WRIT: Web Request Integrity and Attestation against Malicious Browser Extensions

Giorgos Vasiliadis[*][†], Apostolos Karampelas[†][¶], Alexandros Shevtsov[†], Panagiotis Papadopoulos[†], Sotiris Ioannidis[†][‡],and Alexandros Kapravelos[§],

[*]Department of Management Science and Technology, Hellenic Mediterranean University, Agios Nikolaos, Greece [†]Institute of Computer Science, Foundation for Research and Technology - Hellas, Heraklion, Greece [‡]School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece [§]Department of Computer Science, North Carolina State University, USA [¶]Tenable Inc.

Abstract—The powerful capabilities of modern browsers have pushed the web application logic to the user side, in order to minimize latency, increase scalability of the service and improve users' quality of experience. What is more, browsers provide a rich toolchest for browser extensions to provide additional functionality, but at the same time enable them to become a powerful vehicle for malicious actors. Such actors may spy, phish or fraud users, thus making the user's browser untrusted for the web servers.

In this paper, we present WRIT, a practical framework that enables websites to protect critical functionality from abuse in the presence of malicious extensions. In WRIT, the integrity of outgoing web requests is attested and verified to ensure they were triggered by a user's action and not automatically generated by a malicious browser extension. WRIT is immediately applicable by leveraging existing HTML5 and other native browser features and does not require any modification of the browser. Performance results of our prototype show that it adds a negligible 7.29 ms latency to sensitive user-triggered actions (e.g., post message).

1 Introduction

Browser extensions enable developers to augment the baseline functionality provided by browsers and enhance the user experience. Yet, the very same rich capabilities of theirs can provide a powerful vehicle for attackers to perform malicious actions [32], [8], [55]. There are various cases discovered over the years, where malicious browser extensions were used by attackers to infect users' browsers in order to gain control of the code that web servers sent to the users during browsing [13], [44], [35], [59]. Focusing on recently reported attacks though, we can see a new trend in their functionality, in which the malicious extensions do not simply aim to steal user credentials or passwords, but instead mask their malicious actions under the guise of typical user activity [1], [3], [5]. A recent example involved a cluster of scam extensions in Google's Chrome Web Store with combined installations of more than 500,000, which were able to generate fake ad clicks in an automated way, without user knowledge or consent [3]; the malicious extension works by creating the same web requests that would have been created if the advertisement was clicked by a real user. A similar but more advanced and targeted attack scenario was reported in early 2021, where a malicious extension was able to perform several actions on the Gmail accounts of its victims (including reading, forwarding, deleting, and sending e-mails) [5]. The implementation of such sophisticated types of malicious extensions makes it very challenging for web services to distinguish the legitimate actions of real users from malicious actions performed by extensions within their browser.

In this paper, we present a novel and practical framework, called WRIT (Web Request Integrity and aTtestation), that tackles exactly this problem: it enables websites to ensure that critical requests have been created via a benign control-flow execution path (i.e., the request was triggered by an actual user action), and not crafted or automatically generated by malicious third-party code or a browser extension. WRIT provides the necessary security building blocks to verify the execution flow integrity of selective code snippets at the front-end. WRIT is implemented completely in JavaScript and does not require *any* modification of the user's browser. To assess the effectiveness and feasibility of our approach, we implemented a prototype of WRIT (available in [2]). The performance evaluation results show that our approach incurs a low overhead in terms of latency and throughput.

Contributions: In summary, the contributions of our paper are:

1) We implement and provide an open source prototype of our approach (available in [2]), which is able to attest and verify the integrity of web requests and detect any abnormal or malicious requests that are not triggered by the user's actions. We further integrate our approach within Axios [9], a lightweight HTTP client API for creating web requests that is typically used in combination with many popular modern web frameworks, such as ReactJS [34] and Angular [29].
2) We implement a mechanism to trace the execution of selected JavaScript functions at run-time. These traces can be used to verify that a function has not been called from a potentially malicious action.
3) We conduct a thorough analysis to evaluate the security

properties and performance overheads of our approach. The performance evaluation results show that the latency added byWRIT to protect user actions considered as security-critical[1] (i.e., post a message) is practically negligible to the user experience (7.29 ms).

2 Malicious Browser Extensions

A malicious extension can monitor, disrupt, tamper, or block *any* incoming or outgoing traffic from and to the web service, as well as inject JavaScript code to the web page or tamper *any* JavaScript snippet sent by a web server before running on the user's browser. A browser extension can execute code —without any further interaction with the user— through background scripts or content scripts. A background script [17] runs continuously in the browser's background (as the name implies), as long as the browser is running. In this global context, a malicious extension can perform general purpose attacks (e.g., monitor all outgoing HTTP requests), as well as have access to a plethora of information regarding every open tab and leverage it to perform more targeted attacks. In addition, a malicious extension can leverage the browser API for web extensions, i.e., `executeScript` [21], in order to deploy code segments that can run as content scripts directly into one or more open tabs. Alternatively, content scripts can be declared into the extension's manifest to run in tabs matching a URL pattern (a wildcard enables injecting to all tabs) at page load by default or after the page finishes loading; either way, a malicious extension gains the ability to interact with a specific tab's page's context through content scripts. Through a malicious extension an attacker can send a web request from a user's browser using one of the following methods:

1) By synthetically crafting a web request and manually sending it to the server. The request should also include any cookies or session identifiers needed, which can easily be accessed or acquired by a malicious extension.
2) By hooking a pre-existing JavaScript function (that resides either within the DOM or is browser built-in), in which the malicious code is injected. This way, the malicious code will be executed every time the hooked function is invoked.
3) By mimicking a user action that ends up sending a seemingly benign request. For example, autocompleting a form and generating an artificial click event on the button that submits the form, instead of using a programmatic function to submit it directly, i.e., `form.submit()`. This way, the request will be sent through a normal program flow, even though it has been triggered automatically via the malicious code snippet.

3 Threat model

We assume an adversary who manages to run malicious JavaScript code at the client side either by a malicious imported third-party library or by a malicious extension installed in the victim's browser, enabling them to hijack the browsing sessions of the victim to perform specific actions (e.g., web requests or transactions) on their behalf. Such attacks have recently been reported and can be categorized in two attack vectors: (i) *click frauds*, in which malicious extensions aim to imitate their click traffic to look as benign as possible [3]; and (ii) *account hijacking*, in which the extensions perform several actions on users accounts (e.g., message posts, reading and sending of e-mails, etc), as it was in the case for Facebook [1] and Gmail [5]. The common pattern of both vectors is the unfair advantage of extensions to perform actions on users behalf within the browser, where users are already authenticated and the traffic is decrypted. This makes it very difficult for web services to distinguish the actions of a real user from the actions performed by an extension within the user's browser.

Infection. A successful infection can be achieved in many different ways: (i) by deceiving the user that the extension is legitimate, (ii) by attackers purchasing popular extensions and then updating them with malicious operations, (iii) by side-loading from a local archive, or (iv) by compromising popular extensions or JavaScript libraries and subsequently having them serve malicious code [15], [47], [53], [14].

Last but not least, it is important to note that WRIT does not aim to defend against cases where the browser extension tricks the user into performing an action through clickjacking or UI redressing attacks. Our system aims to assess the humanness of an action, regardless if this was intended or not. There are many works already dealing with such kind of user action-jacking attacks [41], [6], [51].

4 Building blocks

In this section, we describe the HTML features that we mainly use to build WRIT.

4.1 HTML5 Features

Service Workers: Service Workers [24] are non-blocking (i.e., fully asynchronous) modules that reside in the user's browser between the web page and the web server, isolated from each visited page's context. A Service Worker is registered the first time the user visits the website and runs in the background, independently from the parent webpage. Typically, when the user browses away from a website its Service Worker is paused by the browser; then it is reactivated once the parent domain is visited again. However, it is possible for the publisher of a website to keep its Service Worker alive by implementing periodic synchronization [25].

Service Workers have the ability to intercept and handle network requests originating in the parent web page; this feature allows them to be used as programmable network proxies, allowing developers to control how network requests from a web page are handled. Moreover, Service Workers cannot access the DOM directly, however they can communicate with web pages under their scope directly via the `postMessage` interface [18] or indirectly via web requests. A Service Worker can be registered using the `serviceWorkerContainer.register()` function. This function takes the URL of the Service Worker's script as an

1. WRIT operates only on marked-as-sensitive web requests, leaving the rest (such as those for getting normal web content, object fetches, and asynchronous updates) unaffected.

argument and passes it internally to the browser, where it is fetched over HTTPS. As a result, no browser extension or any in-browser entity can have access to the browser's C++ implementation that handles the Service Worker's retrieval and registration with the first-party domain [52]. Moreover, this JavaScript file can only be fetched from the first-party domain (i.e., it cannot be hosted in a CDN or any other third-party server) and cannot be registered from an iframe or third-party script.

JavaScript Closures: Functions in JavaScript can form closures [19], which combine functions with the lexical environment they are created in; the lexical environment includes any variables or other functions that share the same scope with the function forming the closure. The closure allows them to live on (i.e., to remain operational and accessible) past their original scope through the function that formed the closure. This property makes closures a powerful tool that allows us to emulate private methods, in order to regulate access to sensitive library script functions and variables.

JavaScript closures make it feasible to generate a function that contains one or more private variables that store secret or sensitive data (e.g., secret tokens, private keys, etc.). At execution time, private variables are kept hidden and cannot be accessed by any other JavaScript scope, except the calling function. Notice that this approach is safe in the presence of a malicious browser extension that tries to override native JavaScript APIs in order to extract secrets from a closure, since the tampering of original native APIs can be easily detected and then restored via an iframe [4].

4.2 Our approach: WRIT

WRIT utilizes both JavaScript closures and Service Workers to form a safe component that can verify if a request was created through a benign control flow execution path and not by any form of automation, like originating from a malicious browser extension deployed on the user's browser.

Our approach leverages the flow of a request/response as this was recently specified (and explained in this bug report [12]) across all modern browsers (e.g., Chrome, Opera, Edge, Brave, etc.). Browsers already do not allow extensions to use or interact with Service Workers (e.g., monitor their execution, inspect their variables, etc.), except for detecting their registration. Recent browser versions restrict this further by not allowing extensions to communicate with Service Workers or monitor their messages with the webpage, e.g., requests originated from background or content scripts do not pass through Service Workers.

5 System Overview

WRIT ensures that critical requests have been created via a benign control-flow execution path, and not crafted or automatically generated by a malicious browser extension or injected third-party code. This is achieved by verifying that the execution flow integrity of selective code snippets at the front-end.

WRIT is comprised of three parts, as shown in Figure 2. The first part is the JavaScript code that resides within the web page that the server wants to protect; the second is

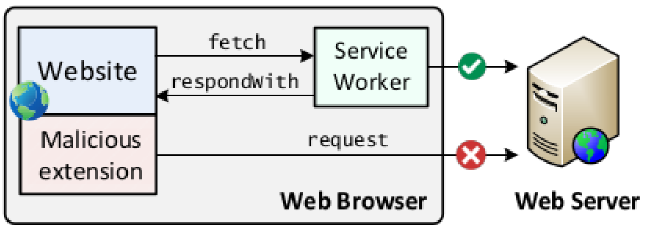

Fig. 1: The Service Worker interacts with the website, the web server and possible browser extensions. The inability of browser extensions to communicate with the Service Worker or monitor their messages with the website allows WRIT to detect requests that have been issued by a browser extension.

the Service Worker residing in the user's browser and the third is a server-side component, whose primary role is to verify that any received request is properly signed. The in-page script is the essential link that allows interaction with the page's context (i.e., JavaScript and DOM), since Service Workers cannot access the content of web pages under their registered domain by design.

5.1 Isolated environment

WRIT utilizes Service Workers in order to guarantee a persistent environment, completely isolated and protected against any possible malicious extensions that may be running on the user's browser.

Setup phase: The setup phase cannot be performed arbitrarily, but only right after the user signs in for the first time. In particular, if the authentication process succeeds, the user is redirected to a landing page where the registration of the Service Worker takes place. The setup phase occurs *only once* - after the registration, the Service Worker lives in the background and is activated automatically every time the user visits the website. We note that the registration of the Service Worker can be completed even in the presence of a malicious extension intercepting the traffic exchanged with the server, using a continuous code update technique such as those presented in [11], [16]. The user can optionally verify and inspect the Service Worker anytime, via the browser's configuration menu.

As we described in Section 4.1, the Service Worker is registered through the `serviceWorkerContainer.register()` function, which takes as input the URL of the Service Worker's script. We note that even though the corresponding JavaScript file can be fetched from the first-party domain only, its filename can be arbitrary. That aspect allows us to utilize a different one-time URL per user, that is used to fetch the Service Worker script from the server. The one-time URL consists of a unique *id* that the server provides to the client. This unique *id* should be appended as a string at the end of the URL, every time a new client requests the script of the Service Worker; the server will not respond to Service Worker requests if they do not contain a valid *id*. Also, the server will respond to Service Worker requests *only once*: the first time such a request is made. If the server receives a request with a given *id* for a second time, it will trigger an alert for an abnormal situation (i.e., a malicious extension installed in the user's browser tried to access

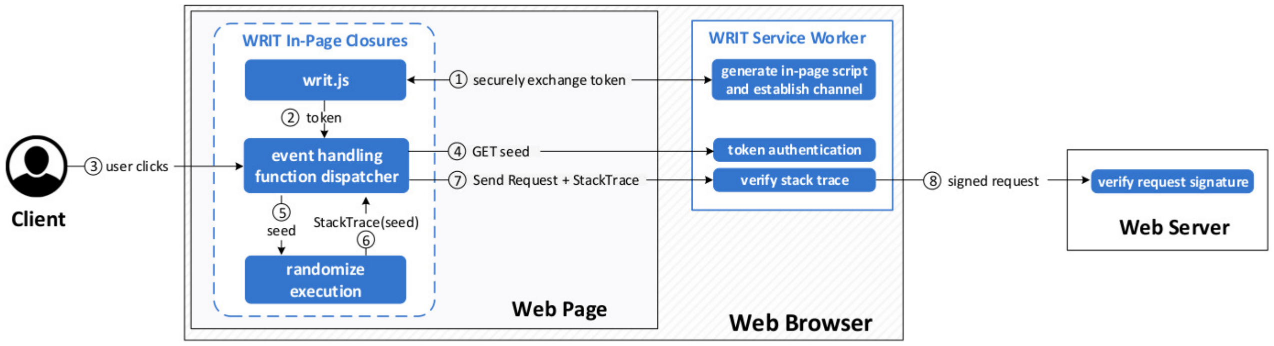

Fig. 2: High-level overview of our approach. WRIT generates a separate in-page script at each user session, which contains a unique, hard-coded, identifier. This identifier is used to establish a communication channel, where the exchanged messages can be securely authenticated. Every time the user performs an action, WRIT's in-page component uses a secret token to request a random seed from the Service Worker. The seed is used to randomize the execution of the event handling function trace, which can be verified by the Service Worker using the exact same seed; if the verification succeeds the Service Worker signs the corresponding request, using a pre-established key, and forwards it to the back-end web server.

it or has already accessed it before). In such cases, the user needs to re-run the setup phase from the beginning. The Service Worker uses this id to exchange a key k with the web server, that will be used for the signing of each request. The server associates each issued key k to the respective user that requested it and binds it internally with the corresponding session cookie. As a result, even in the case where a malicious party has managed to acquire the unique key of a real user, the incident will be detected by the server.

Attestation phase: After the Service Worker has been successfully installed on the client's browser, a safe working environment has been established; The Service Worker is running in the background and monitors every web request that is exchanged between the webpage and the server; each request is further signed using the unique key k. In particular, the Service Worker employs SubtleCrypto [26] to sign every validated request with k using one of the available options for digital signature production, e.g., HMAC with SHA512 (configurable). When the server component receives one of these requests, it can verify its integrity using its signature. The requests should also include a counter, in order to defend against replay attacks; counters are maintained on a per-user basis, so that the same count never repeats twice. If the signature generated by the server differs from the one in the received request or if there is a mismatch in the expected counter value, then the server can safely assume a malicious extension has tampered with the Service Worker's request.

We notice that even though a malicious web extension cannot access the context of the Service Worker, nor its key k that is required for the signing process of the outgoing requests, it can still unregister it: first, by getting a hold of the installed Service Worker through the API-provided `getRegistration` function and then calling an `unregister` function on it. Even in that case though, any subsequent web request received by the server will not be signed by the key k, which can in turn trigger an alert.

Updating the Service Worker: Typically the web browser will occasionally check for Service Worker updates in certain circumstances [27] and update the Service Worker's script automatically in case a new version is available. In order to prevent this default behavior that could endanger new Service Worker script versions to interception by malicious extensions, the server never updates it directly (e.g., responds with a 404). Instead, updated Service Worker code segments can be encrypted by the server using the unique key k that is stored within the Service Worker script; the latter can then use k to decrypt the update and use it as necessary.

```
(async () => {
   // the WRIT library is fetched as a string from
   // the Service Worker and NOT from the web server
   let lib_string = await fetch("/writ_lib.js");
   let writ, ua = navigator.userAgentString;
   // Function() not fully supported in Safari; use
   // eval() instead
   if (ua.indexOf("Safari") != -1 &&
           ua.indexOf("Chrome") == -1)
     writ = eval(lib_string);
   else
     writ = (new Function("return " + lib_string))();
   // reveal public library function(s) to the page
   window.WRIT = writ;
})();
```

Listing 1: Initialization of WRIT's in-page component within closures. The closure ensures that any sensitive data is kept safe against a malicious browser extension or third-party library that resides in page's scope.

5.2 In-Page Access

Given that the Service Worker can neither access the DOM of the webpage nor its JavaScript context (such as code or variables) it is necessary to also have a JavaScript component running in the page. The primary role of this in-page counterpart is to monitor the integrity of selected webpage JavaScript code, a process we discuss in detail in Section 5.3. This component needs to be protected from potential threats by malicious extensions that are described in Section 3.

A major challenge of the in-page component is how to safely initialize and execute it within the page scope, given the staggering power that browser extensions have over the DOM of the page. Obviously, it is not sufficient to request the in-page WRIT component from the web server, as a malicious extension could easily tamper its code, e.g.,

```
async function post(callback, args, e, funcs=10) {
  // check if the event was triggered manually
  if (e && e.isTrusted === false)
    return new Error("Artificial event fired!");
  // get a new seed from the SW
  let seed = await fetch("/seed",
          {body: funcs, method: "POST"});
  // run the page's protected function (callback)
  // save the request it produces & capture stack trace
  // add the randomly generated functions to the trace
  let package = gen_trace(seed, callback, args, funcs);
  // forward final trace & page's request to the SW
  return fetch("/trace",
          {body: JSON.stringify(package), method: "POST"
              });
}
```

Listing 2: WRIT's dispatch function for request sending.

using the webRequest API [28]. For instance, the Firefox browser provides the `filterResponseData()` function[2], which allows any extension to monitor and modify the body of a HTTP response before the page's DOM tree is built. To protect against such cases we follow a different approach, in which we utilize the Service Worker to serve WRIT's JavaScript code that resides in the webpage. The communication between the web page and the Service Worker (both ways) cannot be accessed by browser extensions employing webRequest monitoring [28], as such a malicious extension is not able to communicate with Service Workers, neither monitor their exchanged traffic. As such, none of WRIT's in-page scripts can be requested (e.g., due to an impersonated attack), or accessed and modified by any means (e.g., by injecting malicious code within an in-page script or extract any sensitive data by accessing the corresponding variables). The Service Worker is responsible for periodically synchronizing with the web server and acquiring any possible software updates in order to always have the last version. These updates can be performed securely, using the key that they have exchanged at the bootstrap phase (Section 5.1).

At the beginning of every session, WRIT's in-page JavaScript component is requested, as shown in Listing 1. The request is handled by the Service Worker instead of the web server. The in-page script deploys a closure that is crucial in keeping safe any sensitive or secret data (i.e., WRIT's library), as well as a unique hardcoded identifier used to provide distinguishability between WRIT's in-page counterpart and the Service Worker. This identifier is used to exchange a secret *token* generated by the Service Worker, which also provides mitigation against impersonation attacks, where a malicious extension crafts and sends messages to the Service Worker through a page-side script, in an attempt to imitate the page component and/or probe the Service Worker for information. This type of impersonation attack leverages the extension's ability to inject code within the page and generate requests, which blend in with the page's ordinary requests and become indistinguishable to a Service Worker that intercepts them. However, since it will not have the correct *token* it will fail to authenticate with the Service Worker.

2. Apart from Mozilla Firefox, the other contemporary browsers have removed this feature, mainly for performance reasons.

The *token* is stored within the secure confines of WRIT's closure. As described in Section 4.1, the closures allow us to regulate access to sensitive functions and variables. In particular, WRIT exposes a public function to the page's general JavaScript context (i.e., `window`) for other scripts to use within the DOM context, namely `post()`, that takes as argument a user-defined callback function to execute internally. The callback function contains the code necessary for crafting a web request or any other sensitive transaction and sending it to the server. As shown in Listing 2, this public function thinly wraps around inner private functions to provide other page scripts access to WRIT's core functionality, without leaking any information about their inner workings. The private scope prevents sensitive data from being accessed by a malicious extension and contains the code for mutually exchanging the secret *token* with the Service Worker. This *token* will be used to attest the execution of the user-defined callback function, as described in Section 5.3.

In addition, we follow a serve-once policy for the in-page script and also make sure that it is the first JavaScript that will be requested and executed at page load. This ensures that only the page will execute the in-page script and not an untrusted third-party entity within the page or a malicious browser extension; a malicious extension that hijacks the page loading process and tries to request the in-page script will fail, as the in-page script will have already been requested from the Service Worker. Indeed, a content script can actually execute before the in-page script or any other DOM is constructed (when runs at `document_start`), however even in that phase the corresponding request for the in-page script has already been sent and received (even though not rendered in the screen).

The procedure above, enables WRIT to achieve its primary objective in a way that is easily incorporated into any piece of existing client-sided code, while remaining as private and robust as possible in the page context where malicious extensions may operate in. In practice, this means that an attacker cannot access or change any internal WRIT library function or variable that stores critical system information. It could be the case though that a malicious extension overwrites (hooks) the library's public functions with new versions that execute malicious code before executing the original function, and vice versa. Even in that scenario though, the library can repel this attack by verifying the integrity of its public functions upon use, restoring them to their original state if necessary and optionally raising an alarm notifying the user and/or the server for malicious activity [45], [4]. Finally, the majority of the in-page WRIT JavaScript is structured around and makes heavy use of the Promise API [23], a hard requirement for many of the provided functions (e.g., fetch) and also crucial in ensuring that the system performs efficiently without blocking function calls.

5.3 Execution Sequence of Function Calls

Using the in-page closures described in Section 5.2, WRIT is capable of protecting sensitive data and the code that performs the web requests. However, the code can still be executed by third-party JavaScript or malicious extensions, by calling the corresponding public function. To protect against such cases, it is necessary to monitor the execution of these public functions, in order to detect any misbehavior or malicious actions.

We notice that a static analysis of the corresponding code snippets does not suffice, since an attacker can change JavaScript on the client at will (e.g., dynamically hook functions, inject code, etc.). Instead, it is necessary to monitor the JavaScript execution at runtime, which is challenging for several reasons. First, the execution model of modern web environments should be taken into consideration, that is based in event-driven programming. In this model, functions often execute asynchronously in response to events that are triggered by e.g., network activity or user input; second, the monitoring needs to be performed within the webpage, hence it should be implemented in a way that cannot be tampered by a malicious browser extension, either by hooking the corresponding monitoring function or by directly changing the variable(s) where the monitoring information is stored.

WRIT aims to take a snapshot of the current stack trace at a critical point of execution, typically within the `post()` function shown in Listing 2, that executes the user-defined function responsible for collecting the required parameters and then crafting the request. The main motivation for using stack traces is that by generating them within the public API functions that we want to protect, we can verify the integrity of their operation and detect if they have been called from a benign execution flow or if they are the product of a potentially malicious action.

The majority of modern web browsers provide at least one native implementation for stack trace generation, e.g., through the Error standard built-in object [20] in Firefox and Chrome. The correct function call sequence must be known in advance, so that any injection or alteration (caused by a malicious browser extension) of the sequence can be spotted by checking the function call sequence. The benign sequence can either be extracted manually by the developer (e.g., via a debugger [38]) or automatically (using a dynamic analysis tool [7]). However, the inspection of the stack trace alone as provided by the browser API is not enough, mainly because browser APIs do not provide any further details about the active stack frames (such as the program counter) besides the function names. Hence, WRIT is not able to defend against attack cases where, an adversary injects a malicious function that (intentionally) has the same name as a benign function residing in a different scope to avoid naming conflicts, or in the same scope effectively overwriting that function. As a result, the attacker would successfully spoof a naive inspection of the stack trace, simply because in both cases the stack trace will contain a seemingly innocuous sequence of function calls.

To overcome this, we further enhance WRIT with the ability to diversify the original execution trace of critical functions, by creating **a series of pseudo-randomly generated redundancy layers** in the form of empty JavaScript functions. These new functions form a chain by calling one another in a particular sequence, dictated by a $seed$ that has been used to randomly generate them. Every time WRIT needs to protect a critical function call $func$, which, e.g., sends a request to the server, it creates one such chain, appends $func$ to the tail of the chain followed by a stack trace capture. By doing so, the generated stack trace will be enriched by the sequence of newly created functions, which uniquely identifies that particular execution of the critical function. More importantly, the random functions add extra levels of differentiation and entropy in the generated stack trace, which becomes exponentially hard for an adversary to spoof or predict (due to the added randomness). Hence, if a malicious extension manages to inject an extra function within the original execution path, the resulting stack trace will not match the benign one.

The random functions are generated through `seedrandom` [10] and a $seed$ that is known only by the Service Worker and the web page, so that both parties end up generating the same functions. Every time a new function chain has to be created, WRIT's in-page counterpart requests a new $seed$ from the Service Worker. The reason we request a new seed every time is to reduce synchronization logic and the state that would otherwise be needed to be kept in the Service Worker in order to keep track of the asynchronous requests made from different parts of the web site.

We note that the communication between page and the Service Worker cannot be tampered by any browser extension, as it is securely performed via the in-page script, as we describe in more detail in Section 5.5. Once the $seed$ has arrived safely inside WRIT's in-page component, malicious extensions are unable to read the $seed$ from within it because, as we described in Section 5.2 extensions do not have access to WRIT's interior scope. The $seed$ is then used to generate a sequence of random numbers, each of which is appended to the name of a newly created dummy function. Each function contains a single instruction, which is set to invoke another function that has been assigned the next number of the sequence. The last function of the sequence is set to invoke the real function $func$ that WRIT wants to protect instead, which is hooked to produce a stack trace before it runs. This approach allows to control the randomization levels by adjusting the number of the resulted permutations $P(n, r) = n!/(n-r)!$, where n is the number of different function names and r is the number of random functions that we actually use each time. As we will see in Section 7, choosing a number for r between 10 and 100 incurs minor performance overhead while the probability of guessing the correct permutation is in the order of 10^{-18} for e.g., $r = 20$.

At this point the chain is formed and critical function $func$ can be run by the page's client-sided code normally, setting off the chain reaction WRIT planted. After executing $func$ successfully, page code can use a library function to send the stack trace that was produced to the Service Worker, who will in turn use the same $seed$ to generate a sequence of numbers and compare it against the sequence contained in the random functions previously added to the stack trace. The Service Worker's pivotal role must be stressed for this part of the process: validation is performed within the Service Worker, where no extension can observe or hinder it. If the sequences are identical, execution of $func$ was successful without complications incurred by malicious extension intervention. In case of sequence mismatch, it is assumed that either the code segments have not executed as expected or have been tweaked by an extension, resulting in validation failure. In either case of success or failure, the Service Worker is configured to remove the random functions from the stack trace, sign the remainder and send it to the end server including a single bit indicating the outcome of validation. That is very useful in case of failure where a server would want to know something went wrong, but it could also prove useful in case of success as an indication that the Service Worker is still functional. This feature could be baked into the Service Worker as a more secure approach, assuming that a clever malicious extension could entirely block the Service Worker's signed requests to the server. It would also yield a small performance improvement by minimizing the network overhead imposed due to these (potentially numerous) requests to the end server.

5.4 Distinguishing Event Origins

The stack trace monitoring methodology described in Section 5.3, allows us to track conformant program execution and verify if a request has been generated through a benign control flow path (e.g., when a user clicks a button). We notice though that if a malicious extension tries to generate the `MouseEvent` [22] that triggers the critical function $func$, our methodology would not be able to distinguish if it is a benign request created by the user or a request that has been artificially created by a malicious extension. The reason for that is that the resulting stack trace will neither contain any non-benign function, nor any non-expected execution flow; since the event has been triggered asynchronously by a different function, its call will not be included in the same stack trace with the function that has been registered by the website to handle the event. Instead, the stack trace will be the same as if it had been produced by a legitimate user action.

In many cases, in order to successfully complete the generation of a web request, client-sided code would also require the completion of any other required action (e.g., input text in a text form, selection of an item from a menu list, etc.); however a malicious extension can still perform such actions through the corresponding DOM elements a priori, without revealing any of these actions in the stack trace that is captured by WRIT. To protect against such actions, we need to distinguish between human and non-human action triggering events on DOM elements, e.g., click events. Typically, the metadata that are included on each generated event include several properties, whose values are different between human and non-human DOM element interaction (i.e., their `isTrusted` field has negative value, while the corresponding mouse position coordinates have zero value in case of a JavaScript-triggered event). In addition, the browser protects these metadata implicitly, by restricting their access to read-only permissions. Therefore, by checking the value of a combination of properties of a triggered event, WRIT can distinguish the source of DOM element interaction. These checks are performed within the private scope of WRIT's closures, which described in Section 5.2.

5.5 Out-of-band Communications

We define two distinct communication channels in WRIT, as shown in Figure 2. One lies between the web page and the Service Worker, and the other between the Service Worker and the web server. In the following two sections, we discuss in detail the role each channel plays in the context of WRIT's operation.

Communication between the web page and the Service Worker: As we describe in Section 4, the Service Worker has the ability to intercept and handle any request originating from the web page. In WRIT, we use custom URL requests in order to distinguish between normal HTTP traffic and requests that are meant for internal communication between the JavaScript component that runs within the page and the Service Worker. There are several unique URLs, each linked to a specific page script operation that either needs to request input from or send output to the Service Worker. The most significant operations are: (a) sending the id to the Service Worker (after it has been installed, during initial setup), (b) requesting a new $seed$ from the Service Worker for stack trace generation and (c) sending a newly generated stack trace to the Service Worker for validation. The Service Worker is aware that these specific requests are meant for internal communication and responds back to the page accordingly without implicating the server. To mitigate against page-side impersonation attacks and guarantee secure communication between the page and the Service Worker, we authenticate the exchanged messages using the $token$ that is exchanged between the Service Worker and WRIT's in-page component (see Section 5.2).

Communication between the Service Worker and the Web Server: The communication between the Service Worker and the web server is taking place over the network, as such it is susceptible to monitoring, interception and even blocking by malicious extensions through the webRequest API [28]. To overcome this threat scenario, we force the Service Worker to explicitly sign every protected request, so the server can verify them and safely detect if a malicious extension has tampered with the Service Worker's request. In particular, the Service Worker validates that a protected request is benign (using the procedure that is described in Section 5.3) and then signs it using the secret key k that is stored within the Service Worker's isolated environment, provided by the server during the initial setup (see Section 5.1).

By doing so, the server can verify them and safely detect if a signed request has been tampered by a malicious extension or not. We note though, that a malicious extension has the power to completely block a request (as well as completely un-registering the Service Worker) and disrupt the normal operation of the end service. To detect such attacks, WRIT can be configured to use periodic heartbeats hat are exchanged between the Service Worker and the server (signed by the secret key k). However, it would still not be easy to distinguish between cases that a user went offline due to a legitimate but unfortunate event (e.g., system crash, network failure, etc.) or due to a malicious action. As the main purpose of WRIT is only to attest the integrity of web requests, providing mitigation against these attacks is out of the scope of this work.

5.6 End-to-End Example

In this section we present a complete, step-by-step example scenario of a user visiting a website that uses WRIT. The

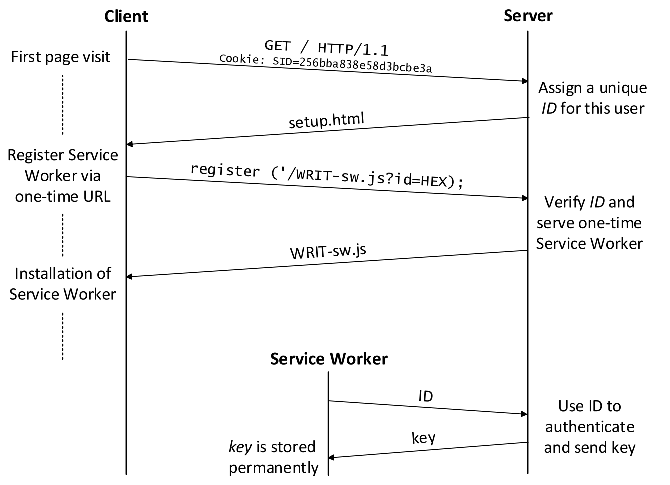
Client
Server
First page visit
GET / HTTP/1.1
Cookie: SID=256bba838e58d3bcbe3a
Assign a unique ID for this user
setup.html
Register Service Worker via one-time URL
register ('/WRIT-sw.js?id=HEX);
Verify ID and serve one-time Service Worker
WRIT-sw.js
Installation of Service Worker
Service Worker
ID
Use ID to authenticate and send key
key
key is stored permanently

Fig. 3: The initial setup of WRIT. The setup needs to run *only once*, typically when the user visits the website for the first time.

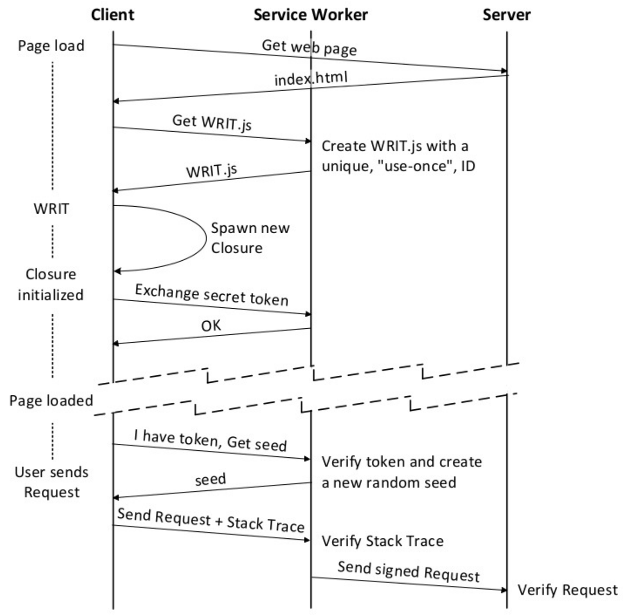

Fig. 4: The messages exchanged during a user's session with WRIT.

corresponding steps are outlined in Figure 3 and Figure 4. If this is the first time that the user visits the website, WRIT initiates the setup phase from a separate webpage (Figure 3); In particular, this webpage (i) asks for a randomly generated unique id from the server, (ii) it subsequently uses id to fetch the Service Worker via a one-time URL, (iii) then runs `register()` to install the new Service Worker, and finally, (iv) uses id to exchange a secret key k with the web server that keeps safe in a local variable.

As described in Section 5.1, the setup phase needs to run only once. To ensure a safe installation of WRIT's Service Worker, the setup needs to run from a clean environment (typically by temporarily disabling any browser extension that is installed in the user's browser). After that, the Service Worker has been registered and lives in the background persistently, even after browser reboots. It is also re-enabled automatically every time the user visits the website. As shown in Figure 4, on every subsequent user visit to the website, WRIT's page component first needs to authenticate the communication channel between itself and the Service Worker. It fetches a new in-page script and then it sends its hardcoded, unique id to the Service Worker, which is then used between the page closure and the Service Worker to exchange a secret token.

After this point every time the user interacts with an element and triggers one of these events, (e.g., the user clicks on a button) the following actions take place before the original event handling function $func$: (i) a random $seed$ is obtained from the Service Worker and is used to generate a sequence of functions forming a call chain with $func$, (ii) the chain is invoked and a stack trace then captures the random function sequence and $func$, (iii) the trace is sent to the Service Worker for validation of the random function sequence generated in the page against the one generated in the Service Worker using the same $seed$, (iv) lastly the Service Worker signs the request and sends it to the server.

6 Implementation

We developed a prototype implementation of WRIT, available at [2], in order to evaluate our approach and demonstrate its feasibility. Listing 3 presents a code snippet that shows how WRIT can be used to attest the requests that originate from an input element of the web page. As we can see, the developer only needs to provide a function that is responsible for getting the data from the corresponding DOM element(s) and crafting the request that needs to be sent to the web server. This function is passed as an argument to the `post()` function together with the user event that triggered this action. We also integrate our approach into Axios version v0.19.2; Axios [9] is a lightweight HTTP client API for creating web requests. We chose Axios as it offers many desirable features: (i) it offers a user friendly API for creating web requests, on top of the `XMLHttpRequest` API, (ii) it has become very popular in web development and is typically used in combination with the majority of modern web frameworks, such as ReactJS,Angular,etc, and (iii) it is compatible with most modern browsers.

The changes needed to integrate WRIT within `axios.js` required about five lines of code in the `dispatchXhrRequest()` function, which is responsible for creating and sending a customized `XMLHttpRequest` according to user input, passed via a `config` object. The `config` object specifies, among others, the request's method type (GET or POST) and includes any request parameters or body content. Axios performs several tasks and checks after the creation of a new request inside `dispatchXhrRequest()`. The last task before the request is sent, is to check whether or not a "cancel token" is present in the supplied `config` object; if one is found, then the request is cancelled as per user request. We enrich that check with the result of an invocation to our WRIT public function which produces an embellished stack trace that must be verified by the currently operating Service Worker. If the latter verifies the trace successfully, Axios' request is

```
<body>
  <textarea id="tx">Hello World!</textarea>
  <script>
    function create_request() {
      let text = document.getElementById("tx").value;
      let config = {method: "POST", body: text};
      return {path: "/some_path", config};
    }
  </script>
  <button onclick='WRIT.post(create_request, e=event);'>
    Send POST via WRIT
  </button>
</body>
```

Listing 3: Protected POST via WRIT.

```
<body>
  <textarea id="tx">Hello World!</textarea>
  <script>
    function create_request() {
      let text = document.getElementById("tx").value;
      let config = {method: "POST", body: text};
      return {path: "/some_path", config};
    }
  </script>
  <button onclick='axios.post("", event, create_request)
      ;'>
    Send POST via WRIT-enabled Axios
  </button>
</body>
```

Listing 4: Protected POST via WRIT-enabled Axios.

sent, otherwise it is cancelled. The modified, WRIT-enabled, Axios library is served to the client through our Service Worker, as described in Section 5.2.

Listing 4 shows an example of how the WRIT-enabled Axios can be used to attest user requests *transparently*, even for legacy web applications that already utilize the Axios library.

7 System Evaluation

In this section, we evaluate our proposed architecture in terms of performance and security. Our base setup consists of two different machines: one server that hosts a simple web site and one machine that acts as a web client. The server is equipped with an AMD R5-3600 and 16GB of RAM, the client is equipped with an Intel I5-6300U and 8GB of RAM. The two machines are connected over a 1 GbE connection, which we shape accordingly to evaluate how the performance of WRIT scales on different network environments that represent different kind of web users, such as 3G, 4G, and LAN connections. The server that is used to host our website is running Flask v1.1.1 and uses Python v3.6.8. We have also configured our server so that it can serve over HTTPS, a requirement for the Service Worker API to expose itself in the page's context and become available to the client.

Browser Compatibility: Our approach is based on HTML5 components, such as Service Workers, which are supported by the majority of popular web browsers (such as Chrome, Firefox, Opera, Edge, and Safari), as well as by many mobile devices (including Samsung Internet, Chrome Android, iOS Safari, and Firefox Android). A complete list of compatible browsers can be found in [48].

7.1 Performance Evaluation

We now evaluate WRIT in terms of performance. In particular, we measure the added overhead of WRIT in two scenarios: one under different network connection types and one in which the client increases the number of functions that are used for stack generation. For each scenario, we measure the time needed to perform a simple POST request on top of WRIT with all of its security mechanisms enabled and we compare it with the vanilla case (in which the request is simply sent to the server, with and without a Service Worker installed). For all our experiments we use Chrome with caching manually disabled. We also break WRIT's overhead down to four key components, listed in order of occurrence: the request to the Service Worker for a new seed, the generation of the stack trace in WRIT's closure, the processing and signing that takes place in the Service Worker and finally the signed request that is sent to the server.

Figure 5a shows the end-to-end time of a POST request in the vanilla case versus the case where WRIT is enabled. We also plot a case, where we place an empty Service Worker in the vanilla setup to show the overhead added by the Service Worker alone. As we can see, an empty Service Worker adds an overhead of about 2.97 ms on average (1.6 ms in the case of LAN). On top of that, we find that WRIT adds an additional overhead of as low as 5.69 ms in the case of a typical LAN setup (or 13.63 ms in the case of wifi). As a consequence, the overall end-to-end latency that WRIT adds to protect a sensitive POST request is as low as 7.29 ms.

In Figure 5b, we break down the above overhead across all network presets and we see that latency is clearly affected by the time needed to send the final request to the server. This was expected as it is the only network-bound operation. In contrast, given that WRIT runs with a baseline of 10 added stack functions in this scenario, the three remaining components remain nearly constant, within the margin of error. Overall, WRIT's network overhead increases as network conditions degrade across the different presets.

This limitation becomes even more evident as we move on to our second testing scenario, where by increasing the number of functions added to the stack trace, the trace's size directly increases along with the final request's body size. For clarity, the request's body size starts at 1.3 KB (when using 5 functions), that first grows to 1.8 KB with 10 functions, then to 6 KB with 50 functions, and finally reaches 11.3 MB with 100 functions. The results of this scenario are shown in Figure 5c, illustrating how end-to-end request latency is affected by the number of functions in the stack trace, in the LAN preset. What we see is that WRIT's baseline of 10 added functions can be easily expanded to 50 and 100 functions for a 0.7 ms and 1.5 ms latency increase respectively. Alternatively, halving the number of functions to 5 yields a 1 ms latency decrease. The number of extra functions directly affects the related WRIT components as shown in Figure 5d. The stack trace generation itself grows from 0.18 ms latency using 10 functions, to 0.53 ms in case of 50 functions, and 0.68 ms in case of 100 functions. By reducing down to 5 functions from 10, we get a latency of 0.24 ms, which is however within our margin of error

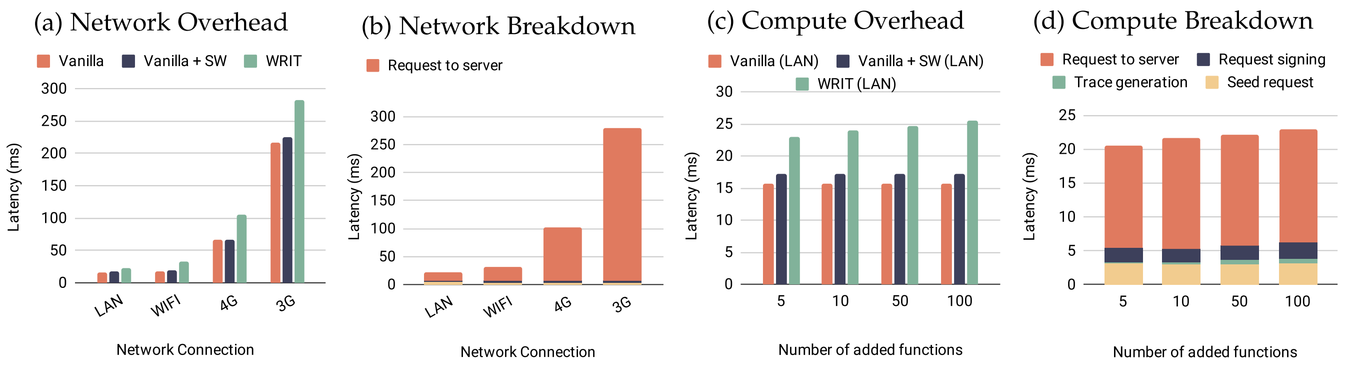

Fig. 5: WRIT's performance under different network conditions and different added stack function quantities.

(0.1 ms). The Service Worker's seed generation procedure sees minor increases, taking up to 3 ms latency at both 10 and 100 added functions. Likewise, the signing procedure starts at 1.96 ms latency with 10 functions and peaks at 2.37 ms with 100 functions. As expected, the final request to the server remains constant at about 15-16 ms latency throughout all four function tiers. Finally, there is a constant 2.5 ms latency not accounted for in the plot, which is composed of miscellaneous Service Worker and browser tasks that are out of our control.

Discussion on UI/UX: WRIT increases the latency of the web requests that have been issued explicitly through its API by 5-57 ms depending on the network and browser conditions. We note though that this increase affects only security-critical web operations (e.g., submitting a form), which are usually generated through human-triggered actions: the effects in responsiveness are negligible, as the added latency remains well below the limits of having the user feel that the system is reacting instantaneously [50]. Finally, the latency of any other web request (such as those for getting normal web content, object fetches and asynchronous updates) is not affected by WRIT, hence neither the corresponding user-experience.

7.2 Security Analysis

We now evaluate the security properties of our proposed design by describing possible threat scenarios or attacks, and showing how WRIT protects against them. For many of the attacks described below, we have implemented custom browser extensions. We also utilize the browser built-in developer tools and the debugger to get more insights and low-level operational details.

7.2.1 Tampering WRIT Service Worker

As described in Section 4.2, browser extensions cannot interact with deployed Service Workers (e.g., monitor their execution, inspect their variables, etc.). That means that they cannot access the secret key k that is used for the attestation between the Service Worker and the end web server. In addition, the Service Worker script with the accompanied key is served *only-once*. Even though a malicious browser extension can request the corresponding JavaScript file after the user or access it by other means (e.g., through a legitimate sign process), it will not obtain the targetted user's key. On the other hand, requesting it before the user is not possible because the Service Worker is registered within a clean environment, typically when the user signs in for the first time as described in Section 5.1. Any future updates are initiated only within its script instead of replacing it entirely with a new one and are always signed with the corresponding key. Furthermore the Service Worker can be served over HTTPS only, hence its script is also well protected against MITM attacks.

Finally, a malicious extension that has been successfully installed on the user's browser can unregister WRIT's Service Worker. This can happen by getting a hold of the installed Service Worker through the API-provided `getRegistration` function and then calling the `unregister` function on it. Even in that case though, WRIT ensures that no malicious requests will be performed on the user's behalf, as any subsequent requests received by the server will not be signed by the key k, which in turn trigger an alert.

7.2.2 Tampering WRIT's in-page component

Similar to Service Workers a browser extension cannot interact with a closure, neither interfere with their execution, access their context or the data stored within. However, contrary to the Service Worker which is permanent after its first registration, a closure is instantiated every time the user visits the web page. The most critical part is when the in-page script is fetched from the server, since browser extensions can hook the web page in various ways in order to tamper with the content that is received from the web server. As described in Section 3, browser extensions can deploy malicious code within the visited web page's context either directly through a content script or indirectly through a call to `executeScript` from the background script. Besides that, they can also utilize JavaScript code that is available in the page's context, just like any regular script fetched from the server, including WRIT's in-page script.

To protect the in-page script from tampering of its code when fetched from the server, we follow a different approach: instead of having the web server serve the script, we assign this task to our trusted Service Worker. By doing so, the transfer is invisible to any browser extension, thus any malicious action on it is abolished. Moreover, it eliminates the possibility of a browser extension requesting the script on the user's behalf from the end server and using it to craft non-benign requests. Since browser extensions cannot communicate with the Service Worker, a background script cannot obtain a valid in-page script, neither a content script.

Furthermore, any critical data in WRIT, i.e., the in-page script's id, the key, $seed$s and $stack\ trace$s, are kept private within appropriate closures, as described in Section 5.2. WRIT's library exposes only a set of public functions in the page's global scope, which must be used to interact with WRIT. A malicious extension could overwrite (hook) the library's public functions with new versions that execute malicious code before executing the original function and vice versa. The library can repel this attack by verifying the integrity of its public functions upon use, restoring them to their original state if necessary and optionally raising an alarm notifying the user and/or the server for malicious activity. Alternatively, if a malicious extension prints them to probe for information, each public function simply reveals a call to the corresponding private library function, which is inaccessible due to the library's closure. If they are outright deleted, there is no repercussion to the library's operation beyond denial of access to the library for other (benign) page scripts, that can then re-fetch a fresh copy of the library's script (from the Service Worker).

7.2.3 Traffic Monitoring

Browser extensions can observe, intercept and modify the requests and the responses that are exchanged between the client and the end web servers via the `webRequest`, the `devtools.network` and the Chrome-only `chrome.debugger` APIs. All these three options grant extensions different permissions and they are required to claim them accordingly in their manifests.

After experimentation, we have discovered that the `webRequest` API in Chrome does not provide any means of accessing the full responses, but only the headers. Even though this can be helpful (since it decreases the risk network monitoring posed by extensions) it is not the case for Firefox, which provides the `filterResponseData()` function in `webRequest` API. This function allows any extension to monitor and modify the body of a HTTP response received by the web server. WRIT is not vulnerable to this feature though, as it uses the Service Worker to serve any sensitive JavaScript code and data at the beginning of each session. Any communication between the web page and the Service Worker cannot be accessed by browser extensions employing `webRequest`, as such a malicious extension is not able to monitor their exchanged traffic (both ways). Moreover, the Service Worker uses a secret key that has been securely exchanged with the web server, as described in Section 5. This key is able to sign each and every request that enters the network and as such, can sufficiently protect it against any kind of tampering or spoofing.

Besides `webRequest`, many popular browsers like Chrome and Firefox, offer `devtools` APIs that grant the extensions extra capabilities (such as access to the console, network and performance tabs). These capabilities are typically available only in the developer tools panel and once obtained they allow a browser extension to monitor and control different aspects of the browser, originally only intended for development and debugging. However, extensions claiming the `devtools.network` permission can gain access to the respective API only while the browser's developer tools panel is open, due to the fact that the API is exposed only to very specific pages and scripts that are used to implement new tabs for the panel. Albeit very powerful, the danger it poses is severely limited as any user is bound to notice that their devtools panel opens randomly (assuming extensions are or become capable of programmatically opening the panel, a task we have been unable to achieve so far). In case an extension attempts to trick the user into opening the panel manually, many websites and popular web applications have been inserting warnings in the devtools console that should help mitigate self-hacking (e.g., Facebook).

Similarly, the Chrome-specific `chrome.debugger` API provides extensions with functions that have elevated access to a visited page's network activity, DOM elements and script execution. In essence, it provides elevated access to response (and request) handling, in turn allowing the inspection and modification of response bodies. Extensions claiming the `chrome.debugger` permission can attach Chrome's debugger to an open tab, causing the browser to display a screen-wide banner below the user's bookmark bar, indicating that a named extension is debugging the browser, accompanied by a button to cancel its operation. This visual indication is even stronger, alarming any user to the possibility of their browser being compromised, especially after an extension's installation. There are also JS antidebugging tricks [36] that we can leverage in order to detect the presence of the debugger and act accordingly.

7.2.4 Replay attacks

We defend against the replay of signed requests by including a non-repeating count in every request (Section 5.1. Each request is signed using the string representation of the request, combined with the value of the current count. A malicious extension that is eavesdropping benign requests cannot replay them in the future, as the server will notice the repetition of an old count. However, a malicious extension could block a specific signed request from reaching the web server and storing it locally; this signed request could be sent later on to the server successfully. Even though we are not aware —to the best of our knowledge— of a real-world scenario that can be exploited from this hiccup, this can still be fixed by further applying one-time passwords or even timestamps to the signing process.

7.2.5 Blocking web requests and responses

A malicious extension has the ability to block any network from and to the web service. To detect such types of attacks, we could configure the Service Worker to periodically send heartbeats (signed by the secret key k). However, it would still not be easy to distinguish between cases that a user went offline due to a legitimate but unfortunate event (e.g., system crash, network failure, etc.) or due to a malicious action. In a case where a malicious extension selectively blocks the web traffic flowing to the website, the heartbeats would operate successfully. Even though WRIT is not able to provide further protection to the already infected client, it can still ensure that no malicious requests will be performed on the user's behalf. Overall, the main purpose of WRIT is only to attest the integrity of web requests, defending against these attacks is out of the scope of this work.

8 Discussion

Portability: The HTML5 features (such as Service Workers) that are required by WRIT to operate correctly are currently supported by the vast majority of browsers, both in desktop and mobile devices [3]. In addition, WRIT has been integrated within Axios [9], a popular lightweight HTTP client API for creating web requests that is typically used in combination with many modern web frameworks, such as ReactJS and Angular.

Deployment: To protect against unauthorized execution of JavaScript functions, WRIT monitors the stack trace and verifies that they have not been called from a potentially malicious action. This requires that the correct function call sequence is known in advance. As we described in Section 5.3, such sequences can be either extracted manually by the developer or even automatically, using dynamic analysis tools.

We note that the operation described above is vital for WRIT to operate correctly. If not done properly, it can result to either false negatives (e.g., in the case a security-critical function is not monitored) or false positives (e.g., the benign function call sequence has not be obtained correctly). Even though we believe that the generation of the benign function call sequences is a reasonable assumption, we plan to explore mechanisms to provide further assistance as part of our future work.

9 Related Work

Malicious browser extensions. Malicious extensions have always been used over time to infect end-user web browsers and exfiltrate sensitive data or perform malicious actions on users' behalf, sometimes installed by millions of users [46], [42]. Kapravelos et al. [46] analyzed 48K extensions from the Chrome Web store and identified several large classes of malicious behavior, including affiliate fraud, credential theft, ad injection or replacement, and social network abuse. In [42] the authors analyze multiple browser extensions and indentify a set of 9,523 malicious ones. By using both static and dynamic analysis techniques, they show that extensions typically abuse `contentScript` permissions to perform malicious activities, such as Facebook hijacking, ad injection, search leakage and user tracking. To overcome this problem, many approaches propose to analyse the extensions offline and try to detect any malicious behavior. For example, in [58], the authors develop a deep learning framework that detects malicious JavaScript code, with an accuracy of about 94%. Even though such frameworks can provide high accuracy, they always need to feed on new data in order to keep the accuracy high, as new obfuscation/malicious JavaScript techniques are developed.

Malicious scripts in page context. Besides malicious extensions, abnormal behavior can originate from third party and web applications as well. In [30], authors characterize a dataset of 4.4 million public posts and identified 11,217 malicious posts that most of them originated from third party web applications. As a countermeasure, they propose an extensive feature set based on entity profile, textual content, metadata, and URL features in an attempt to automatically identify malicious content on Facebook. Such server-side anomaly detection approaches though, no matter how sophisticated they become, cannot provide a full solution for this problem. Instead, extra security mechanisms should be deployed at the client-side, which will ensure a trusted behavior on the user's side.

Web page integrity. In [60], authors propose ZigZag to strengthen JavaScript-based web applications against client-side validation attacks. Ripley [57] tries to tackle the problem of untrusted clients by automatically replicating the execution of client-side JavaScript on a trusted server tier, thus preserving the integrity of a distributed computation. However, Ripley imposes network and memory overhead, as it transfers and replays every client event to the server. Web Tripwires [54] is client-side JavaScript code that can detect most in-flight modifications to a web page, and prevent changes in the received changes (e.g., popup blocking scripts, advertisements, malicious code that can cause harm). Similary, Glasstube [40] uses a lightweight approach that protects the integrity of web applications against network MITM attacks (such as session hijacking, reordering and replay attacks). However, with the great adoption of the HTTPS in the Internet nowadays, such techniques have become outdated.

Recent approaches focus on utilizing software-only solutions, that are also based on open standards, such as HTML5. DOMtegrity [56] is an approach that ensures the integrity of the web content in the client's browser, by using client-side code to verify that the DOM structure has not been altered by a malicious browser extension. However, malicious extensions or third party applications can still perform malicious actions, such as crafting requests on a user's behalf. WRIT is able to attest all requests and detect any abnormal or malicious requests that have not been created with the user's consent. Caja [37] is a tool that allows to safely embed third-party HTML, CSS and JavaScript in a website and enables developers to control the permissions of code over user's data. However, a browser extension can still perform malicious actions on users' behalf. WRIT is able to further protect against malicious extensions, as well as third-party libraries and code.

Requests integrity and attestation. Previous works use a trusted execution environment to certify user requests. The trusted environment can be hardware-based, a VM, or within the browser. NAB [39] uses a small hardware-based trusted software component to approximately certify human-generated activity based on the elapsed time since the last legitimate keyboard or mouse activity. However, NAB is not able to link these activities to the exact request semantics, hence it is not possible to detect forged requests that are transmitted after a benign keyboard or mouse activity, neither to to handle asynchronous operations (e.g., an email queued to be sent later). More recent works, such as VButton [49], Fidelius [33], and ProtectIOn [31], utilize hardware-based mechanisms (e.g., Intel SGX, ARM TrustZone). Even though such methods provide strong security guarantees, they have specific hardware requirements, which are avaiable only on specialized or custom setups. In addition, they also require browser support, which might

3. https://caniuse.com/serviceworkers

Approach	No Hardware support	No OS support	No Browser support	No Extensions support
NAB [39]	-	✓	-	✓
VButton [49]	-	-	-	✓
Fidelius [33]	-	-	-	✓
ProtectIOn [31]	-	-	-	✓
Gyrus [43]	✓	-	✓	-
WRIT	✓	✓	✓	✓

TABLE 1: Comparing WRIT with state-of-the-art approaches for verifying the integrity and attestation of user requests.

not always be feasible or practical. Gyrus [43] is able to handle such cases by capturing more semantics of the user's intent. In particular, it permits requests only when they are attributed to text-based user input. However, it uses a VM as an isolated environment to securely acquire the on-screen user inputs and match them with outgoing requests, which impose high deployment overhead.

Table 1 provides a qualitative analysis of WRIT with previous works. In contrast with all previous works, WRIT establishes a trusted environment using software mechanisms, without requiring any hardware or browser support.

10 Conclusions

The current state-of-the-art shows a significant lack of client-based integrity and attestation techniques. In this paper, we presented WRIT, as a first step of a new line of security mechanisms: a lightweight framework that is capable of verifying the integrity of web requests and ensuring that they have been created through a benign control flow execution path (e.g., when a user performs a request through a specific button), and not generated by any third-party JavaScript code nor a malicious browser extension. WRIT is immediately applicable as it is implemented solely using HTML5 features that are available across all modern browsers without requiring any browser modifications or extensions. Developers can utilize WRIT's API to protect security-critical web requests at a practically negligible cost of about 7.29 ms latency.

Acknowledgements

The authors would like to thank the editor-in-chief, associate editor, and reviewers for their valuable comments and suggestions. This work was supported by the National Science Foundation under the grant CNS-2047260, and by the projects dAIEdge and SecOPERA funded by the European Commission under Grant Agreements No. 101120726 and No. 101070599.

References

[1] Trojan:JS/Kilim is a family of malicious browser extensions that post unauthorized content to the user's Facebook Wall. https://www.f-secure.com/v-descs/trojan_js_kilim.shtml.

[2] WRIT Project. https://anonymous.4open.science/r/WRIT-1437/.

[3] Malicious chrome extensions enable criminals to impact half a million users and global businesses. https://atr-blog.gigamon.com/2018/01/18/malicious-chrome-extensions-enable-criminals-to-impact-half-a-million-users-and-global-businesses/, 2018.

[4] Chromium issue 793217: "document_start" hook on child frames should fire before control is returned to the parent frame. https://bugs.chromium.org/p/chromium/issues/detail?id=793217, 2019.

[5] Ta413 leverages new friarfox browser extension to target the gmail accounts of global tibetan organizations. https://www.proofpoint.com/us/blog/threat-insight/ta413-leverages-new-friarfox-browser-extension-target-gmail-accounts-global, 2021.

[6] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, USENIX WOOT, 2014.

[7] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Comput. Surv.*, 50(5), September 2017.

[8] Sajjad Arshad, Amin Kharraz, and William Robertson. Identifying Extension-based Ad Injection via Fine-grained Web Content Provenance. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID, 2016.

[9] Axios. The axios project. https://github.com/axios/axios.

[10] David Bau. Javascript random number generator "seedrandom" repository. https://github.com/davidbau/seedrandom.

[11] Mariano Ceccato and Paolo Tonella. CodeBender: Remote Software Protection Using Orthogonal Replacement. *IEEE Software*, 28(2):28–34, 2011.

[12] Chromium Bugs. Figure out how Service Worker and Web Request API should interact. https://bugs.chromium.org/p/chromium/issues/detail?id=766433.

[13] Catalin Cimpanu. Google removes 500+ malicious Chrome extensions from the Web Store. https://www.zdnet.com/article/google-removes-500-malicious-chrome-extensions-from-the-web-store/.

[14] Catalin Cimpanu. "Particle" Chrome Extension Sold to New Dev Who Immediately Turns It Into Adware. https://www.bleepingcomputer.com/news/security/-particle-chrome-extension-sold-to-new-dev-who-immediately-turns-it-into-adware/, 2017.

[15] Tomer Cohen. Game of Chromes:Owning the Web with Zombie Chrome Extensions. https://www.blackhat.com/docs/us-17/thursday/us-17-Cohen-Game-Of-Chromes-Owning-The-Web-With-Zombie-Chrome-Extensions-wp.pdf.

[16] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed Application Tamper Detection via Continuous Software Updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12. Association for Computing Machinery, 2012.

[17] MDN contributors. Background. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/background.

[18] MDN contributors. Client.postmessage(). https://developer.mozilla.org/en-US/docs/Web/API/Client/postMessage.

[19] MDN contributors. Closures. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures.

[20] MDN contributors. Error. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error.

[21] MDN contributors. Javascript apis. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API.

[22] MDN contributors. Mouseevent. https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent.

[23] MDN contributors. Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[24] MDN contributors. Service worker api. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[25] MDN contributors. Serviceworkerglobalscope.skipwaiting(). https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/skipWaiting.

[26] MDN contributors. Subtlecrypto. https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto.

[27] MDN contributors. Updating the service worker. https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle#updates.

[28] MDN contributors. webrequest. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest.

[29] Google Developers. Angular: One framework. mobile & desktop. https://angular.io.

[30] Prateek Dewan and Ponnurangam Kumaraguru. Towards automatic real time identification of malicious posts on Facebook. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 85–92. IEEE, 2015.

[31] Aritra Dhar, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. ProtectIOn: Root-of-trust for IO in compromised platforms. *Cryptology ePrint Archive*, 2019.
[32] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *2009 Annual Computer Security Applications Conference*, ACSAC, 2009.
[33] Saba Eskandarian, Jonathan Cogan, Sawyer Birnbaum, Peh Chang Wei Brandon, Dillon Franke, Forest Fraser, Gaspar Garcia, Eric Gong, Hung T Nguyen, Taresh K Sethi, et al. Fidelius: Protecting user secrets from compromised browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280. IEEE, 2019.
[34] Facebook Developers. React: A javascript library for building user interfaces. https://reactjs.org.
[35] Nicholas Fearn. Nearly 80 chrome extensions caught spying – how to protect yourself. https://www.tomsguide.com/news/chrome-extension-spyware, 2020.
[36] Juan Manuel Fernández. Javascript antidebugging tricks. https://x-c3ll.github.io/posts/javascript-antidebugging/, 2020.
[37] Gogle Develoopers. Caja Project. https://developers.google.com/caja/.
[38] Google Developers. chrome.debugger. https://developer.chrome.com/extensions/debugger.
[39] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks. In *NSDI*, pages 307–320, 2009.
[40] Per A. Hallgren, Daniel T. Mauritzson, and Andrei Sabelfeld. GlassTube: A Lightweight Approach to Web Application Integrity. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ACM PLAS, 2013.
[41] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, USENIX Security, 2012.
[42] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and Lessons from Three Years Fighting Malicious Extensions. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security, 2015.
[43] Yeongjin Jang, Simon P Chung, Bryan D Payne, and Wenke Lee. Gyrus: A Framework for User-Intent Monitoring of Text-based Networked Applications. In *NDSS*, 2014.
[44] Richi Jennings. Chrome web store fail: 300+ more scam browser extensions. https://securityboulevard.com/2020/08/chrome-web-store-fail-300-more-scam-browser-extensions/, 2020.
[45] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference*, ACM IMC, 2019.
[46] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX Security, 2014.
[47] Maxime Kjaer. Malware in the browser: how you might get hacked by a chrome extension. https://kjaer.io/extension-malware/, 2016.
[48] Alexis Deveria Lennart Schoors. Can i use service workers? https://caniuse.com/#feat=serviceworkers.
[49] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. VButton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, pages 28–40, 2018.
[50] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.
[51] Marcus Niemietz and Jörg Schwenk. Out of the Dark: UI Redressing and Trustworthy Events. *Cryptology and Network Security*, pages 229–249, 2018.
[52] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P. Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, NDSS, 2019.
[53] Alex Perekalin. Why you should be careful with browser extensions. https://www.kaspersky.com/blog/browser-extensions-security/20886/, 2018.
[54] Charles Reis, Steven Gribble, Tadayoshi Kohno, and Nicholas Weaver. Detecting in-flight page changes with web tripwires. In *5th USENIX Symposium on Networked Systems Design & Implementation*, NSDI 2008, pages 31–44, 01 2008.
[55] Guido Schwenk, Alexander Bikadorov, Tammo Krueger, and Konrad Rieck. Autonomous Learning for Detection of JavaScript Attacks: Vision or Reality? In *Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security*, ACM AISEC, 2012.
[56] Ehsan Toreini, Siamak F. Shahandashti, Maryam Mehrnezhad, and Feng Hao. DOMtegrity: ensuring web page integrity against malicious browser extensions. *International Journal of Information Security*, 18(6):801–814, 2019.
[57] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically Securing Web 2.0 Applications Through Replicated Execution. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.
[58] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. A deep learning approach for detecting malicious JavaScript code. In *Security and Communication Networks*, 2016.
[59] Michael Weissbacher, Enrico Mariconti, Guillermo Suarez-Tangil, Gianluca Stringhini, William Robertson, and Engin Kirda. Ex-ray: Detection of history-leaking browser extensions. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017.
[60] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Zigzag: Automatically hardening web applications against client-side validation vulnerabilities. In *24th USENIX Security Symposium*, 2015.

Giorgos Vasiliadis is an Assistant Professor with the Department of Management Science and Technology at the Hellenic Mediterranean University. He is also an Affiliated Researcher with the Institute of Computer Science at the Foundation for Research and Technology - Hellas (FORTH). His research interests include systems, security, and computer networks.

Apostolos Karampelas is a Research Engineer at Tenable's Vulnerability Detection team and an alumni of the Institute of Computer Science at the Foundation for Research and Technology - Hellas (FORTH). His research interests are centered around security and privacy.

Alexandros Shevtsov is presently a Ph.D. candidate in the Computer Science Department at the University of Crete, Greece. Additionally, he holds a research fellowship at the Foundation for Research and Technology - Hellas (FORTH). His research pursuits center around the confluence of security, networking, and machine learning.

Panagiotis Papadopoulos is Head of Red Team at iProov Ltd and a Visiting Researcher at the Institute of Computer Science at the Foundation for Research and Technology - Hellas (FORTH). His research interests include security, privacy-enhancing technologies, biometrics and distributed systems.

Sotiris Ioannidis is Associate Professor with the School of Electrical and Computer Engineering at the Technical University of Crete, and also Affiliated Researcher with the Institute of Computer Science at the Foundation for Research and Technology - Hellas. His research interests include systems, networks, and security.

Alexandros Kapravelos is an Associate Professor in the Department of Computer Science at North Carolina State University. His research interests span the areas of systems and software security.