What Gets Measured Gets Managed: Mitigating Supply Chain Attacks with a Link Integrity Management System

Johnny So
josso@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Michael Ferdman
mferdman@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Nick Nikiforakis
nick@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Abstract

The web continues to grow, but dependency-monitoring tools and standards for resource integrity lag behind. Currently, there exists no robust method to verify the integrity of web resources, much less in a generalizable yet performant manner, and supply chains remain one of the most targeted parts of the attack surface of web applications.

In this paper, we present the design of LiMS, a transparent system to bootstrap link integrity guarantees in web browsing sessions with minimal overhead. At its core, LiMS uses a set of customizable integrity policies to declare the (un)expected properties of resources, verifies these policies, and enforces them for website visitors. We discuss how basic integrity policies can serve as building blocks for a comprehensive set of integrity policies, while providing guarantees that would be sufficient to defend against recent supply chain attacks detailed by security industry reports. Finally, we evaluate our open-sourced prototype by simulating deployments on a representative sample of 450 domains that are diverse in ranking and category. We find that our proposal offers the ability to bootstrap marked security improvements with an overall overhead of hundreds of milliseconds on initial page loads, and negligible overhead on reloads, regardless of network speeds. In addition, from examining archived data for the sample sites, we find that several of the proposed policy building blocks suit their dependency usage patterns, and would incur minimal administrative overhead.

CCS Concepts

• **Security and privacy → Web application security**.

Keywords

Web Resource Integrity; Policies; Browser; Service Worker

ACM Reference Format:
Johnny So, Michael Ferdman, and Nick Nikiforakis. 2025. What Gets Measured Gets Managed: Mitigating Supply Chain Attacks with a Link Integrity Management System. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3719027.3765094

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765094

1 Introduction

The Internet is an interconnected web formed by the linking of resources. By providing core standards such as how to specify a Uniform Resource Locator (URL) for the address of a resource, the web enables a diverse array of applications to interface with one another. These standards build on one another, relying on their foundations to perform their intended functions, as they provide higher-level, and often more easily programmable features. However, there currently exists no robust standard that provides adequate integrity guarantees for a web that relies on addresses whose contents can change.

Thus, the external parties which provide resources such as scripts and stylesheets comprise a significant part of the attack surface of web applications. In supply chain attacks, adversaries compromise existing third parties of a site to modify requested resources to deliver malicious payloads to visitors. Such attacks have recently taken the form of redirectors injected in a popular polyfill library, affecting hundreds of thousands of sites [30]; credit card skimmers injected into a chatbot on an e-commerce site, resulting in a $1.7M USD fine [21, 45]; a skimmer that abused residual trust in an expired domain, affecting dozens of e-commerce sites [19]; cryptojackers injected into an accessibility library, with government sites among the over 4,000 affected [13]; fake browser updaters served from blockchains [61]; and keyloggers in place of trust seals [12]. Ideally, administrators should discover these attacks immediately after they occur, to minimize the impact on their users. However, this may be impractical to do in reality with existing solutions, and so they are often detected after a portion of the site userbase has been exploited.

State-of-the-art integrity mechanisms, limited to only Subresource Integrity (SRI) [66] and Content Security Policy (CSP) [64], can offer some level of protection against supply chain attacks. SRI is a web standard that includes a new integrity attribute to the HTML script tag that leverages cryptographic hashes to ensure that the received script content is exactly the same as the expected content. Similarly, CSP provides features that are relevant to integrity, although it was originally designed to prevent cross-site scripting attacks. At its core, CSP provides a guarantee that resources are loaded from explicitly allowed origins. Newer versions of CSP have introduced support for strict content integrity of scripts by leveraging cryptographic hashes to check for exact matches [65, 69]. Both of these standards use strict, hash-based content integrity checks. However, such integrity verification mechanisms are applicable only for a minority of resources which are not expected to change, such as specific versions of JavaScript libraries loaded from CDNs. In all other cases (for both JavaScript as well as arbitrary resources), exact byte-for-byte checks are impractical in real-world scenarios [52].

Contributions. In this work, we propose a Link Integrity Management System (LiMS) to primarily combat supply chain attacks

by ensuring the integrity of links on web applications, according to configured integrity policies, at the client in near real time. In particular, we summarize our contributions as follows:

- **Integrity Policies**: the concept of granular *integrity policies* as methods to describe the integrity of a web resource by declaring (un)expected properties.
- **Policy Enforcement**: the application-agnostic design of an integrity policy verification and enforcement system that blocks HTTPS requests from being sent by clients to resources which have violated their corresponding integrity policies, minimizing the potential for data exfiltration.
- **Centralized Link Management**: the ability to discover all types of links on deployed sites, including first-party, third-party, and anchor links with high fidelity, and manage them from a centralized solution.
- **Policy Building Blocks**: the proposal of basic policies that can be used as building blocks to form a comprehensive set of integrity policies, while providing guarantees that would be sufficient to defend against recent supply chain attacks detailed by industry reports.
- **Evaluation**: the implementation and evaluation of a prototype, finding minimal performance overhead and no loss of functionality to first-party applications, and demonstrating the applicability of several policy building blocks based on archived snapshots of sites.

The rest of this paper is organized as follows: we discuss relevant background in Section 2, the system design in Section 3, the policy building blocks in Section 5, the evaluation of our prototype in Section 6, the related works in Section 7, the limitations and planned work in Section 8, and our conclusions in Section 9.

2 Background

In this section, we discuss web integrity and the threat model.

2.1 Resource Integrity

Subresource Integrity (SRI) [66] and Content Security Policy (CSP) [64] are existing web standards that provide integrity guarantees based on strict mechanisms that check for exact matches between expected and received content, by leveraging cryptographic hash functions. Through these mechanisms, website developers may specify the expected hash digest(s) of JavaScript files on webpages through the `integrity` attribute in HTML `<script>` tags for SRI, or through the `script-src` directive for CSP. If these are present, the user agent (e.g., browser) is expected to compare the hash of the actually received content against the predefined hashes, and block the loading of resources whose computed hashes do not match their expected ones.

Exact matching works for static resources that are not expected to change (e.g., a specific version of a library from a CDN), but does not apply to a significant portion of resources that are dynamic. For such resources, their URL address, content (e.g., through updates or dynamic modifications to whitespace, syntax, block ordering, comments or data), or dependencies (e.g., fourth-party scripts) [52] may frequently change. Furthermore, prior work has uncovered that *frequently-changing scripts are no longer the exception, but the norm in the modern web*: when crawling tens of thousands of domains daily, the study found that only 11% of script URLs are static, and only 3% have static content [52]. The lack of integrity guarantees for these resources is concerning — there are no security measures that can protect users if the contents of such resources are unexpected.

2.2 Threat Model

Consider the following scenario: a user visits example.com which pulls in external subresources on its pages, such as fonts, images, cascading style sheets and JavaScript files. The administrators of example.com use an appropriate CSP that defines lists of trusted origins for their external resources, and mark static resources with their expected SRI hash digests. However, they also use resources that are not easily integrated with strict content integrity checks offered by SRI or CSP, such as foo.com/foo.js. Although the example.com administrators include the origin foo.com in their CSP script-src directive, there is no expected hash for that script.

Scenario (A): Expiration. If foo.com expires and is re-registered, visitors of example.com could suffer from a supply chain attack when fetching, and executing, malicious JavaScript from foo.com/foo.js in their browsers. Ideally, example.com administrators would discover this, remove the script from their page, and update their CSP, before the foo.com domain is expired or re-registered, but this process could take an extended amount of time [53]. Similarly, the domains of anchor links (HTML `<a>` tags) found on example.com are also subject to the same issue. Although the damage may not be as severe as a supply chain attack (e.g., with the foo.js script), if an anchor links points to a malicious domain, it will still negatively affect the reputation of example.com. SRI and CSP cannot be applied to anchor links.

Scenario (B): Link Content Change. Alternatively, it may be the case that the domain foo.com did not expire, but the content of foo.com/foo.js unexpectedly changes. SRI and CSP are not applicable because the script is dynamic. Regardless of whether the change occurs because of a malicious compromise, or an unscrupulous update to its data collection practices, the administrators of example.com may desire to have an automatic mechanism to inform them if foo.com/foo.js changes in an unexpected manner, and block that resource for their visitors. As in Scenario (A), site admins may also be interested in applying this mechanism to anchor links as well.

Threat Model. A user interacting with an online, benign web application that properly utilizes SRI and CSP loads external resources that may have been undesirably modified. The goal of LiMS is to prevent supply chain attacks on its deployed website, assuming a sufficient and robust set of integrity policies, by detecting "significant" changes to dependencies in near real time, blocking all website visitors' requests to these resources, and flagging the incident to administrators. The threshold for the significance of change is determined by the set of integrity policies that are enabled by administrators, and should ideally encapsulate both Scenarios (A) and (B). If a malicious third-party script is able to execute in the browser of a site visitor, that means the configured integrity policies did not flag the (change in that) resource. Section 3.4 elaborates on the trust model under which LiMS operates, and Section 4 further discusses security-critical details that are imposed by design and implementation choices.

```
policy = { rule };
rule = action, url_pattern_page
     , url_pattern_resource, [ "if", condition_name ], ";" ;
action = "allow" | "deny";
url_pattern_page = url_pattern
url_pattern_resource = url_pattern
url_pattern = '"', { url_char | "*" }, '"' ;
url_char = letter | digit | "." | "/" | ":" | "_" | "-" ;
condition_name = policy_building_block | custom_condition ;
custom_condition = letter, { letter | digit | "_" } ;
```

Listing 1: EBNF grammar for the LiMS integrity policy language.

2.3 Service Workers

In our prototype, we implemented the client of LiMS as a service worker (SW) [44], a performant worker that acts as a proxy between the browser and the network. Although service workers were not explicitly designed for our use case (one of the original design goals was to enable the creation of offline web applications), we found that their capabilities expressly suited our needs: the ability to transparently intercept *all* HTTPS requests (including navigations) that originate from the pages under its purview, and take different actions. Furthermore, this implementation choice leverages a web standard supported by all major browsers [28], and removes the need to directly modify browser code – which are highly-optimized and massive repositories of software – thereby improving the accessibility of LiMS for site administrators and researchers. We further discuss the role of the client SW in Section 3.2 and its disadvantages in Section 8.

3 Link Management System (LiMS)

LiMS guarantees that an HTTPS request originating from a user of a website is sent *if and only if* its pre-configured *integrity policies* hold true at the time of, or near the time of, the request. If at least one policy does not hold true, then LiMS will temporarily block *all visitors' requests* for that resource, and flag the incident for administrators. This functionality is ensured by three main components working in tandem: the client-side component that intercepts requests ("client"), the server-side component that manages link state ("server"), and the server-side component that verifies configured policies on demand ("verifier"). Figure 1 presents a high-level diagram of LiMS that can be followed with the next sections that discuss:

- (§3.1) the fundamental integrity policies that describe the (un)expected properties of resources,
- (§3.2) the client service worker that enforces policy decisions and blocks requests to resources that violate their corresponding policies,
- (§3.3) the server that responds to the client with whether requests should be blocked, as well as the verifier that verifies the configured integrity policies on demand.

In addition, the trust model of LiMS is discussed in Section 3.4, and deployment strategies in Section 3.5. We open-sourced our prototype implementation of LiMS, which can be found at https://github.com/link-integrity-management-system/lims.

Algorithm 1 Request interception logic from the perspective of the LiMS client-side service worker.

```
1:  procedure InterceptRequest(req)
2:      allowed ← False
3:      if HasValidCacheEntry(req) then
4:          allowed ← GetStatusFromCache(req)
5:      else
6:          allowed ← QueryAndCacheLinkStatus(req)
7:      end if
8:      if allowed then
9:          Fetch(req)                          ▷ Defer caching to the browser
10:     else
11:         Generate404Response(req)
12:     end if
13: end procedure
```

3.1 Integrity Policy Language Specification

Integrity policies form the core of LiMS by expressing certain conditions — in code — that are always expected to be true for subsets of resources. Thus, an individual integrity policy is not intended to encompass all conditions that an administrator wishes to ensure, but rather a single check for a class of resources. To this end, LiMS is designed to scale with the use of many concurrently active policies. Furthermore, as the integrity policies themselves are configurable, the conditions they support are highly varied, running the gamut from exactly matching the content of resources, to verifying the initiator of the request itself, to comparing the network infrastructure of a third party.

Listing 1 uses Extended Backus-Naur Form (EBNF) to describe the LiMS policy language. Integrity policies are defined as rule sets that invoke conditions for matching URL patterns. There are two URL patterns for each policy: the first matching a first-party page URL, and the second matching an expected resource request URL on that page. The client either allows or blocks each request by matching it against a policy's URL patterns and evaluating the condition (if one is specified). The condition may refer to the name of a policy building block, as described in Section 5, or a custom policy condition written by an administrator. The policy conditions execute on the server and can verify the (un)expected properties of the requested resource, including re-enacting the request context with high fidelity, if desired. The conditions are evaluated and used asynchronously (§3.3).

Administrators can implement policy conditions in a familiar programming language (e.g., NodeJS). A simple policy can provide the same guarantees as existing defense mechanisms such as SRI (e.g., check that the hash of the response content matches), but the design of LiMS offers greater flexibility. For example, a condition can replay a script request by launching a headless browser and visiting the first-party page on which the script was found, allowing LiMS to check parameters such as the script request initiator, the geographic location(s) of the remote provider servers, structural signatures, fourth-party inclusions, and randomization or time-varying behavior [52].

However, we recognize that complex security policy systems are not always utilized to their full potential (e.g., CSP is often misunderstood and hard to use correctly [43]), and so accordingly we present universally-applicable policies that can be used as the foundation of comprehensive integrity policies in Section 5.

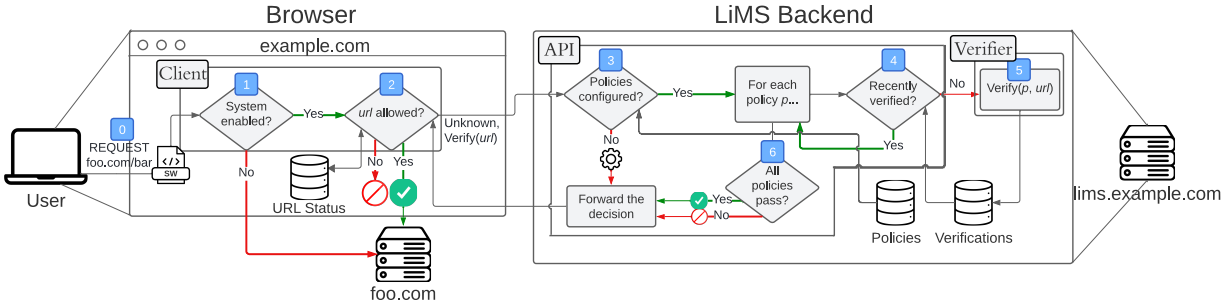

Figure 1: Diagram depicting high-level interactions between the client service worker and the LiMS API that comprise the verification protocol to determine whether an outgoing request should be allowed, or blocked, by the service worker.

3.2 Policy Enforcement

The LiMS client directly impacts users, as it enforces policies by blocking HTTPS requests to resources that violate their corresponding policies, as shown in the left of Figure 1. In our prototype, the client is implemented as a service worker (SW), which brings two main benefits to LiMS: transparent operation and performance.

Transparent Operation. When a site deploys LiMS, they will add a JavaScript function call to `navigator.serviceWorker.register` that instructs browsers to install the specified SW. This SW is capable of intercepting *all fetch requests that originate from a page under its control* (including navigation requests); thus, by registering the SW with a control scope set to the root of the domain, it will be able to intercept fetch requests from any page of that domain. If supported by the browser, the SW will be installed, assume control over all applicable pages, and then refresh all tabs under that domain to ensure that resources on the page only load if their policy verifications succeed. If not supported by the browser, there will be no change to existing, first-party functionality.

While the client SW is active, it will regularly poll the API server for its configuration settings, doubling in functionality as a heartbeat message. This periodic polling serves as an alternative for push notifications that the server may send to revoke verification decisions from the client cache (useful if the client disables or is unable to receive push notifications). If the number of times that the client SW fails to connect to the API server exceeds a configurable threshold, it will automatically revert to a no-op mode by disabling its request interception logic, until it successfully receives a response.

Performance. In our prototype, we opt for a straightforward implementation that offloads the processing of policies, and detections of violations, to a *server-side component that can reuse the verification decisions for different users*, thereby rendering the client as a simple proxy. When the client intercepts a request, it only queries the server for whether the request should be allowed to proceed, and locally caches the decision it receives to be reused for a certain duration of time. An overview of the main client logic can be found in Algorithm 1.

Server-side verification in this manner enables the following: the first user who loads a resource that needs to be (re-)verified will have the policy verifier make a decision that can be *reused on the next request for that resource, even if it is from a different user*. Furthermore, this will keep the overhead of the LiMS design within acceptable bounds, even though LiMS intercepts requests — and sends additional ones — made by web applications that may have explicit requirements for fast user interfaces. In short, a server-side component verifies policies and communicates these decisions to client-side, in-browser, integrity proxies that uphold the guarantees for the user. This choice introduces challenges to accurate verification of integrity policies (which are elaborated in Section 8), but we argue that this is a necessity given modern website-speed requirements [7] and discuss its implications in Section 4.

Algorithm 2 Policy enforcement logic from the perspective of the LiMS server-side API.

```
1:  Class BasePolicy
2:    Variable locationSource : String
3:    Variable locationTarget : String
4:    Variable logic : Function
5:    Variable output : Bool
6:
7:  procedure GetLinkStatus(req)
8:      link ← GetLink(req)
9:      verifications ← GetVerifications(link.id)
10:     for v of verifications do
11:         if v.failed then
12:             return false
13:         end if
14:     end for
15:     return true
16: end procedure
```

3.3 Link Management and Policy Verification

As described in Section 3.2, the client SW component that enforces policies is intentionally simple — it defers all decisions it does not know about to the backend. The backend components, as can be seen on the right side of Figure 1, comprise the API server and the verifier.

LiMS Server. The server-side component that responds to queries from the SW ("server") exposes a simple API to determine whether client resource requests should be allowed. When receiving a request from the client, it records metadata about the request, including the page where the request was encountered and the request URL, determines which policies are applicable to the request, and consults its cache for recent verifications for each policy. If there exists a valid, recent verification for every applicable policy, the server responds to

Algorithm 3 Policy verification logic from the perspective of the LiMS server-side verifier.

```
1: procedure VerifyLink(link)
2:     policies ← GetPolicies(link)
3:     for policy in policies do
4:         success ← ExecutePolicy(policy, link)
5:         CacheVerification(link, policy, success)
6:     end for
7: end procedure
```

the client with the decision to allow the request; if at least one policy has recently failed its verification for the request, the server responds with the decision to block the request. If any policy was not recently verified, the server can respond to the client with a default decision, or optionally schedule the verifier to process it on-demand and use the result, if it does not exceed a configured timeout. Algorithm 2 describes an overview of the main server logic.

LiMS Verifier. The last major LiMS component is the verifier, which processes integrity policies to determine if the conditions they express hold true. These results are termed verification decisions, and they are stored in a cache so that the server can use them to formulate its response to queries from the client. As shown in Figure 1, the verifier is not directly part of the client-server request flow. Instead, it interacts with the server asynchronously to verify policies on demand. If configured, the verifier can also periodically verify all policies before their cached decisions expire, in order to maintain a warm system state and meet performance requirements.

When the verifier receives a verification request, it filters all policies for those that match the page where the request was encountered and the policy target URL pattern, to obtain the set of policies that are applicable to the request. Next, it executes the logic in each policy and caches the outcome of each verification which remains valid for each policy's time-to-live duration. These cached verification decisions will then be used by the server, when it receives a request from the client. Algorithm 3 summarizes the basic verifier functionality.

As the verifier is a server-side component, it is imperative to ensure that the process of policy verification does not sit in the hot path of the client's request to the server. The asynchronous interaction model between the server and the verifier, where they leverage the database as an intermediary, is designed to minimize the overhead imposed by LiMS. In an ideal, warm state, verifiers can periodically verify all links against all policies, before any cached decisions expire, to ensure the cache is always populated for requests received by the server. If there is always a cached decision, the server only requires a database lookup before responding to the client with a decision on whether the request should be allowed.

3.4 Trust Model

LiMS is designed to combat the threat model outlined in Section 2.2, in which a trusted first-party application depends on resources from a third party whose contents may change without notice, by placing flexible integrity policies on resources. To do this, each of the major LiMS components outlined in Figure 1 must be trusted: the integrity policies, the client service worker, the server that responds to client queries, the policy verifier, and the database or cache that stores policy information and verification decisions. If any of these components are compromised or dishonest, then an adversary can effectively disable LiMS. A dishonest server can respond to client queries to trust undesirable resources; a dishonest verifier can decide to trust resources, regardless of the configured policies; and a compromised database can lead the server to believe that policy verifications succeeded. Moreover, LiMS necessarily trusts the first-party application (and the client service worker), and expects an honest browser environment. Otherwise, a compromised first-party may not deliver the expected client service worker code, or a malicious browser extension may silently interfere with regular service worker operations (e.g., by blocking the request that fetches service worker code and preventing installation of the client, or blocking client requests to the LiMS server).

3.5 Deployment

Interested researchers and administrators can deploy LiMS for existing applications at minimal cost. Our open-sourced artifact provides ready-made prototypes for each system component as Docker containers. To self-host a deployment, an administrator needs to obtain a server to host the containers, to expose a public file that contains the service worker code at the root of the sites they wish to protect (e.g., at example.com/sw.js), and to include a snippet of JavaScript code that registers the service worker in HTML documents. For sites with existing service workers, the logic of the SW can be added to the start, or to the end, of the request handling logic to avoid breaking existing SW functionality. As we later see in Section 6.2, the immediate benefits for sites that already utilize service workers is actually greater than those for sites that do not, because of the base overhead imposed by the existing service worker request interception.

Service workers are supported by all major browsers [28], and the enforcement logic is designed to provide smooth opt-in and opt-out experiences. If a user's browser does not support service workers (e.g., because of an old version), or if the user has configured their browser to block service worker installations (e.g., because of privacy concerns), their browsing sessions will be exactly the same for sites that deploy LiMS and those that do not. If a user whose browser supports service workers visits an LiMS-enabled site, the SW will force a refresh on all tabs for that site upon installation to ensure that the integrity policies are applied. If a site owner wishes to remove LiMS, they only need to include a snippet of JavaScript that uninstalls any existing service worker registrations in client browsers, and take down the backend components. Additionally, if any client SW is unable to reach the backend components, it will automatically revert to a no-op mode that does not enforce any integrity policy decision, nor make additional network requests.

3.5.1 Multi-Stage Deployment. Deployment of LiMS can be performed in multiple stages to facilitate a smooth onboarding. In the first *link discovery* stage, the API server can function in a no-op mode, responding with a default message that allows all requested resources, effectively populating the LiMS database with the links that are requested by users in real time. In the next *report-only* stage that functions similarly to the mode in CSP, administrators can review the links (as they populate) to start writing their desired policies, and configuring the server to always respond that requests are allowed, even if any policy is violated. Violations will be reported by the API server and stored in the database by the policy verifier, enabling

administrators to review existing policies to check for errors. When the policies provide sufficient coverage, LiMS can be switched into a normal operation mode: if any corresponding policies are violated, the API server will instruct the client SW to block the request.

4 Security Considerations

As LiMS is designed to provide additional integrity guarantees, LiMS itself must be sufficiently robust in our threat model. This section discusses an array of security-critical topics, including the suitability of service workers in our threat model, policy robustness, high-fidelity link discovery and management, caching exploits, camouflaging, and policy consistency and updates.

4.1 Service Worker

We leverage service workers to provide integrity guarantees for users, such as those presented in Section 5, by assuming that the content delivered over the network may be malicious or undesirable because of domain expirations and re-registrations (Scenario Ⓐ from Section 2) or changes to third-party content (Scenario Ⓑ).

LiMS uses service workers as the component to enforce policy verification decisions within client browsers. In our threat model, an undesirably-modified resource in the supply chain of a first-party site should ideally be flagged by one or more policies, causing user requests for that resource to be blocked. If the resource is not flagged by any policies, then that indicates there is a gap in the set of integrity policies deployed by the site administrators, and the script will be allowed to load. Any HTTPS requests made by this modified script must pass all appropriate policies, or they will be blocked and flagged.

LiMS requires every page of the first-party site to include the JavaScript code for service worker registration, ensuring that the client browser installs the service worker regardless of which page a user visits. The service worker only needs to be registered, installed, and activated once, until a new version of the service worker code is fetched. After installation, the service worker will be present on all subsequent visits. Cautious administrators would configure their sites to install the service worker before other scripts load (i.e., before redirecting to the main content), as the service worker container is exposed to JavaScript via the navigator.serviceWorker property. Before the LiMS service worker is activated, malicious third-party JavaScript could interfere with the service worker registration. We expect a benign first-use environment and consider this case outside of our threat model, but it is possible for the client to be subject to attacks from a malicious script that was not blocked by the configured integrity policies, which we outline next.

4.2 JavaScript-based Attacks

Any JavaScript file that is included and loaded in the webpage will have access to the navigator.serviceWorker object. Access to this object grants the ability to install a service worker hosted at an HTTPS URL within the origin; thus, a malicious third-party script cannot install an arbitrary service worker without control of the origin's web server. A malicious script can uninstall a service worker, but *an uninstalled service worker retains control of the page until the subsequent navigation.* Thus, malicious third-party JavaScript that was not blocked can attempt to disable LiMS by unregistering service workers and triggering a refresh or navigation, but the LiMS service worker, if properly deployed, will be the first thing re-installed on the subsequent page load. Theoretically, attackers can repeat this behavior while a gap in the integrity policies persists, but the attack would be limited to denial-of-service (rather than, for example, skimming credit cards), and the behavior would drastically accelerate discovery of the malicious script.

A malicious script that bypasses the configured integrity policies may also attempt a denial-of-service attack on the client service worker by flooding it with spurious messages through the navigator.serviceWorker.controller.postMessage API, or with fetch requests. The service worker can easily detect the former. During normal usage, the service worker would encounter a message only once from the registration script to forcibly refresh the page after installation, and subsequent messages can be ignored. However, the latter introduces additional network requests and, in turn, additional policy verifications for the resources.

In general, if a malicious third-party script is allowed to load on a client, that means that the configured integrity policies for that site have been bypassed. LiMS is not intended to defend against other attacks such as prototype pollution or DOM hijacking.

4.3 Policy and Cache Robustness

Sites that deploy LiMS can configure an unlimited number of policies. As these policies are not communicated outside of the LiMS backend (see Figure 1), a supply-chain attacker cannot directly learn of the policies that are configured for a site. They can indirectly infer policies by iteratively modifying properties of a resource on the supply chain, and observing whether the LiMS client service worker blocks the request for that resource. However, *indirectly inferring policies can be expensive*, as many variables are unknown: the number of associated policies for each resource, the exact checks performed by each policy, and the cache duration of each policy.

As such, an adversary that wishes to stealthily modify a recently verified resource, while the decision remains cached, must resort to iteratively probing for information by improperly changing the resource and causing LiMS to block it, for extended periods of time. We expect administrators to investigate the resources that LiMS blocks, thereby uncovering the offending link(s) before an adversary is able to extract sufficient knowledge of the policies. In general, we recommend that administrators configure multiple types of policies to check different integrity dimensions [52] for dynamic resources in sensitive locations (e.g., the request initiator for, and fourth-party inclusions of, an external script on an ecommerce checkout page).

4.4 Link Discovery & Management

LiMS provides another security-critical feature, in addition to the integrity guarantees: the ability to map out all types of links on their sites with high fidelity, and the ability to manage all of them from one centralized solution. When the client SW encounters a link with an unknown status, it will query the server to check if the request should be allowed to proceed. In the process, the LiMS server will automatically build a database of all links, on all pages, of the website.

Typically, link discovery mechanisms can be classified as some combination of the following strategies: crawling, intercepting network traffic (e.g., web application firewall), tracking application or web server log messages, analyzing source code or runtime behavior,

or monitoring real users. There exist many link discovery products in industry, but they are primarily crawlers for the purpose of search engine optimization, web application firewalls, or user behavior monitors for analytics services. To the best of our knowledge, LiMS is the first proposal to bootstrap integrity guarantees to web browsing through integrity policies, to offer high-fidelity link discovery based on real-user monitoring, and to provide centralized management.

4.5 Resource Camouflaging

The LiMS client does not directly detect camouflaging, as service workers are prohibited from inspecting cross-origin response contents. If desired, developers can deploy their own cloaking-detection pipeline that simulates different user classes (e.g., desktop or mobile) and networks (e.g., residential or cloud) to detect links whose contents vary by predetermined parameters. This approach was shown effective in prior work [16]. Additionally, developers can also use malware detection services to detect cloaking. Such data can feed into a LiMS policy that denies requests to resources that appear to use camouflaging techniques, as in Appendix A, Algorithm 6.

4.6 Policy Consistency

In the event that a LiMS administrator erroneously configures contradictory policies, this would cause continuously-failing verifications, and the associated resources will always be blocked. The administrator would readily notice and investigate the growing number of verification failures. It is also possible to incorporate logic to identify conflicting policies and warn the administrator when such policies are enabled. For example, another monitoring component can be introduced to the backend of LiMS, which can periodically scan for resources that are governed by multiple policies which always result in contradicting verification decisions.

4.7 Policy Updates

If an administrator wishes to update an existing policy, LiMS can invalidate active verification decisions for that policy in the server-side cache. When the server encounters a client request for a resource governed by the updated policy, it will trigger a re-verification according to the new policy. Verification decisions cached by the client for these resources will be invalidated either via push notifications or heartbeat responses.

Policies may need to be updated in some scenarios. For example, if a policy blocks a changed benign resource, the duration of time from when the resource changed until the policy is updated could result in broken page functionality, but administrators can readily investigate blocked resources. On the other hand, if a policy continues to allow recently-changed resources, a malicious script may be allowed to load, possibly leading to some attacks described in Section 4.2.

5 Policy Building Blocks

In this section, we focus on the capabilities of integrity policies, and present several universally-applicable policies in the context of notable historical security incidents arising from malicious changes in resources used by an online, first-party application. In particular, we focus on how LiMS could have efficiently defended against each breach using simple policies that are not able to be implemented in existing integrity mechanisms (e.g., CSP).

Algorithm 4 A sample integrity policy that denies requests to resources whose domains were recently registered.

```
1:  Class PolicyDomainLifecycle
2:    Variable locationSource = "example.com/.*"
3:    Variable locationTarget = ".*"
4:    Variable logic = IsRecentlyRegistered
5:    Variable output = False
6:
7:  procedure IsRecentlyRegistered(req)
8:      threshold ← GetRegistrationThreshold()
9:      allowlisted ← IsAllowlistedForRegistration(req.domain)
10:     recentReg ← GetRecentRegistration(req.domain)
11:     return NOT allowlisted AND recentReg > threshold
12: end procedure
```

Table 1 summarizes the policies proposed in this section, and lists related security incidents. These policies are intended to illustrate their usefulness in the context of actual security incidents: they are not meant to provide an upper or lower bound on the capabilities of integrity policies. We also note that although the functionality offered by LiMS is a superset of the functionality offered by CSP and SRI, LiMS is not designed to be a drop-in replacement for them.

This section discusses these policies at a high level as the goal is to convey the *potential* of policies, and of LiMS, to provide integrity guarantees in a flexible manner, and not to present exact algorithms. As such, we refer interested readers to Appendix A for pseudocode representations and to the open-sourced artifact for prototypes of the policies in this section.

5.1 Policy: Domain Lifecycle

It is imperative for administrators to be able to detect when their included resources belong to domains that are about to expire, or have already expired, as their sites may be vulnerable to supply chain attacks by malicious re-registrants of the expired domains [53]. However, there are no default or standardized mechanisms that perform this function. It is possible for dependencies of domains to expire and go unnoticed for months, and for attackers to select particular expired domains to target infrastructure, or opportunistically re-register them and exploit whatever is available [25, 53]. For instance, a 2022 report detailed a campaign that re-registered the expired domain of an analytics service that was discontinued in 2014 to serve credit card skimmers, and it was still able to impact over 40 different e-commerce sites [19]. There have been many other campaigns where attackers infect websites, inject malicious code on checkout pages, and exfiltrate credit card numbers as users type them, to newly-registered domain names [6, 19, 47, 48].

Algorithm 4 presents a basic policy that blocks requests for all resources whose domains were registered after a configured threshold. Recent registration data can be obtained from WHOIS data, or inferred from passive DNS data. Although the policy itself prevents requests to domains that are newly registered and does not attempt to distinguish domains that had previously expired, it can be modified to distinguish between previously-observed domains that have expired, and then were re-registered. This policy does not protect against attackers who have compromised an existing party in the supply chain of the first-party site, but it does effectively negate the

Table 1: Basic integrity policies to use as building blocks to build a comprehensive set of integrity policies.

Policy	Description	Recent Incidents
Domain Lifecycle	Domain was recently registered	[6, 19, 47, 48, 53]
Domain Ranking	Ranking of domain is below a threshold	
Threat Intelligence	Domain or IP address found in threat intelligence feeds	[47, 61]
Dependencies	Change in set of third-party origins that are contacted by a script	[6, 12, 13, 22, 29, 32, 46, 47, 60–63, 67]
SRI Violation Reporting	Client-side errors such as failed SRI verifications	-
Infrastructure Attributes	Geographic restrictions on servers that provide dependencies	[61, 63]
CMS Core File Integrity	Modifications of core CMS files (e.g., WordPress or Magento)	[22, 27, 56]

threat of re-registrations of existing domains in the supply chain that were left to expire (Scenario Ⓐ from Section 2) and the common DNS evasion pattern that uses throwaway, short-lived domains in web malware infrastructure [2, 3]. Note that this policy also applies to domains that exist in CSP allowlists.

5.2 Policy: Domain Ranking

Domain reputation and popularity can serve as an indicator of quality and security: higher-ranking sites are generally expected to have better security postures because of their amount of users. Although it is not an absolute relationship, studies have found some evidence that generally support this correlation [43, 50]. Thus, administrators may desire restricting resources that are fetched from, and linked to via anchor links, lower-ranked domains for security, or to improve the ranking of their own domains. Appendix A Algorithm 5 encodes this logic as an integrity policy, denying requests for resources to domains with a ranking that is below a configured threshold, and not explicitly allowed. This policy does not protect against attackers who can introduce a high-ranking domain, but adversaries commonly resort to using throwaway domains, which will inevitably be low ranked or unranked in robust ranking lists (e.g., Tranco [41]).

5.3 Policy: Threat Intelligence

Threat intelligence feeds and domain blocklists provide data sources comprising indicators of compromise. These sources aggregate suspicious indicators from prior security incidents, such as data exfiltration endpoints (domains or IP addresses), or hashes of compromised files, and are often integrated into security infrastructure. Similarly, malware detection services (e.g., VirusTotal) check for not only signatures of previously-identified malicious content, but also suspicious behavior through static and dynamic analyses. Thus, administrators may desire to use such services to scan the external resources they are linking to from their first-party site, while maintaining their own cache of previously-scanned content to minimize API calls. Appendix A Algorithm 6 incorporates this idea into one LiMS policy, enabling the verifier to automatically block requests for domains that appear on threat intelligence services, or whose contents were flagged as suspicious by static or dynamic analyses.

One method to minimize costs of external scanning is to check files whose contents have changed, particularly for file types for which SRI is not applicable, such as images and audio files. The metadata of image files may be used for covert payload delivery or data exfiltration. Malwarebytes has published a number of reports detailing adversaries masquerading skimmers as favicons: in one 2020 incident, the favicon request URL would serve a web skimmer if the word "checkout" was in the Referer header [47], and in another, the skimmer payload was hidden in the EXIF metadata field of a favicon [48]. In contrast, maliciously-crafted audio files may trigger arbitrary code execution when browsers process them [9, 70].

This type of policy can also be designed with heuristics to identify common behavioral patterns in supply chain attacks, by leveraging first-party knowledge and resources. For example, it is common for obfuscated, malicious snippets of JavaScript to be injected into otherwise-benign code, which is sometimes called a *benign-append* attack in industry reports. In 2018, a common JavaScript library loaded by 4,000 sites, many of which belonged to governments, was injected with a cryptojacking snippet [13]; in 2019, an advertising agency was compromised and delivered credit card skimmers to 277 e-commerce sites [32]; in 2023, 510 WordPress sites were found embedded with bridgehead code that retrieved second-stage fake browser update payloads from a malicious smart contract, effectively leveraging the principles of blockchains to serve as bulletproof hosting [61]. In all of these incidents, obfuscated, malicious code was injected into existing resources. Policies that leverage first-party knowledge will know whether certain scripts were previously obfuscated, or expected data values in analytics scripts (e.g., the ID used by the Google Tag Manager loader script that determines what further scripts are pulled into the site). Another targeted behavioral pattern could be the tendency for attackers to reuse or revive older infrastructure that have been previously marked suspicious; an incident in 2020 involved a credit card skimmer that masqueraded as a favicon, and the IP address of the domain that served the skimmer had been flagged malicious three years prior [47].

5.4 Policy: Dependencies

Scripts may regularly contact different origins (or URLs) during their execution. LiMS supports the ability to check for changes to the set of *URLs* contacted by individual *resources*, as opposed to the granularity of *origins* contacted by individual *pages* as offered by CSP. Consider a scenario in which script foo.js has changed and contacts an origin bar.com that it never contacted before. If bar.com is already contacted by a different script on the page, and thus present on the CSP allowlist, this issue may never be flagged to administrators without LiMS. Or, it might be the case that foo.js contacts a different URL on a domain bar.com that it has previously contacted. This would be particularly concerning if bar.com allows for user-provided scripts to be uploaded. For example, Google Tag Manager (GTM), a popular analytics library, enables developers to host their own set of custom scripts, differentiated by a single ID parameter in the URL and in a variable in the initial loader script content. Adversaries are known

to abuse brand trust and change the parameter in the GTM script URL that controls which scripts are loaded [1, 51, 57].

Appendix A, Algorithm 7 specifies a straightforward version of this policy. In production, this policy may not be readily applicable for dynamically-generated URLs. For example, if a third-party script contacts dynamically-generated subdomains of a fourth party, the policy may have to be adjusted to consider only the eTLD+1 instead of the full domain name, if all subdomains are controlled by the same entity. However, if different subdomains redirect to different user-controlled content, it would be prudent to consider the full domain name.

5.5 Policy: SRI Violation Reporting

One challenge faced by web developers may be the lack of visibility into silent errors encountered by visitors. For example, although CSP offers a built-in reporting mechanism for resources that were blocked because of violations, SRI does not. When a script is blocked from loading because of an SRI violation, the browser only reports the failure in its developer console. The closest feature to a reporting mechanism for SRI is a now-deprecated proposal to include `require-sri-for` in CSP, which would block scripts without an SRI `integrity` attribute and leverage CSP `report-to` [68].

Administrators can use a LiMS policy to ensure that scripts that do not match their expected SRI digests will provide violation reports to administrators, in addition to providing the same functionality as SRI if desired. Appendix A Algorithm 8 describes a policy that provides visibility into these errors that are, by default, only visible in the developer console for browsers. Although it does not directly provide additional integrity guarantees for users, it improves the ability for administrators to detect problems with critical resources, and thus minimize the response time during attacks. Additional client-side errors that might be of interest to developers are TLS connection errors for subresources on the page — when the browser encounters such errors, they are also only reported in the developer console.

5.6 Policy: Infrastructure Attributes

Depending on the application infrastructure and the domain context, administrators may need to make policies that encode restrictions or business logic, such as where the physical remote servers of dependencies are expected. For example, a common deployment pattern geographically distributes hosting servers, and directs users to servers that are physically close to their location to minimize latency. In addition, security-conscious developers may be concerned about dependencies from foreign countries for critical applications (e.g., government websites), or connections to servers in unexpected locations. A security analysis published in 2023 discusses malicious JavaScript code that queried an Ethereum provider to retrieve a second-stage domain from a malicious smart contract, which resolved to IP addresses located in a foreign country [61]. In another security incident report in 2023, skimmers targeted sites across the globe and exfiltrated data to a server located in Japan [63].

Appendix A Algorithm 9 encodes a broad check for resources hosted by servers in unexpected locations, by performing a DNS query for the domain, and geolocating the resulting IP address of the remote server. Using this result, LiMS can compare the country of the result against a pre-defined allowlist or blocklist, and compare the distance to the remote server against a distance threshold as a means to enforce a maximum distance restriction. Variations of this policy can additionally check for suspicious indicators based on the DNS responses to queries for the requested domain, such as the number of distinct IP addresses or countries [3] or autonomous systems for those IP addresses [2].

5.7 Policy: CMS Core File Integrity

Content management systems (CMS) comprise a significant fraction of the websites deployed today — some sources report that over 75% of the top 1M sites by traffic are built with CMS software [4]. Given their prevalence, attackers frequently target sites that use CMS software, and one of the common targets are the "core" files that are sent to users (e.g., WordPress jQuery [22] and themes [56]), or encode server-side logic (e.g., session management in Magento [27]).

Appendix A Algorithm 10 encodes core file checks as a special type of LiMS policy that can be run to check for both client-side (e.g., JavaScript) or server-side (e.g., PHP) core file integrity. Client-side checks would be performed as usual, but for server-side checks, the verifier would require access to the corresponding source files used by the web application. For these cases, the policy would need to reference another source to obtain the expected content for core files, and compare it to the content that is actually received by the client, or the content that is actually present in the web application, respectively. This is in contrast to SRI and CSP, which cannot help with server-side checks, but can help perform client-side checks, provided the CMS software supports their use. Further, content checks with SRI or CSP do not apply to all file types that may be considered core CMS files.

6 Evaluation

To evaluate the performance overhead of our prototype, we implemented a pipeline that simulated deployments by injecting LiMS service worker registration scripts into responses from the server. In this section, we evaluate the performance overhead of our LiMS prototype, and evaluate the robustness of several proposed policy building blocks from Section 5. As the simulated deployment methodology is not a critical component of LiMS, we refer readers interested in the technical details to Appendix C, which discusses them at depth.

6.1 Domain Sample

We selected a representative sample of domains to be used in our performance and policy evaluations. First, we obtained a ranked list of the top one million domains from Tranco on September 18, 2024. We divided the domains into high-, mid-, and low-ranking buckets of increasing size, [1, 1K], (1K, 100K], and (100K, 1M) respectively. We then randomly sampled 200 domains from each bucket. The resulting sample contains domains that span a diverse array of services, as outlined in Appendix B, Table 3. We filtered the selected domains, excluding 5 domains with adult content keywords, 132 domains for which our crawler did not receive a response in 30 seconds, and 13 domains with existing service workers. The remaining 450 domains were used in our performance evaluation, following the simulated deployment methodology described in Appendix C.

Request Patterns. Before discussing the evaluations in detail, we characterize the inclusion patterns of the domains in our sample from the perspective of a LiMS administrator by using the generated data from our simulated deployments. For each domain, the median

Table 2: Different stages of the performance overhead evaluation imposed by our prototype LiMS.

Stage	Overhead Measurement
`No SW`	Baseline measurement with no SW
`No-op SW`	No-op request interception with SW
`No-op API`	Queries link status from no-op API
`Full`	Queries link status from normal API

number of external URLs that were requested is 197, number of external origins is 7, and number of URLs per external origin is 32. As expected, third-party advertising and tracking sites are requested very often, from many of the domains in our sample. We observe that googletagmanager.com is the most commonly-requested external origin, with 170 domains in our sample triggering an average of 2 unique URLs to this external origin on each visit, with a total of 1,247 different URLs encountered throughout our performance evaluation. This site is responsible for delivering advertising-related scripts, based on the URL parameter `id` that identifies the set of scripts to load. In contrast, we also observed external origins that were used by only one domain in our sample, but were requested with many unique URLs per visit. We found that, on each visit, whatsapp.com requested, on average, 63 unique transient URLs from whatsapp.net, most of which were images that return an error message outside a regular browsing session (e.g., "Bad URL timestamp/hash"). Similarly, sunsky-online.com triggered requests for 20 URLs to myipadbox.com on average, but these were static URLs that corresponded to images of their e-commerce inventory.

Takeaways. Domains in our sample are representative in terms of ranking and categorization: our sampling methodology is designed to represent low-, middle-, and high-ranked domains, and also to select domains that serve a diverse array of functions. There is no universally-applicable third-party resource inclusion pattern, and we observe evidence of different request inclusion patterns in our simulated deployments. As such, administrators must first understand their existing third-party resources before writing policies. The multi-stage deployment strategy of LiMS is expressly suited to assist administrators in developing an understanding of existing resources by allowing administrators to recognize URL patterns and develop policies. Furthermore, it may be valuable to develop common policies for commonly-included third parties; for example, policies designed for Google Tag Manager may be attractive to site administrators and ease adoption of systems like LiMS.

6.2 Performance Overhead

We conducted our performance evaluation on the 450 domains by simulating the deployment of LiMS as in Appendix C, which injects service worker registration scripts in server responses. Next, we measured the overall page load times while progressively activating more components of LiMS to analyze the overhead contributed by each one, under varying network speeds. Table 2 summarizes the different stages of this incremental activation process, starting with a baseline measurement with no SW, and progressing to a full activation with normal LiMS functionality.

The overhead measurements are reported as median page load times in a series of bar charts in Figure 2. Each of the plots represents

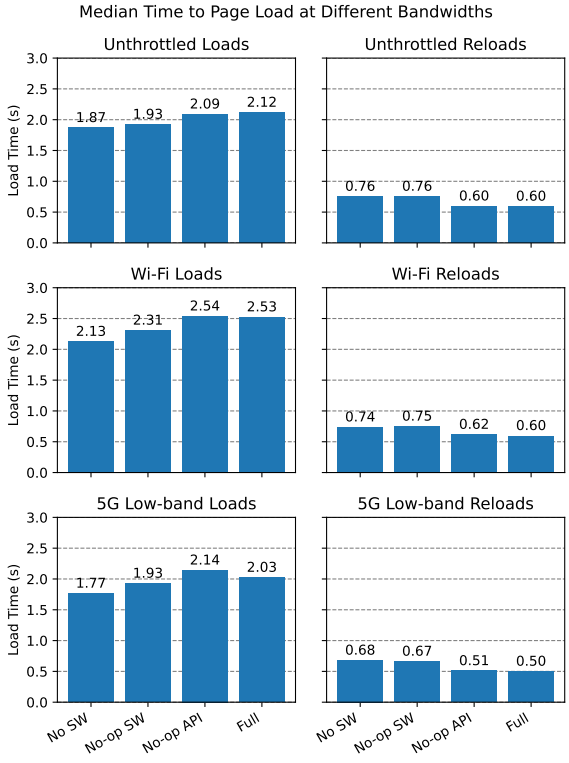

Figure 2: Bar charts depicting the total overhead for users introduced by the LiMS prototype implementation.

a different network configuration, with the left half denoting the median page load times on the first visit, and the right half denoting those for subsequent (second) visits. Each domain is visited 30 times for each combination of the network configuration and the evaluation stage, and the median of these is taken to be the page load time for that domain under those conditions. From the 450 domains after prefiltering, an additional 54 were filtered out because they were missing data for at least one evaluation stage, which were caused by transient failures of the evaluation infrastructure (e.g., unresponsive browsers on evaluation workers) and transient network issues (e.g., timeouts when connecting to the website). The height of each bar in a plot corresponds to the median of the page load times across all remaining 394 domains for the corresponding evaluation stage. For example, in the 5G low-band network profile, the median page load time across the 394 domains is 1.77 seconds in the `No SW` stage.

If we instead consider the 90th percentile of load time overhead, the resulting plot is very similar to Figure 2, except that times are a few hundred milliseconds larger. At the 99th percentile, the same is true, but with greater overhead for load times that take over 2× compared to the 50th percentile of loads (e.g., WiFi has a 2.13s baseline and 2.53s total at the 50th percentile, compared to a 4.92s baseline and 6.26s total at the 99th percentile). Although we did not measure the execution time for each policy verification, proactive re-execution of policy checks can warm the verification cache, preventing user requests from triggering expired policy verifications and improving user experience as mentioned in Section 3.3.

Takeaways. In each of the three network configurations, we observe that the overhead introduced at the last two evaluation stages are significantly higher than the overhead introduced at the `No-op SW` stage. This aligns with expectations: `No-op SW` does not introduce any additional network communication, whereas the `No-op API` and `Full` stages do on the first load of a page. The overhead introduced by a no-op service worker that intercepts requests, and immediately returns from the intercept handler, is 60 ms, 180 ms, and 160 ms for the three network profiles. In contrast, the overall, additional overhead to the median first load time is 250 ms, 400 ms, and 260 ms for the unthrottled, Wi-Fi, and 5G low-band speeds respectively. Despite this, the additional overhead is within the generally-accepted range for performant UI response times [17].

In contrast to the first page load times, the median page reload times are unusual: the median page reload times for the last two stages are lower than those for the first two. On the second load, the `No-op API` and `Full` service worker does not perform any additional network requests: it will consult its local cache and allow the request to pass through. Thus, we hypothesize that this phenomenon is caused by subtle differences in browser behavior when a service worker's request interception logic is not empty, as this phenomenon does not manifest for the `No-op SW` reload measurements. In short, the actual overhead for reloading pages for `No-op API` and `Full` should be similar to `No-op SW`, as the only additional work introduced is the SW checking its local cache.

In summary, intercepting requests with a service worker introduced 60 ms, 180 ms, and 160 ms total overhead to the median page first load time across the three different network profiles. Full operation introduced 250 ms, 400 ms, and 260 ms overhead to the median page first load time for the unthrottled, Wi-Fi, and 5G low-band speeds respectively. Additionally, the overhead on subsequent visits with policies that do not need to be re-verified is negligible.

6.3 Policy Evaluation

LiMS also offers a unique vantage point for administrators to analyze their site: even without custom policies, default policies (§5) can present new insights into the dependency usage of a web application. We discuss several ways that administrators, who may not have the expertise to write customized and robust integrity policies, can use LiMS. In particular, we demonstrate how LiMS can provide insights into dependency usage and generate thresholds for several default policies for a site. The following analyses are performed by components of the open-sourced artifact accompanying this text.

6.3.1 Data Source. To quantify this discussion, we leverage the data archived by the Common Crawl (CC) project [8] to emulate longitudinal analyses. The CC project periodically archives the web with their crawlers and provides its data for public use in an *index* defined by the year and the week in which the crawl was finished (e.g., `2024-42` refers to week 42 in the year 2024). We used 10 contiguous indices spanning approximately 54 weeks from the index `2023-40` to `2024-42`, in order to analyze the links present on domains in a longitudinal manner.

For each index, we attempted to download the corresponding snapshot of the landing page of each domain in the sample from Section 6.2, and extracted all the links in the HTML of the landing page. We filtered this data so that we only consider the 85 domains that were successfully crawled and archived in all 10 snapshots. Various factors contribute to this low number, with the most prominent being that sites may explicitly forbid the CC bots from crawling their site, present an HTTP 301 response that redirects to another URL that was not successfully archived (e.g., `example.com` redirects to the URL `www.example.com` which was not archived), or were not online or crawled at the time. Regardless, the remaining 85 constitute a reasonable sample for this discussion.

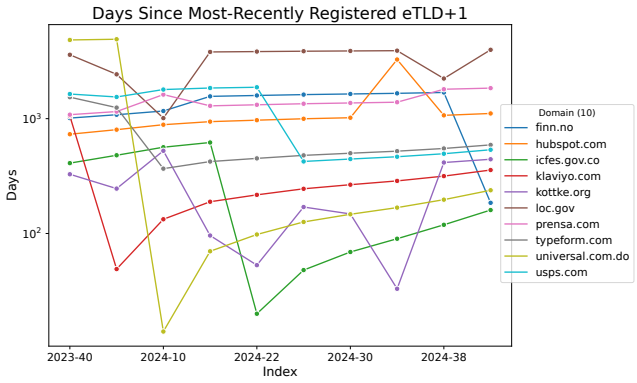

Figure 3: Number of days since the most recent registration of all linked eTLD+1 domains.

6.3.2 Policy: Domain Lifecycle. Figure 3 depicts the thresholds that would be necessary to use the policy described in Section 5.1 on a subset of the 85 domains from the CC dataset. In particular, for each domain at every CC index, we aggregated the eTLD+1 of all linked domains on its landing page, and used a commercial passive DNS database [10] to identify the approximate date when each eTLD+1 domain was registered by using similar methodology as other studies for passive DNS analyses [54]. Then, we approximated the date corresponding to the CC index by taking the first day of the specified week, with weeks corresponding to the ISO 8601 definition [15]. Afterwards, for each of the 85 domains, we computed the number of days since the registration of the most recently registered eTLD+1 among its links. 45 of the domains exhibited monotonically increasing behavior, indicating they never included a newer eTLD+1.

We observe two main patterns in Figure 3, which plots a sample of the 40 remaining domains. The first is that there is a general monotonically-increasing trend for each domain, indicating that domains do not continuously add recently-registered domains to their supply chain. This supports the use of the corresponding policy discussed in Section 5.1, which restricts the use of newly-registered domains. There is one notable exception displayed in the plot: kottke.org. This domain hosts one of the oldest blogs on the web [23] and by its nature, includes links to a multitude of other domains on its landing page, resulting in erratic fluctuations with no discernible trend. The second pattern is that most domains, after including a domain that was more recently registered than all other domains in their supply chains, continue with the general monotonically-increasing trend. This suggests that the domains only periodically include such new domains, thereby implying that if the domain lifecycle policy were deployed, it does not require frequent updates to the threshold.

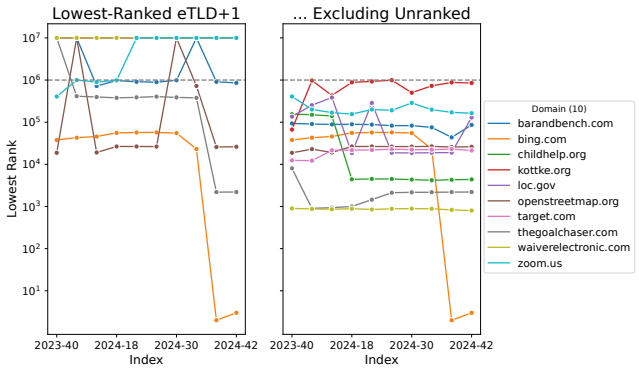

Figure 4: The lowest-ranked domains of all linked eTLD+1 domains. The left plot imputes a default rank for unranked domains, whereas the right excludes them. The dashed line represents the Tranco top 1M.

In fact, the threshold for the duration of time since registration of linked eTLD+1 domains can be automatically and incrementally raised, making it more difficult for adversaries to introduce new domains. If administrators desire to add a new domain, they can temporarily lower the threshold, before continuing to harden the policy threshold. We observe that most domains linked to newer eTLD+1s when introducing new services (e.g., usps.com linking to uspssmartpackagelockers.com 424 days after its registration and universal.com.do linking to asistenciauniversal.om.do only 14 days after its registration), or temporal content (e.g., loc.gov linking to blackhistorymonth.gov in February, approximately 1,011 days after its registration). In the former case, these domains generally keep these services, and this behavior will manifest as a sudden drop and gradual increase in the plot. In the latter case, these domains generally remove these linked eTLD+1s after a certain time window, and this behavior will manifest as a sudden drop, a sudden increase to the same level before the drop, and then continue to gradually increase.

6.3.3 Policy: Domain Ranking. We conduct a similar analysis to examine the feasibility of the policy discussed in Section 5.2. Using the approximate dates for each CC index as previously described, we download the Tranco top 1M ranking list generated on those days, and look up the ranking for each of the eTLD+1 domains that were linked on the landing pages of the 85 CC domains. Figure 4 depicts the results for a sample of the 20 domains that experienced significant changes in their trends, through two perspectives: the left plot, which includes unranked domains by imputing a default rank value of 10^7, and the right, which excludes them.

We observe several patterns in Figure 4 that are similar to those in Figure 3. The first is that the exclusion of unranked domains yields an approximately constant trend for domains that do not introduce new, lower-ranked eTLD+1s. As domains may experience slight changes from one day to the next, the trends will appear approximately constant with slight fluctuations. The second pattern is that when domains introduce domains for new services, these are usually unranked as the domains themselves are new, but may become ranked if they are kept for extended periods of time (e.g., childhelp.org linked to childhelphotline.org since 2023-40, and it was finally ranked 995K on 2024-33). If the lowest-ranked is instead removed, then it will present the opportunity to significantly harden the ranking threshold (e.g., bing.com which removed takelessons.com after 2024-30 and start.gg after 2024-33). Alternatively, if the new linked eTLD+1s correspond to temporal content, they may be removed from the main site before they appear on ranking lists (e.g., loc.gov linked to the unranked jewishheritagemonth.gov on only 2024-22). Regardless of the scenario, administrators can manage the domain ranking policy similarly to the domain lifecycle policy. When new domains are added or removed, the policy threshold, or policy-specific exceptions, can be adjusted, before continuing to gradually harden the threshold.

6.3.4 Summary. Overall, the archived CC snapshots of these domains support the feasibility of the domain lifecycle and domain ranking policies from Section 5. Although we are unable to evaluate the remaining policy building blocks with CC data, or evaluate how well they detect new attacks, we argue that integrity policies in LiMS can provide customizable integrity guarantees in flexible manners, and there exist general policies (e.g., the domain lifecycle and domain ranking) that add valuable protections for users that existing integrity mechanisms cannot.

7 Related Work

To our knowledge, there exists no other work that introduces the concept of granular and flexible integrity policies and an automatic enforcement system that can provides integrity guarantees for clients by blocking network requests to resources whose configured policies are violated. Prior works have introduced the notion of integrity to resources, such as newer versions of Content Security Policy (CSP) and Subresource Integrity (SRI), but the goals, flexibility, and design of such methods are significantly different. This paper marks the first attempt in extending the traditional notion of data integrity in flexible ways that are more fitting for the modern web, enabling integrity to be applied to different dimensions. Further, we provide open source prototype implementations, and evaluate them, as part of this work to hopefully foster more research in this area. In the rest of this section, we discuss prior works that introduce link integrity guarantees, security policies, and malicious JavaScript detection.

Content Security Policy. CSP is the most closely related work to the system proposed in this paper. Originally proposed to defend against Cross-Site Scripting (XSS) attacks [58], it adds an HTTP header that can be included by servers to define allowlists of trusted origins for different resource types, such as scripts, media, and images, on individual pages. Browsers must check all network requests to make sure that the origin of the request URL is included in applicable policies, if specified. If violations are found, browsers generate and send a report to a preconfigured server in the policy header, and can block the request if the policy is configured in the enforcement mode. In summary, CSP can be used to *restrict the resources that are loaded on individual pages to predefined sets of trusted origins.* Furthermore, extensions to its standard have enabled mechanisms to verify the authenticity of inline scripts via nonces and content hashes [69].

The first major difference between the two systems — besides the fact that policies are verified by the browser in CSP, and by a client service worker in LiMS — is the threat model, and subsequently what is considered to be *trusted*. CSP originally considers a resource to be trusted if and only if the resource is loaded from an explicit allowlist

of origins. In short, *CSP trusts certain origins, or exact file contents.* LiMS is concerned with defending against supply chain attacks. The concept of integrity is much more flexible and is reflected in the generalizability of integrity policies. In short, *LiMS trusts in more dimensions than the origin of the provider and exact file contents, including resource behavior, request context, and location information.* Thus, the set of integrity dimensions afforded by CSP is a strict subset of that of LiMS, but this does not mean that LiMS provides the same set of e.g., XSS protections from CSP.

The other major difference relates to *when and where policies are verified.* In CSP, policies are verified by the browser which has received the set of policies for the current page in the HTTP headers. In LiMS, policies are verified on demand by dedicated server-side workers that cache decisions for all users. *Separating the component that performs the actual verification from the browser enables LiMS to use complex policies* and cache its verification decisions for reuse among all users, minimizing the overhead imparted to any individual user.

Subresource Integrity. SRI is another existing standard that provides integrity guarantees of web resources. In contrast to CSP, SRI is expressly designed to combat this problem, and it does so by defining a new attribute in the HTML script tag that specifies the expected hash value(s) for the content of the script [66]. This enables user agents to compare the hash value of the *received* content against the hash value of the content that is *expected*, loading received content if and only if the hash (if specified) matches.

In the same vein as CSP integrity, such guarantees are *strict* by nature and do not allow any type of change in content, regardless of the scope of the change. Prior work has empirically shown that strict integrity is not a good fit to ensure the integrity of JavaScript, one of the primary resource types on the web. Steffens et al. found that high-profile parties randomize select parts of their scripts, rendering it impossible to apply SRI for them [59]. Similarly, So et al. reported that even if every static script were protected with SRI, it would not provide the intended security guarantees because sites often retrieve both static and non-static scripts from third-party origins [52].

The lack of adoption of SRI seems to agree with these studies that challenge the usability and practicality of strict integrity verification. SRI was first proposed as a standard in 2016 by the World Wide Web Consortium [66], and multiple studies conducted over the years have found the adoption rate to be low. Kumar et al. [24] and Shah and Patil [49] both found that less than 1% of sites use SRI in 2017 and in 2018 respectively. Chapuis et al. [5] reported that the adoption had increased to 3.4% of all webpages in 2020 in a longitudinal study with a much larger sample size.

Security Policies. There are a number of prior works that introduce security policies for the web, aside from CSP, to guarantee that JavaScript execution is limited to trusted code. One of the first proposals by Jim et al. to defend against cross-site scripting, Browser-Enforced Embedded Policies (BEEP), introduced the notion of website-defined security policies to be used by the browser to determine which scripts are allowed to run [18]. Another proposal by Oda et al. reimagined web resource usage permissions if cross-origin communications were required to be mutually approved, finding that it would defend against otherwise-successful cross-site scripting (XSS) and request forgery (XSRF) attacks [37]. Other works proposed policy systems that acted directly on JavaScript. Reis et al. proposed Browsershield, a system that intercepts and rewrites JavaScript code subject to site-defined execution policies, aims to avoid executing code that exploits web browser vulnerabilities [42]. Similarly, Meyerovich and Livshits proposed Conscript to grant browsers the ability to enforce fine-grained security policies for JavaScript, effectively adding constraints to the code during execution [31]. Phung et al. proposed FlashJaX, a cross-language inline reference monitor that enforced security policies on third-party, mixed JavaScript and ActionScript content [40], which also used a robust client-side mechanism that did not require browser modifications.

When these designs were proposed more than a decade ago, the primary security incidents were XSS, XSRF, and web browser vulnerabilities, and their considered threat models accordingly reflect this. As such, these systems do not immediately align in the threat model of supply chain attacks where users may receive unexpected or malicious content from trusted origins.

JavaScript Integrity. A different line of related work studies the feasibility of different integrity schemes for JavaScript in terms of producing signatures and fingerprints with program analysis. LiMS is a system designed to offer flexible integrity policies, and an ideal policy would compare the signature or fingerprint of a script against a configured allowlist, using a signature or fingerprint scheme that is robust even in the presence of content changes.

It has proven difficult to produce such a robust, general signature scheme. Strict integrity schemes such as those created by Nakhaei et al. [35] and Mignerey et al. [33] do not address the threat model or require adding a new component to the web public key infrastructure. Soni et al. [55] and Mitropoulos et al. [34] offered novel, relaxed integrity schemes to generate structural signatures and contextual script fingerprints respectively. However, a recent study by So et al. found that these relaxed integrity schemes are unstable in the context of modern web scripts [52].

8 Discussion

This text introduces the concept of integrity policies, an application-agnostic design of a corresponding verification and enforcement system to prevent supply chain attacks, and basic, yet efficient, policies that can be readily implemented and enforced based on commonalities from recent security incidents. Furthermore, we evaluate the overall overhead introduced by our prototype LiMS in the form of page load times for simulated deployments, and evaluate several proposed policies, finding that our prototype LiMS introduced minimal overhead, and that policies can serve as building blocks.

8.1 Limitations

Despite the advantages, there are inherent limitations of our design: LiMS will not be able to intercept WebSocket traffic. Additionally, the LiMS client will also be susceptible to any exploits that leverage the service worker design and implementation (e.g., privacy sniffing [20]). Also, server-side integrity policy verification introduces a non-trivial problem: server-side workers are verifying that remote resources are safe on behalf of clients. There is no guarantee that the server-side workers will receive the same responses as clients. Thus, it may be the case that cloaking of a compromised resource may be able to bypass an integrity policy. Further, policies that are robust in the face of content updates may be difficult to write, but there may

be better-fitting policies that check the integrity of other dimensions for such resources. In addition, the evaluation of our prototype did not take into consideration the execution time of policies, but the overhead is negligible when verifications are performed periodically by verifiers that do not block the request flow. Our evaluation also did not take into consideration the latency between the client and the LiMS server, but techniques such as load-balancing and distribution of hosting servers are expressly designed to tackle this problem.

Additionally, as with all deployed applications, a deployment of LiMS increases the attack surface as the number of nodes in the organization's infrastructure itself will increase. In addition, policy writers will need to cautiously use external libraries, as it may be possible to induce a supply chain attack on a LiMS verifier that relies on external code. However, we remark that the addition of LiMS does not introduce any additional, significant attack surface to website visitors. An attacker that can maliciously modify the LiMS client service worker itself would also be capable of modifying any of the other first-party content, or injecting their own malicious service worker.

8.2 Design Considerations

The prototype LiMS which we introduce in this work has a number of different aspects that can be adjusted. One area is the choice of server-side policy verification — an alternate implementation can include client-side verification for non-content-related policies (because service workers cannot access the contents of cross-origin responses), or directly enable support for content-related policies by implementing the enforcement logic in the browser. This is beneficial because client-side verification guarantees that policies will be verified on the responses that clients receive. By delegating verification to a server-side component, policies must avoid depending on user-specific information in resource requests. However, as previously mentioned, client-side verification may not be ideal as we suspect that it will introduce significant overhead for users, and the current implementation is sufficient for a working prototype.

Another area that can be optimized is the verification protocol between the client SW and the API server. As each newly-observed resource triggers an additional network request in the prototype, it is likely that minimizing the overhead of communications can drastically minimize the overall delay. One method to achieve this is to modify the API server to eagerly respond to navigation requests with the status of resources that are expected to load on the first-party page, or to move the client SW and the API server communication to a WebSocket channel to reduce the overhead of TLS handshakes.

Lastly, additional features can be readily incorporated into LiMS. One feature involves leveraging push subscriptions to send notifications from the server to users, granting the ability to force cache refreshes upon failed or revoked verifications, and also remove the need for the client service worker to regularly poll the API server. Another feature that would prove to be useful is to add support to track the contents of resources to enable longitudinal analyses of content changes for administrators to review when, and how, resources change.

9 Conclusion

In this paper, we introduced the concept of granular and flexible integrity policies that declare the (un)expected properties of resources in different dimensions, and the application-agnostic design of a corresponding verification and enforcement system LiMS. In the design of LiMS, policy verification is performed by dedicated server-side workers who cache decisions for reuse among all users, and policy enforcement is upheld in the form of a service worker that is installed in user browsers. We also introduce universally-applicable integrity policies to serve as building blocks for a comprehensive set of integrity policies and discuss how they could have prevented recent supply chain attacks reported in the industry. Finally, we implemented an open-source prototype of LiMS and found that it adds minimal performance overhead and no loss of functionality to first-party applications during a systematic evaluation of the overhead introduced by the different components. The overall overhead encountered during the first load of a page is several hundred milliseconds, and there is negligible overhead during subsequent loads, for each of the tested network configurations. We also examine the feasibility of several proposed policy building blocks, finding that they suit the dependency usage patterns of sites and would incur minimal overhead for administrators.

Acknowledgments

We thank the reviewers for their helpful comments. This work was supported by the Office of Naval Research (ONR) under grant N00014-24-1-2193 as well as by the National Science Foundation (NSF) under grants CNS-1941617, CNS-2126654, and CNS-2211575.

References

[1] Cesar Anjos. 2018. Malicious Activities with Google Tag Manager. https://blog.sucuri.net/2018/04/malicious-activities-google-tag-manager.html
[2] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. 2010. Building a dynamic reputation system for {DNS}. In *19th USENIX Security Symposium (USENIX Security 10)*.
[3] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. 2011. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2011*. 1–17.
[4] BuiltWith. [n. d.]. CMS technologies Web Usage Distribution. https://trends.builtwith.com/cms
[5] Bertil Chapuis, Olamide Omolola, Mauro Cherubini, Mathias Humbert, and Kévin Huguenin. 2020. An empirical study of the use of integrity verification mechanisms for web subresources. In *Proceedings of The Web Conference 2020*. 34–45.
[6] Joseph Chen. 2019. Mirrorthief Hits Campus Online Stores Using Magecart. https://www.trendmicro.com/en_us/research/19/e/mirrorthief-group-uses-magecart-skimming-attack-to-hit-hundreds-of-campus-online-stores-in-us-and-canada.html
[7] Cloudflare. [n. d.]. How website performance affects conversion rates. https://www.cloudflare.com/learning/performance/more/website-performance-conversion-rates/
[8] Common Crawl. 2024. Common Crawl. https://commoncrawl.org
[9] National Vulnerability Database. 2020. NVD - CVE-2020-27948 Detail. https://nvd.nist.gov/vuln/detail/cve-2020-27948
[10] farsightdnsdb [n. d.]. Passive DNS historical internet database: Farsight DNSDB. https://www.farsightsecurity.com/solutions/dnsdb/
[11] Firefox. [n. d.]. Throttling – Firefox Source Docs documentation. https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/throttling/index.html
[12] Sergiu Gatlan. 2019. Keyloggers Injected in Web Trust Seal Supply Chain Attack. https://www.bleepingcomputer.com/news/security/keyloggers-injected-in-web-trust-seal-supply-chain-attack/
[13] Scott Helme. 2023. 5 Years On: What did we learn from the Government Cryptojacking Attack? https://scotthelme.co.uk/5-years-on/
[14] Bert Hubert, Jacco Geul, and Simon Séhier. 2020. The Wonder Shaper 1.4.1. https://github.com/magnific0/wondershaper
[15] International Organization for Standardization. 2019. ISO 8601:2019 - Date and time format. https://www.iso.org/standard/70907.html
[16] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. 2016. Cloak of visibility: Detecting when machines browse a different web. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 743–758.
[17] Nielsen Jakob. 1993. Response Times: The 3 Important Limits. (1993). https://www.nngroup.com/articles/response-times-3-important-limits/

[18] Trevor Jim, Nikhil Swamy, and Michael Hicks. 2007. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*. 601–610.
[19] Jscrambler. 2022. Defcon Skimming: A new batch of Web Skimming attacks. https://jscrambler.com/blog/defcon-skimming-a-new-batch-of-web-skimming-attacks
[20] Soroush Karami, Panagiotis Ilia, and Jason Polakis. 2021. Awakening the web's sleeper agents: Misusing service workers for privacy leakage. In *Network and Distributed System Security Symposium*.
[21] Yonathan Klijnsma and Jordan Herman. 2018. Inside and Beyond Ticketmaster: The Many Breaches of Magecart. https://web.archive.org/web/20181221151431/https://www.riskiq.com/blog/labs/magecart-ticketmaster-breach/
[22] Krasimir Konov. 2022. Massive WordPress JavaScript Injection Campaign Redirects to Ads. https://blog.sucuri.net/2022/05/massive-wordpress-javascript-injection-campaign-redirects-to-ads.html
[23] Jason Kottke. [n. d.]. About kottke.org. https://kottke.org/about/
[24] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J Alex Halderman, and Michael Bailey. 2017. Security challenges in an increasingly tangled web. In *Proceedings of the 26th International Conference on World Wide Web*. 677–684.
[25] LinkSentry: Link Auditing. 2025. From the Ivory Tower, to Putting It All on Red. https://linksentry.io/blog/from-the-ivory-tower-to-putting-it-all-on-red
[26] Jun Luzon. 2018. Setting offline=true via Network.emulateNetworkConditions does not stop serviceworkers from going online. https://github.com/ChromeDevTools/devtools-protocol/issues/102
[27] Ben Martin. 2021. Magecart Swiper Uses Unorthodox Concatenation. https://blog.sucuri.net/2021/07/magecart-swiper-uses-unorthodox-concatenation.html
[28] MDNContributors. 2023. ServiceWorker - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker
[29] Gal Meiri. 2020. A New Skimmer Uses WebSockets and a Fake Credit Card Form to Steal Sensitive Data. https://www.akamai.com/blog/security/a-new-skimmer-uses-websockets-and-a-fake-credit-card-form-to-steal-sensitive-data
[30] Gal Meiri. 2024. Examining the Polyfill Attack from Akamai's Point of View. https://www.akamai.com/blog/security/2024-polyfill-supply-chain-attack-what-to-know
[31] Leo A Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 481–496.
[32] Trend Micro. 2019. Magecart Delivered Via Advertising Supply Chain. https://www.trendmicro.com/en_us/research/19/a/new-magecart-attack-delivered-through-compromised-advertising-supply-chain.html
[33] Josselin Mignerey, Cyrille Mucchietto, and Jean-Baptiste Orfila. 2020. Ensuring the Integrity of Outsourced Web Scripts. In *ICETE (2)*. 155–166.
[34] Dimitris Mitropoulos, Konstantinos Stroggylos, Diomidis Spinellis, and Angelos D Keromytis. 2016. How to train your browser: Preventing XSS attacks using contextual script fingerprints. *ACM Transactions on Privacy and Security (TOPS)* 19, 1 (2016), 1–31.
[35] Kousha Nakhaei, Fateme Ansari, and Ebrahim Ansari. 2020. JSSignature: eliminating third-party-hosted JavaScript infection threats using digital signatures. *SN Applied Sciences* 2, 1 (2020), 1–11.
[36] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. 2021. A variegated look at 5G in the wild: performance, power, and QoE implications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 610–625.
[37] Terri Oda, Glenn Wurster, Paul C van Oorschot, and Anil Somayaji. 2008. SOMA: Mutual approval for included content in web pages. In *Proceedings of the 15th ACM conference on Computer and communications security*. 89–98.
[38] Ookla. [n. d.]. Speedtest CLI: Internet speed test for the command line. https://www.speedtest.net/apps/cli
[39] Opensignal. [n. d.]. USA, January 2022, 5G Experience Report. https://www.opensignal.com/reports/2022/01/usa/mobile-network-experience-0
[40] Phu H Phung, Maliheh Monshizadeh, Meera Sridhar, Kevin W Hamlen, et al. 2014. Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (2014), 443–457.
[41] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. TRANCO: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2019*.
[42] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. 2007. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)* 1, 3 (2007), 11–es.
[43] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex security policy? a longitudinal analysis of deployed content security policies. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*.
[44] Alex Russell and Jungkee Song. 2022. Service Workers. https://www.w3.org/TR/service-workers/
[45] Matthew Schwartz. 2020. Ticketmaster Fined $1.7 Million for Data Security Failures. https://www.bankinfosecurity.com/ticketmaster-fined-17-million-for-data-security-failures-a-15369
[46] Jérôme Seguar. 2019. New evasion techniques found in web skimmers. https://www.malwarebytes.com/blog/news/2019/12/new-evasion-techniques-found-in-web-skimmers
[47] Jérôme Seguar. 2020. Credit card skimmer masquerades as favicon. https://www.malwarebytes.com/blog/news/2020/05/credit-card-skimmer-masquerades-as-favicon
[48] Jérôme Seguar. 2020. Web skimmer hides within EXIF metadata, exfiltrates credit cards via image files. https://www.malwarebytes.com/blog/news/2020/06/web-skimmer-hides-within-exif-metadata-exfiltrates-credit-cards-via-image-files
[49] Ronak Shah and Kailas Patil. 2018. A measurement study of the subresource integrity mechanism on real-world applications. *International Journal of Security and Networks* 13, 2 (2018), 129–138.
[50] João Marco Silva, Diogo Ribeiro, Luis Felipe Ramos, and Vítor Fonte. 2024. A worldwide overview on the information security posture of online public services. In *Proceedings of the 57th Hawaii International Conference on System Sciences*.
[51] Denis Sinegubko. 2023. 40 New Domains of Magecart Veteran ATMZOW Found in Google Tag Manager. https://blog.sucuri.net/2023/12/40-new-domains-of-magecart-veteran-atmzow-found-in-google-tag-manager.html
[52] Johnny So, Michael Ferdman, and Nick Nikiforakis. 2023. The More Things Change, the More They Stay the Same: Integrity of Modern JavaScript. In *Proceedings of the ACM Web Conference 2023*. 2295–2305.
[53] Johnny So, Najmeh Miramirkhani, Michael Ferdman, and Nick Nikiforakis. 2022. Domains Do Change Their Spots: Quantifying Potential Abuse of Residual Trust. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 119–133. doi:10.1109/SP46214.2022.00008
[54] Johnny So, Iskander Sanchez-Rola, and Nick Nikiforakis. 2025. Lost in the Mists of Time: Expirations in DNS Footprints of Mobile Apps. In *Proceedings of the 34th USENIX Security Symposium* (2025). https://johnny.so/publication/so-2025-lost/so-2025-lost.pdf (to appear).
[55] Pratik Soni, Enrico Budianto, and Prateek Saxena. 2015. The SICILIAN defense: Signature-based whitelisting of web JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1542–1557.
[56] Puja Srivastava. 2025. Cascading Redirects: Unmasking a Multi-Site JavaScript Malware Campaign. https://blog.sucuri.net/2025/03/cascading-redirects-unmasking-a-multi-site-javascript-malware-campaign.html
[57] Puja Srivastava. 2025. Google Tag Manager Skimmer Steals Credit Card Info From Magento Site. https://blog.sucuri.net/2025/02/google-tag-manager-skimmer-steals-credit-card-info-from-magento-site.html
[58] Sid Stamm, Brandon Sterne, and Gervase Markham. 2010. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*. 921–930.
[59] Marius Steffens, Marius Musch, Martin Johns, and Ben Stock. 2021. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In *Network and Distributed Systems Security (NDSS) Symposium 2021*.
[60] Eliya Stein. 2021. Tag Barnakle One Year Later: 120+ More Revive Adserver Hacks. https://blog.confiant.com/tag-barnakle-one-year-later-120-more-revive-adserver-hacks-f3e5b3bc8e70
[61] Nati Tal and Oleg Zaytsev. 2023. "EtherHiding" — Hiding Web2 Malicious Code in Web3 Smart Contracts. https://labs.guard.io/etherhiding-hiding-web2-malicious-code-in-web3-smart-contracts-65ea78efad16
[62] Sansec Forensics Team. 2020. Sansec reveals longest Magecart skimming operation to date [Analysis]. https://sansec.io/research/longest-skimming-operation-yet
[63] The BlackBerry Research & Intelligence Team. 2023. Silent Skimmer: Online Payment Scraping Campaign Shifts Targets From APAC to NALA. https://blogs.blackberry.com/en/2023/09/silent-skimmer-online-payment-scraping-campaign-shifts-targets-from-apac-to-nala
[64] W3C. 2015. Content Security Policy Level 1. https://www.w3.org/TR/CSP1/
[65] W3C. 2016. Content Security Policy Level 2. https://www.w3.org/TR/CSP2/
[66] W3C. 2016. Subresource Integrity. https://www.w3.org/TR/SRI/
[67] Taojie Wang, Jin Chen, and Tao Yan. 2022. A New Web Skimmer Campaign Targets Real Estate Websites Through Attacking Cloud Video Distribution Supply Chain. https://unit42.paloaltonetworks.com/web-skimmer-video-distribution/
[68] UDN web docs backup. [n. d.]. CSP: require-sri-for - HTTP. https://udn.realityripple.com/docs/Web/HTTP/Headers/Content-Security-Policy/require-sri-for
[69] Mike West. 2024. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/
[70] JunDong Xie. 2021. New Attack Surface in Safari: Using Just one Web Audio vulnerability to rule the Safari. https://i.blackhat.com/asia-21/Friday-Handouts/as-21-Xie-New-Attack-Surface-In-Safari-Use-Just-One-WebAudio-Vulnerability-To-Rule-Safari-wp.pdf

A High-Level Policy Building Blocks

This section contains a series of algorithms that correspond to the proposed policy building blocks described in Section 5. In particular, Algorithm 5 corresponds to Section 5.2; Algorithm 6 corresponds

Algorithm 8 A sample integrity policy that denies requests for resources with TLS connection errors or SRI violations.

```
Class PolicySRIViolations
  Variable locationSource = "example.com/.*"
  Variable locationTarget = ".*"
  Variable logic = FailsSRICheck
  Variable output = False

procedure FailsSRICheck(req)
    page ← req.page
    matchesSRI ← true
    sriDigest ← GetExpectedSRIDigest(page, req)
    matchesSRI ← sriDigest != null AND MatchesSRI(req, sriDigest)
    return not matchesSRI
end procedure
```

Algorithm 9 A sample integrity policy that denies requests if the remote server is in an unexpected location.

```
Class PolicyInfrastructureAttributes
  Variable locationSource = "example.com/.*"
  Variable locationTarget = ".*"
  Variable logic = IsInUnexpectedLocation
  Variable output = False

procedure IsInUnexpectedLocation(req)
    distanceThreshold ← GetDistanceThreshold()
    ip ← GetRemoteIP(req)
    loc ← GetServerLocation(req)
    if IsExpectedCountry(loc.country) then
        return True
    end if                          ▷ Reference location can be adjusted
    return GetDistance(loc) >= distanceThreshold
end procedure
```

Algorithm 10 A sample integrity policy that denies requests if modifications to client-side, or server-side, core files are detected.

```
Class PolicyCoreFiles
  Variable locationSource = "example.com/.*"
  Variable locationTarget = "example.com/core/.*"
  Variable logic = HasCoreFileChanged
  Variable output = False

procedure HasCoreFileChanged(req)
    expected ← GetExpectedContent(req)
    actual ← GetActualContent(req)
    return expected != actual
end procedure
```

Table 3: Domain sample categorization.

Category	Count
Technology/Software/Cloud	127
Other/Uncategorized	115
E-commerce/Retail	50
News/Media/Publishing	48
Government/Non-profit/Educational	45
Entertainment/Adult Content	56
Advertising/Marketing/Tracking	39
Financial/Banking	23
ISP/Telecommunications	20
Social Media/Communication	19
Gambling/Betting	16
Health/Medical	13
Forums/Communities	12
Search/Portal	10
Travel/Hospitality	7

to Section 5.3; Algorithm 7 corresponds to Section 5.4; Algorithm 8 corresponds to Section 5.5 They refrain from technical details as they are not critical to the goals of this paper, which introduces LiMS. However, interested readers can find their implementations in the prototype that will be in the accompanying artifact submission.

Algorithm 5 A sample integrity policy that denies requests to low-ranking domains, unless explicitly allowlisted.

```
Class PolicyLowRanked
  Variable locationSource = "example.com/.*"
  Variable locationTarget = ".*"
  Variable logic = IsLowRanked
  Variable output = False

procedure IsLowRanked(req)
    thresh ← GetRankingThreshold()
    lowRanked ← IsLowRanked(req.domain, thresh)
    allowlisted ← IsAllowListedForRanking(req.domain)
    return lowRanked AND NOT allowlisted
end procedure
```

Algorithm 6 A sample integrity policy that denies requests for resources that are flagged by threat intelligence services or displays suspicious behavior such as camouflaging or cloaking.

```
Class PolicyThreatIntelligence
  Variable locationSource = "example.com/.*"
  Variable locationTarget = ".*"
  Variable logic = IsFlaggedByThreatIntel
  Variable output = False

procedure IsFlaggedByThreatIntel(req)
    isFlagged ← CheckThreatIntel(req)
    usesCamouflaging ← RunCamouflagingDetection(req)
    return isFlagged OR usesCamouflaging
end procedure
```

Algorithm 7 A sample integrity policy that denies requests for resources whose dependencies have changed.

```
Class PolicyChangedDependencies
  Variable locationSource = "example.com/.*"
  Variable locationTarget = ".*"
  Variable logic = HasChangedDependencies
  Variable output = False

procedure HasChangedDependencies(req)
    cached ← GetPreviousDependencies(req)
    curr ← GetCurrentDependencies(req)
    return cached == curr
end procedure
```

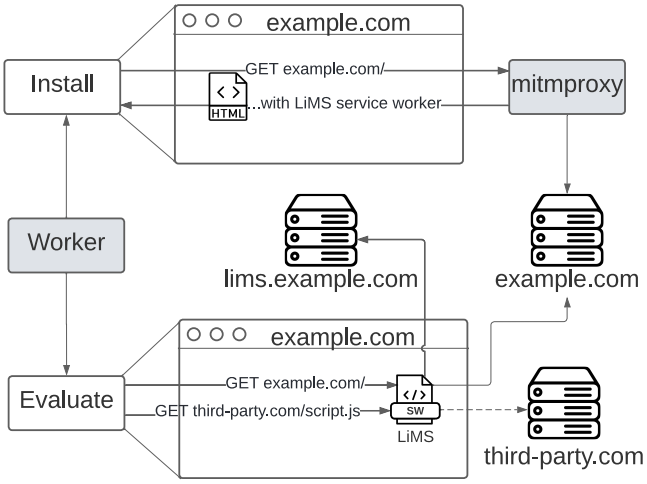

Figure 5: Diagram depicting the simulated deployment setup to bootstrap the LiMS client service worker for a live website.

B Domain Sample Categorization

The domain categories in Table 3 were labeled by the free version of Google Gemini with the 2.5 Flash model. Categories were assigned based on inferences made from domains mentioned in its training data and domain keyword analysis.

C Simulated Deployment

We designed an evaluation pipeline to measure the performance overhead introduced by our system for an arbitrary website by simulating a deployment of LiMS: the pipeline deploys the backend system components and leverages a man-in-the-middle proxy to inject the client SW in the HTML responses from the target site, as depicted in Figure 5.

For a given site, an evaluation worker visits it with Google Chrome for Testing v128.0.6613.137 over a series of four different stages: the first with no service worker, and the next three corresponding to the LiMS prototype operating under each evaluation stage in Table 2, with at least 12 hours between two evaluation stages for any individual domain. Each stage consists of visiting a domain, twice, with the configured evaluation mode. The first visit bootstraps the client SW installation by leveraging a man-in-the-middle proxy to inject the client SW into HTML responses. The browser's network cache and the service worker's local cache are cleared before relaunching the browser, without the proxy. In this manner, the second visit is able to use the service worker that was previously injected and installed in the first visit. In addition, we ensure that the browser components responsible for handling service workers are in a warm state by first visiting a website (not part of the evaluation set) that already uses a service worker, before visiting the target site. After this, the browser proceeds to visit the target domain after a short delay, and reloads the page, while tracking these page load times to measure overhead.

Incremental Overhead. The evaluation stages in Table 2 progressively utilize more of the core functionality available in LiMS, and the last stage emulates a real deployment environment. The first stage `No SW` measures the page load time with no service worker to function as a baseline. The second stage `No-op SW` is designed to measure the overhead of a bare-bones service worker that does nothing but intercept network requests and immediately allow them to continue, and corresponds to reaching step 5 in Figure 1. The third stage `No-op API` is where we expect significant overhead — for each new intercepted request, the client SW must send a queryStatus request to the API server who immediately responds "allowed", corresponding to immediately after step 6 in Figure 1. The last stage `Full` adds additional overhead by having the API perform database reads and determining whether requests should be allowed.

Default Policies. We configure a set of default policies from Section 5 for each target site, to simulate an actual deployment as closely as possible. In particular, we install policies that verify that:

- domains were not registered within the last 7 days,
- domains are not due to expire in the next 7 days,
- domains rank within the top 1M on Tranco, and
- the set of requests initiated by links do not change, and
- there are no TLS connection issues.

We warm the LiMS system state by bootstrapping these policies, and their verifications, to ensure that there will always be valid verifications for links. In addition, to avoid influencing the performance overhead evaluation, we specially modify the API server to always respond yes to client SW queries, regardless of the actual verifications. During our performance evaluation experiments discussed in Section 6.2, we encountered several types of failed verification with these default policies. The most common failures were caused by the default ranking policy: low-rank domains often contained links to other low-rank domains, with some not ranked in the top 1M on Tranco. The other most common failure was caused by the default domain dropping policy which reported several third party domains that were due to expire. The most popular among such domains, adroll.com, is likely an advertising service provider used by popular sites such as x.com, macys.com, synology.com, datadoghq.com, and okta.com, and it was re-registered in time.

Network Profiles. Furthermore, as our system introduces additional network requests, we varied the network configurations to investigate whether there exists any undesired effects at slower speeds. We performed our overhead measurements in Section 6.2 at three common network speeds, while keeping the API server location constant, with an average latency of 5.8 ms and average packet loss of 0.53% over 100 packets, as reported by ping from the evaluation nodes. The first configuration is the unthrottled 82 Mbps download and 44 Mbps upload speeds for the virtual machines in our institution's network; the second emulates the lower bounds of a Wi-Fi connection with 30 Mbps download and 15 Mbps upload speeds; the last emulates the lower bounds of a 5G low-band connection with 50 Mbps download and 10 Mbps upload speeds. The unthrottled speeds vary depending on network activity from other tenants on the same physical host, and the numbers we present were obtained from computing the median of ten trials of the Ookla speedtest-cli tool [38] on each worker VM, on all VMs simultaneously. We sourced the speeds for the Wi-Fi connection from its corresponding browser throttling profiles [11], and the 5G low-band connection from industry reports [39] and prior 5G performance measurements [36].

We used wondershaper [14], a wrapper for the traditional Linux traffic shaper tool tc, to control the upload and download speeds from

each of our evaluation nodes. We opted to use wondershaper instead of the network throttling feature built into the browser because its throttling implementation does not automatically affect installed service workers [26], and we were unable to directly throttle the service worker context with the Chrome DevTools Protocol.