esiyok, L. Hanzlik, R. Künnemann, L. M. Budde, and M. Backes, "TrollThrottle —Raising the Cost of Astroturfing," in *Applied Cryptography and Network Security*, 2020. (visited on 05/19/2021).

[2] M. Schwarz, M. Lipp, and D. Gruss, "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks," in *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. (visited on 05/19/2021).

[3] C. Cimpanu, *Chrome extension caught hijacking users' search engine results*, 2019. [Online]. Available: https://www.zdnet.com/article/chrome-extension-caught-hijacking-users-search-engine-results/.

[4] R. Hansen and V. Silveira. "Code verify : An open source browser extension for verifying code authenticity on the web." (2022), [Online]. Available: https://engineering.fb.com/2022/03/10/security/code-verify/ (visited on 03/10/2022).

[5] H. Ritzdorf, K. Wust, A. Gervais, G. Felley, and S. Capkun, "TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing," in *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. (visited on 06/17/2021).

[6] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[7] "Trillian." (2021), [Online]. Available: https://github.com/google/trillian (visited on 09/20/2021).

[8] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[9] S. Meiklejohn, J. DeBlasio, D. O'Brien, C. Thompson, K. Yeo, and E. Stark, "SoK: SCT Auditing in Certificate Transparency," *PoPETs*, no. 3, 2022. (visited on 11/21/2022).

[10] D. Carlin, J. Burgess, P. O'Kane, and S. Sezer, "You could be mine(d): The rise of cryptojacking," *IEEE Secur. Priv.*, no. 2, 2020.

[11] "Adsense program policies." (2021), [Online]. Available: https://support.google.com/adsense/answer/48182?amp;stc=aspe-1pp-en (visited on 07/19/2021).

[12] I. Esiyok, P. Berrang, K.-C. Gordon, and R. Künnemann, *Supplementary material*, 2023. [Online]. Available: https://github.com/iesiyok/accountable-js.

[13] B. Laurie, A. Langley, and E. Kasper, *Certificate Transparency*, RFC Editor, 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6962.txt.

[14] Y. N. Pettersen, *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*, 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc6961.txt.

[15] "Using subresource integrity." (2021), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity#using_subresource_integrity (visited on 11/03/2021).

[16] I. Esiyok, P. Berrang, K.-C. Gordon, and R. Künnemann, *Accountable js full version*, 2023. [Online]. Available: https://arxiv.org/abs/2202.09795.

[17] A. Eicholz, S. Moon, A. Danilo, T. Leithead, and S. Faulkner, "Sandboxing," W3C Recommendation, 2021, https://www.w3.org/TR/2021/SPSD-html52-20210128/browsers.html.

[18] "Is it allowed to use iframe." (2020), [Online]. Available: https://support.google.com/adsense/thread/24384322/is-it-allowed-to-use-iframe?hl=en (visited on 11/05/2021).

[19] *Html living standard: Event handlers on elements, document objects, and window objects*. [Online]. Available: https://html.spec.whatwg.org/#event-handlers-on-elements,-document-objects,-and-window-objects.

[20] D. Eastlake *et al.*, "Transport layer security (tls) extensions: Extension definitions," RFC 6066, 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6066.txt.

[21] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, 2013.

[22] A. Eijdenberg, B. Laurie, and A. Cutter. "Verifiable Data Structures." (2015), [Online]. Available: https://github.com/google/trillian/blob/b7ea8d2ca870e5b8ae1c05e9d2a33c4fdcca4580/docs/papers/VerifiableDataStructures.pdf (visited on 11/05/2021).

[23] "Trillian – experimental Beam Map Generation." (2021), [Online]. Available: https://github.com/google/trillian/tree/b7ea8d2ca870e5b8ae1c05e9d2a33c4fdcca4580/experimental/batchmap (visited on 11/05/2021).

[24] *Measure performance with the rail model*, 10, 2020. [Online]. Available: https://web.dev/rail/.

[25] J. Nielsen, *Response times: The 3 important limits*, 1, 1993. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/.

[26] T. VanToll, *What exactly is..... the 300ms click delay*, 21, 2013. [Online]. Available: https://www.telerik.com/blogs/what-exactly-is.....-the-300ms-click-delay.

[27] *Page.metrics method*. [Online]. Available: https://pptr.dev/api/puppeteer.page.metrics/.

[28] "Signed exchange generation." (2021), [Online]. Available: https://github.com/WICG/webpackage/tree/master/go/signedexchange (visited on 11/05/2021).

[29] M. Steffens, M. Musch, M. Johns, and B. Stock, "Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI," in *Proceedings 2021 Network and Distributed System Security Symposium*, 2021. (visited on 09/20/2021).

[30] J. Yasskin, *Use Cases and Requirements for Web Packages*. [Online]. Available: https://datatracker.ietf.org/doc/draft-yasskin-wpack-use-cases/.

[31] Marcos Cáceres, Kenneth Rohde Christiansen, Mounir Lamouri, Anssi Kostiainen, Matt Giuca, and Aaron Gustafson, *Web App Manifest*. [Online]. Available: https://www.w3.org/TR/appmanifest/.

[32] J. Yasskin, *Web Bundles*. [Online]. Available: https://wicg.github.io/webpackage/draft-yasskin-wpack-bundled-exchanges.html#name-semantics.
[33] J. Yasskin. "Signed http exchanges." (2021), [Online]. Available: https://wicg.github.io/webpackage/draft-yasskin-http-origin-signed-responses.html (visited on 04/13/2021).
[34] R. Fielding, M. Notthingham, and J. Reschke, *HTTP Semantics*. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-15.txt#section-8.
[35] M. West, *Mikewest/signature-based-sri*, 13, 2020. [Online]. Available: https://github.com/mikewest/signature-based-sri (visited on 07/20/2021).
[36] A. Russell, J. Song, J. Archibald, and M. Kruisselbrink, *Service Workers 1*. [Online]. Available: https://www.w3.org/TR/service-workers/.
[37] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, 31, 2008. (visited on 07/21/2021).
[38] L. Fortnow, "Private information Retrieval Survey,"
[39] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, "SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension," in *Advances in Cryptology – CRYPTO 2019*, 2019.
[40] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, no. 3, 1, 1996. (visited on 07/20/2021).
[41] D. Park, A. Stefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 3, 2015. (visited on 07/20/2021).
[42] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic Execution for JavaScript," in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, 3, 2018. (visited on 07/20/2021).
[43] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic HTML," *ACM Trans. Web*, no. 3, 2007.
[44] M. T. Louw, K. T. Ganesh, and V. Venkatakrishnan, "AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements," in *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

Appendix A: Verification of Security Properties

By default, Tamarin assumes that the adversary controls the network. Our model allows the adversary to impersonate the untrusted parties in the protocol and thereby access their secrets. This is logged with a $Corrupted(p)$ event in the trace with p an identifier for the corrupted principal.

We model the principals in the following structure using applied-π calculus.

```
in($p);
(
    ( event Corrupted($p);
      out(sk($p)) )
  | out(pk(sk($p)))
  | (
      /* process goes here */
    )
)
```

The shortcut $\$p$ denotes that the term p is a public value. The attacker, by inputting the public value $\$p$ can pick some identifier for the party. Then, if the attacker corrupts the party, a corruption event is emitted and the attacker gets access to the secret key. The public (verification) key is emitted so that other parties can use it to verify the signed messages of p. We exemplify the process with the example of $P_{Developer}$ as follows:

```
in($D);
(
    [..]
  | (
      in(<$manifest, $url, $v>);
      event DUploads($D, $url, φ);
      out(<'update', $D, $manifest, $url, $v, φ>)
      ...
    )
)
```

This code snippet includes the interaction with the network via `in` and `out`, which is represented by the attacker. The attacker hence inputs the public values $manifest$, url and version number v, then the developer process computes a signature φ from these values and sends an update message to the log including all that information. Events are annotations associated with the parts of the processes that enable to define restrictions and security properties. In this example, before sending the update message to the log, the developer logs a $DUploads$ event in the trace, annotating the developer's new code update request to the transparency logs.

The process P_{Log} represents the transparency log as a protocol party that can receive and send messages, and in addition apply insert and lookup operations to an append-only global store. The applied-π calculus provides constructs for modelling the manipulation of a global store. The code snippet below includes an insert and a lookup operation.

```
insert <$D, $L, 'version', $url>, $v;
...
lookup <$D, $L, 'manifest', $url> as $manifest
    in P else Q
```

The insert construct associates the value $\$v$ to the key which is a tuple $< \$D, \$L, `version`, \$url >$ and successive inserts overwrite the old values. The lookup construct retrieves the value associated with the key $< \$D, \$L, `manifest`, \$url >$ and assigns it to $\$manifest$ variable. If the lookup was successful, it proceeds with process P, otherwise with Q. $\$D$ stands for the developer's identity, whereas $\$L$ stands for the log's identity. Since there are unbounded number of developers and logs; we associate the values that are stored in the global store with the URL and the identities of the related developer and log for uniqueness.

Our model also includes lock and unlock, which the stateful applied-π calculus defines for exclusive access to the global store in the concurrent setting. The code snippet below shows an example of lock and unlock operations used in our protocol.

```
lock $url;
insert <..., $url>, ...;
```

```
...
unlock $url;
```

When a $\$url$ is locked, any subsequent attempt to lock the same $\$url$ will be blocked until it is unlocked. We provide exclusive accesses based on the $\$url$, when the log attempts to insert a new value to the global store. This is an over approximation: if a lock requires exclusive access independent for every write (independent of the URL) our model correctly captures this behavior too. We do not require locks for other reads, which also increases generality.

Security properties and restrictions are first-order formulas over the annotated events and time points. Universal quantification (meaning: for all) and existential quantification (meaning: there exists) are used to check if the security property formula (lemma) holds for all examples in the domain or there exists at least one example that satisfies the formula respectively. If the lemma holds for the former case then the Tamarin Prover shows that it is proven, whereas for the latter case a satisfying example is presented to the user. The time points enable to account for event order in the trace, where e.g. $E@i$ means that event E was emitted at index i in the trace. We prove that the following security properties hold in our protocol:

Theorem 1 (Authentication of origin)**.** *Intuitively, the client will only execute active content code (signified by the event CExec with url and manifest φ) if the code was uploaded by the honest developer D (logged the event DUploads), or the developer was corrupted. The KU event is emitted whenever the attacker (who is acting on behalf of the corrupted party D) constructs a message. We simplify the formula as follows:*

$$CExec(\$D, \$url, \varphi) \implies DUploads(\$D, \$url, \varphi) \vee (Corrupted(\$D) \wedge KU(\$url) \wedge KU(\varphi))$$

Formally, the lemma is: for all CExec events there exists either an earlier DUploads event or there exists a Corrupted($D) event and KU events before CExec event.

Theorem 2 (Transparency)**.** *If the client executes JS code c for url with timestamp ts (CExec′), then there is a corresponding log entry (Log) and it was deemed recent (CRecent) by the client. The session identifier sid binds the moment when the client checks the timestamp is recent (CRecent) to the moment it executes (CExec′) the code.*

$$CExec'(\$url, sid, c, ts) \Rightarrow Log(\$url, c, ts) \wedge CRecent(sid, ts)$$

Authentication of origin and transparency describe the proactive behaviour of the extension. The following theorems cover the reactive behaviour. We first establish that a claim that a client submits to the public is non-repudiable, i.e. that a corrupted client cannot forge false evidence to implicate honest parties.

Theorem 3 (Accountability)**.** *When the public accepts a claim (identified with server id, url, manifest, client nonce and log timestamp) then, even if the client was corrupted, the code must exist in the logs (Log′), and the server must have sent that data, either honestly, or dishonestly via the adversary.*

$$PAccept(\$W, \$url, \varphi, n, ts) \implies Log'(\$url, \varphi, ts) \wedge (WSend(\$W, \$url, \varphi, n) \vee (Corrupted(\$W) \wedge KU(\$W, \$url, \varphi, n))$$

Here, the event WSend is emitted by W (who is honest) right before it sends the signed tuple sig_W to C in Fig. 3.

Theorem 4 (End to end guarantee)**.** *When the client executes a malicious code, then a corrupted developer is necessary to distribute it.*

$$CExec(\$D, \$url, \text{`}malicious\text{`}) \implies Corrupted(\$D)$$

Theorem 5 (End to end non-guarantee)**.** *When the client executes a malicious code, then a corrupted developer is sufficient to distribute it.*

$$Ex.\ CExec(\$D, \$url, \text{`}malicious\text{`}) \implies (All\ x.\ Corrupted(x) \implies (x = \$D))$$

Tamarin reports these results within 3 hours on a 16-core computer with 2.6 GHz Intel Core i5 processors and 64 GB of RAM. The proof is fully automatic, but relies on a so-called 'sources' lemma to specify were certain messages can originate from. We specified this lemma manually, but it is verified automatically. The full protocol can be found in the supplementary material [12].

Appendix B: Claim Verification

The public runs a procedure to verify the claim generated by a client that was allegedly targeted by a website. As shown in the Appendix A Theorem 3, a claim is identified with server name, URL, manifest, request nonce and the timestamp that was set for the manifest by the ledger. The signatures on the request and the response data are verified, and the request nonce is asserted with the server nonce for authenticity. Next, the delivered content behaviours are checked against the manifest using the measurement procedure. Then, the public evaluates if the manifest is the latest version on the ledger using the timestamp. If the evaluation fails in any of these steps, then the claim is accepted.

Appendix C: Glossary

CA Certificate Authority
CDN Content Delivery Network
CSP Content Security Policy
CT Certificate Transparency
DOM Document Object Model
JS JavaScript
OCSP Online Certificate Status Protocol
PKI Public Key Infrastucture
SPA Single Page Applications
SRI Subresource Integrity
SXG Signed HTTP Exchanges
TLS Transport Layer Security
XSS Cross-Site Scripting