What I See Is What You Leak: Detection and Measurement of Browser-Visible Secrets in the Wild

Jiawei Zhou[1], Lingyun Ying[1,*], Zidong Zhang[1], and Yacong Gu[2,3]
[1]*QI-ANXIN Technology Research Institute;* [2]*Tsinghua University;* [3]*Tsinghua University-QI-ANXIN Group JCN*
*Email: {zhoujiawei01, yinglingyun, zhangzidong}@qianxin.com, guyacong@tsinghua.edu.cn; *Corresponding author*

***Abstract*—Modern web applications execute service-facing logic in the browser, where front-end code calls backend and third-party services directly. Some of the secrets these calls require are public by design, such as front-end secrets for maps or payments secured by usage restrictions rather than secrecy; others are server-only secrets that build pipelines inadvertently ship to the browser. Once misconfigured or leaked, either can let an attacker abuse the services or steal sensitive data. However, existing secret scanners are primarily built for source code, not browser-visible files: minification erases the variable-name anchors their rules need, and secret-shaped noise (SDK identifiers, build hashes, etc.) floods the same files.**

In this paper, we present KeySifter, the first detection-and-measurement pipeline for browser-visible secrets. It addresses two challenges. The first is collection: modern web applications ship *shadow assets*, build artifacts that no web page links to and crawlers miss. By analyzing how ten mainstream web frameworks produce such assets, KeySifter extracts probing rules to collect them. The second is detection: KeySifter pairs rule scanning with AST call-site backtracking to catch secrets hidden behind minified aliases, then classifies each candidate against a risk-pattern knowledge base to separate real secrets from noise. KeySifter reaches 93.6% precision and the highest recall among evaluated tools (81.7%), versus below 0.1% for other rule scanners. We apply KeySifter to 65,000 domains (8,393,191 files) sampled from the Tranco top 1M, identifying 1,244 secret leakage issues across 1,437 domains. Three findings emerge. (i) Leakage is non-monotonic with site popularity, peaking in the mid-tier (Tranco 10K–100K) and lowest at the head. (ii) Shadow assets are a non-negligible leakage source: 18.3% of leaks come from them, exposing secrets that no rendered page references. (iii) Public-by-design secrets are widely misconfigured: of 2,684 exposed API secrets, 1,010 (37.6%) are callable with no restriction on at least one service, and 137 reach high-value services such as Vision and Gemini. We disclosed all responsibly; 53 cases have been confirmed or remediated, including a Tranco top-200 consumer-electronics vendor and a top-3K travel platform.

1. Introduction

Modern apps rely on secrets to access backend and third-party services: API credentials, OAuth client secrets, signing keys, and access tokens gate access to cloud platforms, payment processors, and AI back ends. Once such a secret is exposed, attackers may act as the app, consume the owner's quota, or reach protected data. The scale is substantial: GitGuardian reported 23.8 million new secrets in public GitHub commits in 2024 alone [20]; stolen or compromised credentials have been the entry point in roughly a third of breaches over the past decade [63]; and the average breach costs $4.88 million [31].

As the software supply chain has grown, secrets leak at every stage, and detection research has followed them stage by stage. Leakage begins at the *source repository*: developers hard-code secrets and push them to public hosting, as in the 2016 Uber breach of 57 million records [61] and the Toyota access key left public for nearly five years [57]. When code is packaged for distribution, the secrets ship inside the *artifact*: in mobile apps, where hard-coded cloud credentials and AppSecrets leak at scale [56], [75], [72], [74], and in container images, where an Internet-wide scan found secrets in 8.5% of them [12]. When packaging is automated, the *build and supply chain* becomes the next surface, exposing the secrets wired into continuous-integration environments [10]. Across these stages, detection relies on matching secrets that carry a known prefix or a fixed structure [44], [49], [5].

However, the browser-visible web breaks this recipe in three ways. First, the corpus is not always in hand the way it is for a cloned repository, an unpacked artifact, or a build environment read in full. A web app is not delivered in one piece; its code and configuration are scattered across lazily loaded routes, build outputs, source maps, and conventional paths that no rendered page links to. Second, the variable names and service prefixes the rules key on do not always survive into the shipped text. Minification erases this anchor: bundlers rename identifiers to short aliases, so name-anchored scanners [58], [22] no longer match, even though the value sits in the file verbatim. Third, a format-level match does not mean the matched string is a real secret. Packaged web artifacts break this rule: their HTML, JavaScript, and JSON teem with harmless identifiers shaped like secrets (build hashes, resource IDs, etc.), alongside SDK keys whose presence on the page is public by design [27]. In our crawl, barely one rule match in about 3,000 is a genuine secret.

Prior measurements [14] of browser-visible secrets stop short in three ways. First, they stop at the landing page, yet 63.25% of the secrets we observe live in subpage routes, build artifacts, source maps, and probed paths that no

homepage scan reaches. Second, they key on a handful of structured secret formats with fixed prefixes and therefore cannot adjudicate the large class of keys that carry no such prefix, which covers most candidate matches and most of the secrets we confirm. Third, some keys designed to be public in browser-visible assets are not safe; without proper configuration, the corresponding service can be abused, as illustrated by a disclosed billing-abuse incident of $15,400–$128,000 [60], [8]. To the best of our knowledge, our work is the first system that jointly combines framework-aware collection of shadow assets, minification-resilient secret recovery, and web-scale measurement of both private-by-design leaks and public-secret misconfiguration.

Our Work. We designed and implemented KeySifter, a pipeline that addresses these gaps. To recover the assets from a landing-page scan that never loads, we analyze the build artifacts of ten mainstream web frameworks; guided by what each framework hides, we crawl web assets at depth and additionally probe paths that framework conventions make predictable yet no rendered page references. To widen the coverage of recoverable secrets, we extract candidates with a rule scan and recover values that minification renamed by backtracking through their JavaScript call sites. To separate the genuinely threatening secrets from the vast string noise in HTML, bundled JavaScript, and related assets, we parse each candidate's structure, assemble its surrounding context, and match it by similarity against a risk pattern knowledge base. To characterize secret leakage across the web ecosystem, we run KeySifter as a measurement over a sample of the Tranco top 1M. Finally, to study how public-by-design secrets are misconfigured in practice, we validate whether each exposed secret is actually usable in an ethical manner, and we track how long that exposure persists by re-checking the same secrets weeks later.

Findings. Applied to a crawl of 65,000 Tranco-sampled domains and 8,393,191 files (644 GB), KeySifter identifies 1,244 secret leakage issues across 1,437 domains. Five findings are noteworthy.

(i) Existing scanners are effectively unusable on the web ecosystem. With public SDK identifiers, build hashes, and resource IDs swamping the real secrets, traditional scanning matches carry a 1:3,266 signal-to-noise ratio. Tools that equate "matches a format" with "is a secret" therefore stay below 0.1% precision here.

(ii) Browser-visible everyday files are the main carrier of secret leakage. Across the 234 private-by-design secrets on 192 domains, the everyday assets a browser loads (HTML, JavaScript, and JSON) carry 90.74% of all leaked occurrences, while sensitive dotfiles such as `.env` carry few. This means that watching only sensitive files and overlooking everyday assets would miss most of the leakage.

(iii) Most exposure is invisible to a landing-page scan. Only 36.75% of the private-by-design secrets are reachable from the landing page; the rest surface only through deeper crawling and targeted probing of assets that no rendered page links to. These values remain browser-visible, but a landing-page-only scan does not detect them.

(iv) Public-by-design does not mean safe. Of 2,684 API keys, 37.60% are callable with no restrictions and are open to any anonymous caller. Tracking the same keys weeks after disclosure, only 1.4% of these were repaired, and some keys had their permissions widened, deepening the harm.

(v) Leakage recurs across domains and concentrates unevenly. 6.41% of private-by-design secrets recur on more than one domain, and one spans 16, suggesting propagation through shared SDKs or templates. Leakage concentrates in mid-ranked domains, possibly because they lack security mechanisms common among top-ranked sites while running more complex frameworks and APIs than low-ranked domains.

Disclosure. We disclosed the confirmed exposures responsibly. To date, we have received 53 confirmations or remediations for the secrets we disclosed, including a Tranco top-200 consumer-electronics vendor and a Tranco top-3K travel platform.

Contributions. Our contributions are as follows.

- **A framework-aware collection method for browser-visible web assets.** We combine link-driven collection with derived-resource expansion, dictionary probing, and fingerprint-driven probing, guided by a controlled pilot over ten mainstream web framework families. Where prior measurements stop at the landing page, we recover the 63.25% of secrets that lie beyond it; 18.3% of the secret-bearing assets are shadow assets that no rendered page references at all.
- **A high-precision secret detector for the web ecosystem.** KeySifter recovers values that minification renamed by backtracking through their JavaScript call sites, then adjudicates the format-ambiguous candidates that no rule can decide against a knowledge base of credential patterns. Where rule scanners built for source code stay below 0.1% precision on this surface, KeySifter reaches 93.6% precision and 81.7% recall on our evaluation set.
- **A systematic measurement of browser-visible secret exposure.** Applying KeySifter to a Tranco-sampled crawl, we characterized the exposure along four axes: which asset types carry the risk, how much each crawl strategy adds beyond the landing page, which credential classes dominate, and how far a single leak propagates across domains.

Research Questions. We organize the measurement around four questions. **RQ1** identifies which browser-visible asset types carry risky secrets. **RQ2** compares the dominant secret classes and their risk rates. **RQ3** measures what each crawl tier adds beyond a landing-page scan. **RQ4** traces how an exposed value propagates across domains and assets.

Open Science. We have released KeySifter to support future research.[1]

2. Background and Threat Model

2.1. Background

Web secret leakage differs structurally from source code repository leakage: a source code scanner inspects files as

1. https://github.com/XingTuLab/KeySifter

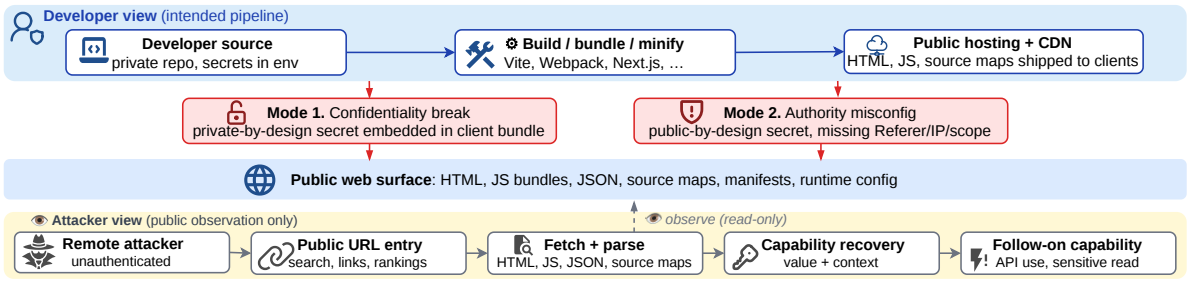


Figure 1: Threat-model overview.

the developer writes them, whereas browser-visible leakage concerns what the build pipeline ships to the browser. These shipped assets are *carriers*: browser-retrievable, text-shaped bodies that can hold secrets, such as HTML, JavaScript bundles, and JSON config. Some are not reachable from the landing page but are fetched only after loading, and a few are *shadow assets* that no page references at all; thus, *browser-visible* does not imply directly user-visible.

From source to browser. A modern site does not ship the developer's source as written. Source code and configuration live in a private repository; a build step (e.g., Vite, Webpack, Next.js) then bundles and renders them into the assets it ships, which a web server (commonly nginx) publishes and a Content Delivery Network (CDN) forwards to visitors. The CDN is only a transit layer that relays whatever the origin exposes. A server-side secret becomes browser-visible chiefly at two points along this path: the build stage may inline it into the delivered code, or the serving stage may expose it as a file when the web server is rooted at the project tree rather than the build output (e.g., `dist/`) alone.

2.2. Threat Model

Figure 1 organizes the setting into three layers: the developer pipeline (top), the *public web surface* (middle), and the attacker (bottom). The public web surface consists of carriers retrievable by an anonymous visitor; an attacker can retrieve such resources with standard browser requests.

Attack Goal. The adversary aims to extract credentials from public carriers that enable unauthorized service use, including invoking paid or privileged APIs, accessing or modifying backend data, or impersonating the victim across trusted services. No server compromise is required.

Attacker Capability. The attacker is a remote, unauthenticated party that interacts solely by issuing anonymous HTTP(S) requests. They can obtain browser-visible assets through page navigation and resource probing. They can either target specific domains for focused crawling or perform large-scale collection by leveraging URL ranking lists (e.g., popular domain rankings). Each recovered credential can then be abused according to the capability it grants.

Scope and Assumptions. To analyze secret leakage from the vantage point of an anonymous visitor as faithfully as possible, all analysis relies solely on public resources and reproducible read-only interactions; we restrict all credential probing to non-destructive requests, so measured impact represents a conservative lower bound on attacker capability. Out of scope: post-login content, authenticated APIs, private repositories, CI artifacts, server-side compromise, and any destructive validation.

2.3. Definitions

To study secret leakage in the web ecosystem, we define the following terms.

- ***Private-by-design secret***: a server-side credential whose security rests entirely on value secrecy. It should never leave the server, so its arrival in the browser is itself the failure, a confidentiality break (*Mode 1*); cloud credentials and private-key material are typical instances.
- ***Public-by-design secret***: a credential designed to be browser-visible, whose security rests not on secrecy but on service-side safeguards (Referer/IP restrictions, quotas) that bound what the value can call. It fails when those safeguards are left unset, an authority misconfiguration (*Mode 2*): the capability widens from "public but constrained" to "public and unrestricted." Google Maps API keys and Firebase config are typical instances.
- ***Non-secret noise***: credential-shaped strings that are not credentials, such as placeholders, UUIDs, and bundle hashes. These cross no security boundary and are excluded from our findings.

A public-by-design secret is already there by design; what fails is its configuration, not its containment. The two cross-boundary arrows in Figure 1 (Modes 1 and 2) trace each kind's path from the pipeline to the surface.

Counting Granularities. We report secret counts at four granularities, used consistently throughout the paper:

- **Raw record:** output may contain the same secret appearing multiple times in different places.
- **Deduplicated occurrence:** one `(file, secret)` pair, collapsing within-file repeats but preserving cross-file copies.
- **Unique value:** one distinct secret after deduplication across files and domains.
- **Affected domain:** one registrable domain on which at least one secret appears.

3. Data Collection

Studying browser-visible resources at web scale first requires knowing what an anonymous visitor can actually receive from a site. That denominator is not directly observable: a live site never reports how much it withholds, and a single landing-page fetch undercounts it, because a modern page is a lazy-evaluated build artifact that reveals only a fraction of itself on first load. We recover the denominator in a controlled pilot over ten mainstream web framework families, where the build output supplies ground truth, and let the misses we observe there define the crawl.

The pilot sorts those misses into two kinds. Link-driven crawling follows the references a rendered page exposes; prior-driven crawling reconstructs the artifacts it hides. Post-processing then reconciles the collected corpus against the pilot ground truth, so that the coverage we later report in the wild is a measured quantity rather than a best effort.

3.1. Pilot: What a Crawl Misses, and Against What

Prior webpage crawlers, from general-purpose archivers to recent research systems (Heritrix [32], OpenWPM [16], Sprinter [25], Demir [14]; Table 1), all aim to collect as much of a site as they can, but none can state how much they miss: in the wild, completeness has no denominator. We construct one. We select ten framework families for their popularity and architectural diversity, covering client-side rendering, server-side rendering, and static-site generation. In the Wappalyzer statistics used for selection, these families account for 61.73% of framework usage across 83 tracked families [65]. We use GitHub stars to select two popular open-source apps from each family (Table 2), then build and deploy the resulting 20 apps. We derive from each build artifact the set of all browser-reachable analyzable assets, then crawl each deployment with the same anonymous configuration we use in the wild. With ground truth in hand, every miss is attributable to the mechanism that produced it, so the pilot yields more than a recall number: it exposes the *structure* of the gap.

The pilot shows that build decisions determine which assets an anonymous visitor can reach. A build that emits a legacy fallback bundle that modern browsers never request places that bundle outside the default crawl path. Three consequences follow, and the pipeline is built around them.

First, because each loss traces to a deterministic framework behavior rather than to chance, misses cluster on a few nameable mechanisms (Table 3, right column) instead of scattering. This is why recall ranges so widely across frameworks: `vue-vben-admin` leaves 61 of its 138 chunks behind login-gated routes, whereas the Angular apps are recovered nearly in full because their lazy-loaded chunks sit on routes reachable without login. A deterministic mechanism admits a targeted recovery step, not blind depth, and this is what link-driven crawling exploits.

Second, the build tells us which misses matter. It separates the secret-bearing carriers we must recover from gaps that are out of scope by design, chiefly locale resources and login-gated content, which dominate the residual yet cannot carry an anonymous leak. The comparison also reveals the opposite blind spot: the crawl captures externally hosted dependencies and runtime-only endpoints that no build-derived ground truth contains. Following references is insufficient, so we also construct URLs from framework conventions. Prior crawlers collect only *in-browser* resources; we additionally probe *out-of-browser* assets through derived-resource expansion, dictionary probing, and fingerprint-driven probing (Table 1; unsupported cells reflect each tool's design scope, not a shortcoming).

Third, the same determinism that hides assets also manufactures noise. A SPA fallback that returns the landing shell for arbitrary paths produces thousands of soft-404 hits (HTTP success with no real asset). Because this noise follows from one framework behavior, post-processing removes it by rule rather than by heuristic.

TABLE 1: Crawl-capability comparison with previous work.

Crawl capability	Heritrix [32]	OpenWPM [16]	Sprinter [25]	Demir's [14]	Ours
Browser-grounded crawling					
Headless browser execution	❍	✓	✓	✓	✓
Same-origin BFS (sitemap, DOM links)	✓	✗	❍	✗	✓
Beyond-browser collection					
Source map + build manifest derivation	✗	✗	✗	✗	✓
Active probe of `/.env`, `/.git`, config	✗	✗	✗	✗	✓
Framework-aware artifact reconstruction	✗	✗	✗	✗	✓

✓ full support ❍ partial / heuristic ✗ not supported.

TABLE 2: Pilot testbed: 20 popular apps across 10 frameworks.

Family / Render‡	Projects (GitHub stars)
React, CSR	bulletproof-react (34.6k), react-redux-realworld (5.6k)
Vue, CSR	vue-vben-admin (31.7k), vue3-realworld (1.1k)
Angular, AOT	coreui-angular-admin (1.8k), ng-matero (1.5k)
Preact, SPA	macos-web (2.6k), preact-www (378)
SolidJS, CSR	solid-realworld (236), solid-hackernews (167)
Next.js, SSR	tailwind-nextjs-blog (10.4k), next-app-playground (2.9k)
Nuxt 3, SSR	elk (6.0k), nuxt-movies (2.2k)
SvelteKit, SSR	svelte-realworld (2.4k), svelte-hackernews (162)
Astro, SSG	astrowind (5.5k), astro-paper (4.3k)
Vanilla, no-build	vanilla-todomvc (28.9k), alpine-admin (358)

‡: CSR, client-side rendering; AOT, ahead-of-time compilation; SSR, server-side rendering; SSG, static-site generation; SPA, single-page app; no-build, source files served without a bundling step.

TABLE 3: Pilot results: per-family full-crawl recall and the dominant residual mechanism. ‡ marks out-of-scope gaps.

Family	Render	Recall	Dominant residual mechanism
Vanilla	no-build	100%	no build; CDN dependencies
SolidJS	minimal	95%	single dynamic chunk
Angular	AOT	93%	i18n JSON‡
Nuxt 3	hashed	91%	hashed build ID; i18n‡
React	split	84%	login-gated chunks‡
Next.js	chunked	83%	unvisited route chunks
Preact	SPA	71%	i18n‡; deep documents
SvelteKit	dual	70%	`nomodule` legacy bundle
Vue (admin)	fine-grain	69%	component chunks; login‡
Astro	islands	61%	deep pages beyond BFS depth

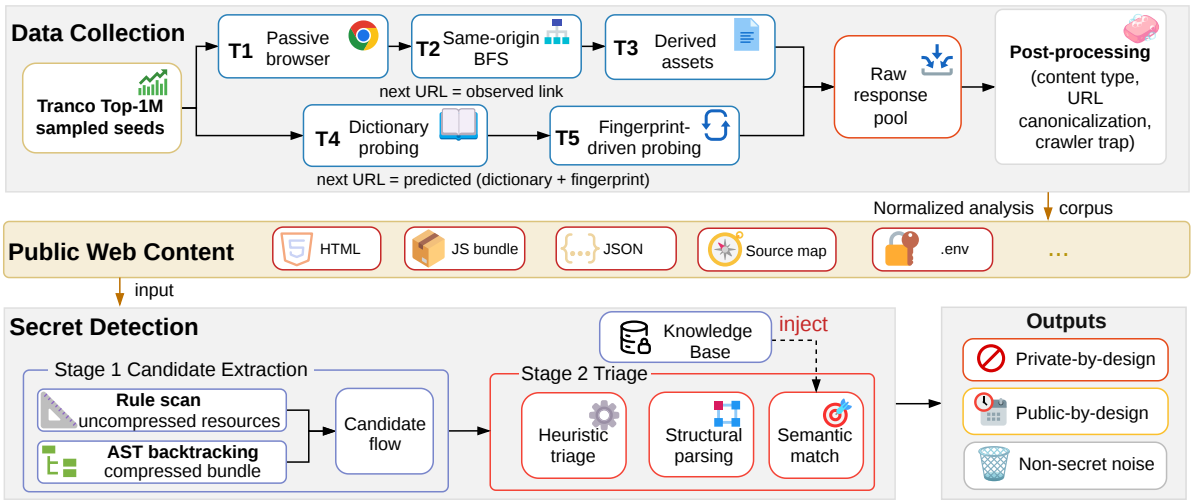


Figure 2: Overall architecture of KeySifter: top, the Data Collection module (§3); bottom, the Secret Detection module (§4).

3.2. Link-Driven Crawling

Link-driven crawling recovers the rendered surface, the assets a page eventually references once it has loaded and been exercised (Figure 2, upper half, T1–T3). A single landing fetch sees only the first layer of that surface; we reach the rest by chaining three hops outward, each starting from URLs the previous hop has already surfaced: passive browser capture (T1), same-origin Breadth-First Search (BFS) (T2), and derived-resource expansion (T3).

T1 Passive browser capture. A headless browser loads each landing page and exercises it enough to surface its lazy-loaded routes. The payoff is *runtime config*: configuration values fetched after page load through endpoints such as `/api/config`, including Firebase init payloads and internal endpoint maps, that a static fetch never observes. The text-shaped responses it records enter the scannable *carrier* set.

T2 Same-origin BFS. The landing page is only one route, and the routes beneath it often inline runtime config of their own. To reach them we run a bounded same-origin BFS over the rendered Document Object Model (DOM), seeded from sitemaps where present [54], [13], bounded to depth 3 and capped at 120 pages per site. This reaches the per-route config and bundles the landing fetch alone cannot.

T3 Derived-resource expansion. T1 and T2 both follow what a rendered page points to, so they never reach *derived assets*: auxiliary build outputs such as source maps and build manifests, deployed alongside bundles but referenced by no page. T3 follows the pointers that bundles, stylesheets, and manifests carry to these files, recovering the original identifier names and module structure that minified bundles otherwise erase.

3.3. Prior-Driven Crawling

Link-driven crawling, by construction, can only fetch what some rendered resource points to, so an asset with no incoming reference stays invisible to it however deep the crawl runs. Yet the pilot showed that many such assets are not random: framework conventions and public-path lists make their URLs predictable even when no page names them, from `/.env` to build manifests and stack-specific entrypoints. Prior-driven crawling closes the gap by predicting those URLs and fetching them directly, via two complementary strategies (Figure 2, upper half, T4–T5): a fixed dictionary swept against every site (T4), and probes tailored to a fingerprinted stack (T5).

T4 Dictionary probing. T4 issues GET requests against a fixed dictionary of public paths, identical for every domain. The paths play two reinforcing roles: some are leaks in themselves, such as a checked-in `/.env` or an exposed `/.git/config`, while others are manifests that name further build outputs to fetch. The first yields secrets directly; the second seeds more crawling.

T5 Fingerprint-driven probing. The remaining assets have a URL or response shape that depends on the site's stack rather than on any fixed path, so a uniform dictionary cannot reach them. T5 fingerprints the site first and then issues matched probes. The clearest case is a legacy-User-Agent replay: re-issuing the landing fetch as an older browser often returns an older bundle still carrying secrets that the current build has since rotated out. Those secrets remain live, yet a default browser crawl never requests the bundle that holds them.

3.4. Post-Processing

Collection across T1–T5 leaves a raw corpus that is redundant, noisy, and partly unreadable, so post-processing reduces it to a canonical scan input by removing three classes of artifact in turn. First, the unreadable: we keep only text-shaped responses (such as HTML and JavaScript) and drop binaries the detector cannot parse. Second, the redundant: we deduplicate so the same asset, reached under many cache-busting or tracking query-string variants, counts once. Third, the noise: we drop responses that succeed at the HTTP level but carry no real asset, chiefly the SPA catch-alls that return the landing shell for arbitrary paths. The detector runs on what remains.

TABLE 4: Persisted-corpus composition by file type.

File type	Share	Source tiers
JavaScript (`.js`, `.mjs`, `.ts`)	35.3%	T1, T3, T5
HTML documents	16.4%	T1, T2
SVG / vector assets	15.3%	T1, T2
CSS	13.0%	T1, T2
JSON / config / dotfiles	6.0%	T1, T4
Source maps (`.map`)	5.4%	T3
XML / sitemaps	3.8%	T2
Plain text / others	4.8%	T1–T4

Re-running the full crawl against the pilot ground truth recovers a per-project median of 85.2% of analyzable assets and 89.7% of analyzable JavaScript, with the residual falling entirely in the out-of-scope categories above. Because the gap is accounted for, the per-strategy yields we report in the wild are a lower bound on exposure.

3.5. Dataset Scope and Crawl Coverage

We run the procedure in the wild over 65,000 domains randomly sampled from the Tranco top 1M [38][2], each crawled under one fixed anonymous configuration: a desktop Chrome 122.0 User-Agent for the default fetch, and a legacy Internet Explorer 11 (`Trident/7.0`) string for the T5 legacy-bundle replay. The crawl persists 8,393,191 files (644 GB; hereafter the *persisted corpus*).

The resulting corpus is dominated by assets reached through rendered pages: link-driven collection supplies 93.0% of carriers, including the routine HTML, JavaScript, and CSS that a browser loads. Prior-driven probing contributes the remaining 7.0%, but reaches unreferenced source maps and configuration files where secrets occur disproportionately often (Table 4). Every sampled domain receives the same universal T1–T4 pipeline, including bounded same-origin exploration, derived-resource expansion, and dictionary probing. T5 adds fingerprint-driven, framework-specific probes learned from the pilot families and is associated with only 5.13% of private-by-design values, whereas the universal T1–T4 union covers 94.87%. Framework-specific probes therefore affect only a small fraction of the crawl, while the main collection path applies to all domains.

4. Secret Detection and Risk Classification

Existing secret scanners largely assume source-code repositories where variable names and surrounding context survive intact [44], [49], [5]. The web ecosystem preserves neither, which raises two structural challenges.

- **C1. Minification erases the variable-name anchor.** Roughly 80% of JavaScript in our corpus ships bundled, reducing identifiers like `client_secret` to one- or two-character short labels. Generic rules then miss the secret even when its value is present verbatim.
- **C2. A browser-visible value cannot be judged by itself alone.** Browser-visible assets are saturated with public-by-design values, such as analytics keys and build hashes, that match real secrets in character class and length yet outnumber them by orders of magnitude.

2. Snapshot 2026-03-11, https://tranco-list.eu.

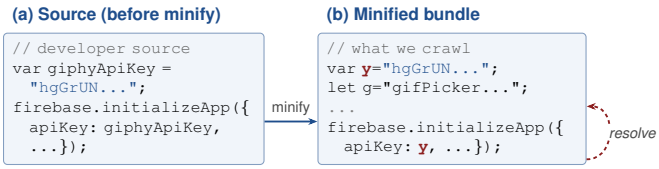


Figure 3: An example of AST call-site backtracking.

Our pipeline (Figure 2, lower half) addresses C1 and C2 in sequence. Stage 1 recovers candidates along two paths: a rule scan for prefix-anchored hits and an AST extractor for the identifiers bundling renames away (C1). Because both paths surface public-by-design lookalikes alongside genuine secrets, Stage 2 assigns labels directly when rule identity and value format are decisive, and uses retrieval against a risk-pattern knowledge base for the remaining candidates (C2).

4.1. Candidate Extraction

Candidate extraction runs two paths in parallel: a rule scan for plaintext and prefix-anchored hits, and an AST extractor for identifiers lost to minification.

4.1.1. Rule-Based Scan. The rule scan finds candidates from their value formats or nearby key names. We use KEYSENTINEL's broad rule set [73], which includes both service-specific and generic rules and provides broad credential-pattern coverage. We match every crawled artifact against two rule classes. *Service-specific rules* anchor on stable value prefixes, fixed lengths, and character sets (e.g., AWS `AKIA`, GitHub `ghp_`); *generic rules* anchor on key-name assignments such as `"api_key = ..."` or `"client_secret: ..."` [44], [49], [5]. Service-specific rules survive minification because the prefix lives inside the value. Generic rules do not: bundlers rename `client_secret` to a one- or two-character short alias and leave no semantic anchor for the next pass.

4.1.2. Minification-Resilient Extraction. To close the C1 gap left by generic rules, we recover credential-bearing identifiers directly from the AST. Rule scanning already covers candidates in inlined SSR JSON, source maps with intact `names[]`, and values identifiable by stable prefixes. AST backtracking recovers additional candidates from minified bundles whose original variable names are not available in the text.

AST Call-Site Backtracking. Bundlers minify *identifiers* but keep object property keys intact by default, so `{apiKey: y}` preserves the credential-bearing key even when `y` is renamed. We parse each bundle with `acorn` [1] and resolve every credential-keyed `Property` node back to its short-aliased declarator (Figure 3). The recovered name then enters the same downstream stages as plaintext-rule hits.

4.2. Heuristic Risk Triage

Stage 1 outputs a candidate set that mixes genuine secrets with deterministic noise and public-by-design lookalikes. Before invoking the costlier semantic match, we assign a label directly when the rule identity and value format are sufficient. KEYSENTINEL's built-in format checks filter generic-rule hits, and AST-recovered candidates with names unrelated to credential semantics are dropped. We skip context matching only when one clear signal determines the label: a high-risk value prefix or a server-side anchor identifies a *private-by-design secret*, a known front-end key family identifies a *public-by-design secret*, and an incidental credential-shaped string such as a bundle hash identifies *non-secret noise*.

The remaining candidates come from the *generic rules*, the catch-all patterns that match any high-entropy token in an API-key- or password-shaped slot regardless of service: values whose format is structurally identical across public and private credential families. A 32-character alphanumeric string could be a public Firebase SDK key or a private backend API key, and only surrounding context separates them. These ambiguous candidates then enter the context-based semantic matching stage.

4.3. Risk-Pattern Knowledge Base

Separating pattern construction from online inference keeps the semantic match deterministic and auditable: patterns are assembled once and only queried at detection time. We realize this separation as the *Risk-Pattern Knowledge Base* (RPKB), a 4,056-entry store held separate from the measurement set. Each entry pairs a credential family's structural fingerprint (prefix, length, character set, naming, usage), drawn from public service documentation, with context-side signal from front-end-identifier samples and false-positive lookalikes in our experimental dataset. Listing 1 gives the on-disk record. The construction details are provided in Appendix A.

4.4. Context-Aware Semantic Match

For the remaining generic-rule candidates, format alone is insufficient. We match each candidate's surrounding context against the RPKB and use the nearest entry to assign its risk label.

Query Construction. Static program analysis extracts each candidate's surrounding context (rule identity, raw value, asset path, code snippet, and lexical position) into a structured row. Where minification erases the variable name, we substitute the identifier recovered by AST call-site backtracking or source map back-recovery. We encode the row with Jina v4 [28] against a schema mirroring an RPKB entry. We L2-normalize the query and RPKB embeddings and retrieve the single nearest entry by cosine similarity above a fixed threshold $\tau = 0.80$; its pre-assigned `risk_level` field labels the candidate.

We adjudicate by retrieval rather than per-candidate LLM inference. LLMs have seen rapid uptake in security analysis, including credential detection on source code and mobile apps [7], [47], [2], [30], [45], but at the scale of web-ecosystem file analysis a generative judge stays unsuitable: measurement-grade detection demands reproducible, auditable verdicts free of fabrication, which it cannot guarantee. §5.4 reports the empirical comparison and the full rationale.

5. Classifier and Baseline Evaluation

We evaluate the classifier against two baselines: rule-based secret scanners (Gitleaks, TruffleHog) and a context-aware LLM judge (Claude Opus 4.7). Rule scanners are necessary candidate extractors but emit tens of thousands to millions of unclassified values; an LLM judge clears 90% precision but trails on recall and is cost-prohibitive at full-corpus scale. The classifier sits on top of rule extraction, not in place of it.

5.1. Evaluation Design

The full persisted corpus contains 8,393,191 files, making manual labeling infeasible. We therefore evaluate on a 20% sample drawn by simple random sampling without replacement (SRSWOR) [9], giving an evaluation set $\mathcal{E}$ with $|\mathcal{E}|$=1,678,644 files. So that no single tool dictates what gets reviewed, every method extracts candidates over $\mathcal{E}$ independently and we label the union of their outputs. The methods are KeySifter, Gitleaks v8, and TruffleHog v3. We evaluate TruffleHog in two configurations: TruffleHog-D (default) and TruffleHog-G (generic detector enabled)[3].

Ground Truth and Metrics. Two security researchers, each with over five years of experience, label the candidates following the same procedure and criteria as Appendix A (Cohen's $\kappa = 0.83$). This confirms 104 distinct private-by-design values, which serve as ground truth for precision and recall on $\mathcal{E}$. Recall is the fraction of these 104 values a method recovers anywhere in $\mathcal{E}$; precision is the fraction of a method's findings whose reported text matches a confirmed value. We evaluate the LLM judge on a separate balanced sample (§5.4).

Environment. All experiments run on an Ubuntu 22.04 server (dual Intel Xeon Gold 6230R, 52 physical cores; 512 GB RAM; 4×NVIDIA RTX A6000, 48 GB each).

5.2. Component Evaluation

The two-stage classifier reduces 401,266 raw `(file,value)` records to 171 private-by-design verdicts at 93.6% precision (Table 5). Without these classification steps, only one in every 3,266 raw rule hits contains a genuine high-risk value on average.

RPKB Ablation. Removing the RPKB semantic match while retaining the extracted candidate set yields 0.047% precision and 87.5% recall. With the RPKB, precision rises to 93.6%

3. The generic detector (`detectors.Generic`) is off by default and must be enabled with an explicit command-line flag [58].

TABLE 5: Per-method detection comparison on $\mathcal{E}$. Precision is record-level (matching records / Records); recall is value-level (recovered / ground-truth values).

Method	Records	Unique values	Recall	Precision
KeySifter	171	95	81.7%	93.6%
Gitleaks	189,180	58,400	68.3%	0.0973%
TruffleHog-D	63,353	42,544	24.0%	0.0931%
TruffleHog-G	6,662,703	1,800,949	32.7%	0.0015%

while recall falls to 81.7%, showing the precision–recall tradeoff introduced by context classification.

AST Contribution. We assess AST call-site backtracking (§4.1.2) separately on the full crawl corpus, using the 1,414,981 minified files among its 1,768,726 JavaScript files. Of the 114 unique private-by-design values recovered from these files, 23.7% are found only through AST backtracking and are missed by rule scanning.

Crawl-Tier Contribution. On the full crawl corpus, the additional crawl tiers find 63.25% of private-by-design secrets beyond landing-page-only scanning (Table 9).

Structured-Rule Stage (Heuristic Triage). This stage commits hits resolved by strong format or service semantics (e.g., private keys, AWS secrets, and GitHub/GitLab tokens) and reaches 99.6% precision on $\mathcal{E}$. The remaining false positives arise from overly permissive rule boundaries.

Vector-Similarity Stage (Semantic Match). This stage classifies the remaining generic-rule candidates by matching their context against the RPKB. In a separate experiment on $\mathcal{E}$, this stage achieves 84.2% precision.

5.3. Comparison with Tool-Level Secret Scanners

We run KeySifter against the two rule scanners on the same evaluation set $\mathcal{E}$, with TruffleHog in its default and generic-detector configurations.

Table 5 shows the headline gap on precision: 93.6% for KeySifter against under 0.1% for every rule scanner evaluated. Without a post-extraction adjudication step, the scanners emit between 42,544 (TruffleHog-D) and 1,800,949 (TruffleHog-G) unique values, overwhelmingly public-by-design tokens that match a secret format but carry no actual risk. On recall, KeySifter attains 81.7% on $\mathcal{E}$.

Many confirmed values lack stable format signatures, so rule scanners miss them. Enabling TruffleHog's generic detector raises recall from 24.0% to 32.7%, but increases candidate records from 63,353 to 6,662,703 (Table 5). Even Gitleaks, with the broadest ruleset of the three, recovers only 68.3%. AST call-site backtracking recovers additional candidates beyond these rule-only baselines.

Why Rule Extraction Is Necessary But Not Sufficient. A rule scanner equates "matches a credential format" with "is a leaked secret," yet format alone cannot separate a public front-end key from a private backend secret of identical shape. The two-stage classifier supplies the missing decision: rule extraction provides the candidates, and the classifier turns them into auditable private-by-design verdicts. Appendix B breaks down the individual Gitleaks and TruffleHog misses by rule coverage, field-extraction edges, length and exclude filters, and the output explosion under broadened rules.

TABLE 6: Results of LLM-based detection.

Setting	Recall	Precision	Cost
w/ location prior	37/50 (74.0%)	37/41 (90.2%)	$23.03
w/o location prior	34/50 (68.0%)	34/37 (91.9%)	$225.60

5.4. Comparison with LLM-Based Detection

A rule scanner cannot adjudicate a candidate, while a context-aware LLM can. Prior work uses LLMs for credential detection on source code and mobile apps [7], [47], [2]. We evaluate Claude Opus 4.7 [4] on 100 samples (50 private-by-design and 50 confirmed clean) under two settings. *With location prior*, the model receives a ±16 KB window [40], [15] centered on the candidate offset; the size keeps the candidate together with its declarations and call sites, sits within our web-asset file-size distribution, and stays below the regime where long-context utilization degrades. *Without location prior*, the model receives the full file. We design the prompt following the credential-triage conventions in the GitGuardian technical report [19]; Appendix C reproduces the full template. Table 6 reports recall, precision, and measured cost.

LLM precision exceeds 90% in both settings, but recall is the binding constraint: 74.0% with the location prior and 68.0% without. The errors point to two limits.

- **Public tokens look like secrets.** False positives concentrate on hard-coded analytics IDs, SDK-embedded tokens, and CAPTCHA site keys. The model reads hard-coded-value-plus-real-endpoint as a secret signal and overlooks that browser-visible tokens are designed for client use.
- **Signal dilutes without a prior.** On full-file input, the credential site is buried in surrounding code, and, because the true signal is outweighed by that noise, the model tends to label the file as containing only public front-end identifiers. The location prior recovers recall, but at full-corpus scale, that prior must come from an upstream detector.

Cost and Reproducibility. Extrapolating from the 100-sample audit, sending all 8,393,191 files into the model would cost roughly $4.25 million. Even after TruffleHog-G prefiltering and a ±16 KB window per candidate, full coverage still costs about $332,851, beyond the budget for reproducible empirical measurement.

6. Measurement

We apply KeySifter to the 65,000-domain dataset and report the findings as four questions that move from surface to consequence: where risky secrets surface (RQ1), which secret types dominate (RQ2), how much each crawl strategy contributes (RQ3), and how far an exposed secret reaches (RQ4). Three case studies then ground RQ4's repair-route taxonomy in independently confirmed disclosures.

TABLE 7: Secret-category distribution by (a, left part) file type and (b, right part) detection rule (log-scaled rate).

File type	Total hits	Priv	Priv rate	Priv share	Pub rate	Noise rate
JavaScript	969,129	393	0.04%	**43.3%**	2.5%	97.5%
HTML	752,461	349	0.05%	38.5%	19.1%	80.9%
JSON	46,297	81	0.18%	8.9%	10.6%	89.2%
CSS	7,214	0	0.00%	0.0%	16.8%	83.2%
source map	162,851	62	0.04%	6.8%	0.9%	**99.1%**
.env	89	7	**7.87%**	0.8%	23.6%	68.5%
meta_json	38,450	12	0.03%	1.3%	**31.0%**	68.9%
PHP	3,878	3	0.08%	0.3%	7.4%	92.5%
Others	16,815	0	0.00%	0.0%	13.0%	87.0%
Total	1,997,184	907	0.05%	100%	9.5%	90.5%

Detection rule	Total hits	Priv	Priv rate	Priv share	Pub rate	Noise rate
generic-api-key	1,789,602	548	0.03%	**72.4%**	5.4%	94.6%
AWS API Secret	88	86	97.7%	11.4%	0.0%	2.3%
Private Key	23	23	**100%**	3.0%	0.0%	0.0%
generic-password	72,057	15	0.02%	2.0%	0.4%	**99.6%**
Ideogram-api-key	19	10	52.6%	1.3%	0.0%	47.4%
flickr-access-token	8	8	**100%**	1.1%	0.0%	0.0%
jwt	3,954	7	0.18%	0.9%	**77.3%**	22.5%
sentry-access-token	7	7	**100%**	0.9%	0.0%	0.0%
Onelogin OAuth	4	4	**100%**	0.5%	0.0%	0.0%
Other rules	7,068	49	0.69%	6.5%	8.8%	90.7%
Total	1,872,830	757	0.04%	100%	5.4%	94.6%

Note. Priv rate, Pub rate, and Noise rate are each row's category count divided by Total hits; Priv share is the row's Priv count divided by the panel's total Priv count. Within each shaded column, darker shading indicates a larger percentage; colors distinguish private, public, and noise.

TABLE 8: Result overview of the real-world measurement.

Type	Raw records	Dedup. occ.	Uniq. values	Domains
Private-by-design secret	941	757	234	192
Public-by-design secret	112,012	100,283	17,033	6,910
Non-secret noise	2,042,848	1,771,790	291,402	15,013

6.1. Result Overview

Post-processing (§3.4) reduces the persisted corpus to 1,997,184 text-shaped files (hereafter the *analyzable corpus*); KeySifter classifies the raw candidate records it extracts from this set into the three kinds of §2.3. Table 8 gives the breakdown by deduplicated occurrence, unique value, and affected domain. Two properties of this distribution shape the rest of the section.

First, a genuine confidentiality break is rare. Private-by-design secrets are a fraction of a percent of all records, and the overwhelming noise rate is the inherent cost of telling real credentials apart from the placeholders, build hashes, and sample values that share their surface form. Detection therefore cannot stop at pattern matching, and the rest of this section reasons about the few hundred values that survive classification rather than the millions of raw matches.

Second, the real secrets do not stay put. The 234 unique private-by-design values recur 757 times within the sites that expose them, because one secret is typically duplicated across a site's rendered pages, bundles, and configuration files. A single exposed value is thus rarely a single point of exposure, and the blast-radius analysis (RQ1–RQ4) quantifies how far that replication reaches.

6.2. RQ1: Where Do Risky Secrets Surface?

This section analyzes how secrets distribute across response file types. Table 7(a) breaks the deduplicated occurrences down by file type along two axes: where most secrets are, and where the per-hit odds of a secret are highest.

Most secrets surface in the assets a browser already loads. HTML, JavaScript, and JSON together carry 90.74% of the 907 file-type occurrences, simply because these are the files every site serves in volume and the ones into which server-side rendering and client configuration routinely inline credentials. A scanner restricted to dotfiles and templates, as public source-code scanners often are [23], misses this majority outright.

The highest per-hit risk lives on surfaces a passive browser never fetches. Most file types dilute a few real secrets among megabytes of benign code, but a handful of niche surfaces invert that ratio and concentrate them. Genuine `.env` responses are the clearest case: they are credential files by intent, and their private-by-design rate runs an order of magnitude above any other file type (Table 7(a)). What such high-yield surfaces share is that a browser never requests them on its own, which is precisely what makes reaching them worthwhile (RQ3).

6.3. RQ2: Which Secret Types Dominate?

Having located the exposure, we ask what kind of secret produces it, since capability is what turns a leaked string into a usable secret. The hits span nearly every capability class, from cloud credentials and private keys to OAuth secrets and browser-visible API keys, yet only 0.040% resolve to a private-by-design secret (Table 7(b)). The detection rules behind them fall into two regimes that demand opposite treatment, and the gap between the two is exactly what the context classifier exists to close.

Tightly patterned rules need almost no adjudication. When a rule's value pattern admits few non-secret strings, as for private keys and OAuth client secrets, a match is almost always a real secret, so the rule's evidence alone is enough to confirm it. These rules are precise but narrow: each covers one credential format and together they account for only a small share of the secrets we find.

Broad rules supply the volume but none of the precision. The generic family, by contrast, matches across capability classes and so cannot judge a hit by its shape alone. `generic-api-key` is the extreme case and the dominant one: it accounts for 72.39% of all private-by-design secrets, yet fewer than one in three thousand of its hits is real, the rest being the public keys and noise that share the same loose shape. Recovering that majority of true secrets from the surrounding noise is work that pattern matching alone cannot do, and is exactly what the classifier's context stage contributes.

TABLE 9: Crawl-strategy coverage of the private-by-design secrets.

	T1 Land.†	**T2**	**T3**	**T4**	**T5**	Resid.
Coverage	36.75%	35.90%	28.63%	8.12%	5.13%	6.41%

†Additionally adds lazy-load scrolling and click actions over prior work's [14] passive HAR baseline. Total coverages exceed 100% because T1–T5 partially overlap.

6.4. RQ3: What Does Each Crawl Tier Contribute?

Knowing which surfaces and types carry risk, we ask how much of that risk a simpler crawl would have missed. We attribute the 234 unique private-by-design values to the T1–T5 steps and report any values not assigned to a single step as residual. Per-strategy coverage appears in Table 9.

Landing-page scanning recovers only a third of the secrets; two-thirds require deeper strategies. Just over a third of the values are reachable from the landing page and the resources it directly triggers (Table 9). The remaining 63.25% live in subpage routes, build artifacts, source maps, legacy bundles, error pages, or paths only an active probe reveals. Our T1 baseline already extends the passive HAR-style landing capture of prior work [14] with lazy-load scrolling and navigation clicks, yet it stays in the same order of magnitude as that work, confirming that the gap is not an artifact of a weak baseline. Closing the other two-thirds is what the deeper strategies are for.

No single strategy is redundant, because each reaches an entry class the others cannot observe. The five strategies are not a ranked list in which the deepest subsumes the rest. Their coverages overlap on the same 234-value denominator rather than summing, because they often reach the same secret by different routes, yet each also reaches a class of asset the others never see: a subpage route that the landing page never links, a build artifact behind a source map, a credential file that responds only to a probed path. Drop any one strategy and a distinct slice of secrets goes dark, which is why the gap beyond T1 cannot be closed by simply deepening a single technique.

Shadow assets are a non-negligible leakage source. To gauge how much exposure sits outside the reference graph, we classify each secret-bearing file by whether any rendered page reaches it. We find that 18.3% of the leaks come from *shadow assets*, files that no rendered page references and that only the prior-driven probes of T4–T5 recover. Nearly one secret-bearing file in five is therefore invisible to any crawler that follows links alone.

6.5. RQ4: How Large Is the Blast Radius?

A count of distinct exposed secrets understates attacker leverage. We measure how far each exposed value reaches along four axes: cross-domain reuse (D1), intra-site fanout (D2), Tranco visibility (D3), and owner-exposer separation (D4).

D1. Cross-domain reuse. Cross-domain reuse affects only a minority of values (6.41%), but where it occurs the "one site, one secret" assumption breaks: the broadest single value spans 16 domains (Table 10). A value reused across unrelated sites is almost always pushed down from a shared upstream,

TABLE 10: Cross-domain reuse and blast radius. CD: Cross-Domain, MD: Max Domain, VD: Value-Domain.

Type	Uniq. values	CD values	Share	MDs	VD pairs	Edges total
Private-by-design secret	234	15	6.41%	16	73	113
Public-by-design secret	17,033	908	5.33%	321	4,903	20,126

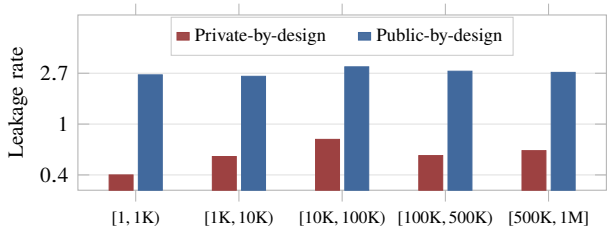


Figure 4: Per-domain leakage rate (%, log scale) by Tranco rank.

whether a copied configuration, an embedded SDK, or a platform-issued credential, so the fix has to route to that upstream rather than to any one exposing site. D4 develops these routes.

D2. Intra-site fanout. Rotating a secret is complete only once every file that carries it has been found, so the operative unit is not the 234 unique values but the 757 occurrences they spread into within their own sites. That spread is heavily skewed: a handful of worst-affected domains hold most of the copies (Appendix D), and on those sites a single missed file silently re-exposes a freshly rotated secret.

D3. Tranco visibility. Per-domain leakage rate is non-monotonic with site prominence: for both kinds of secret it peaks in the middle of the ranking rather than at the head or tail (Figure 4).

The middle tier leaks most often; the head tier leaks less but accumulates the largest surface. The lower rate at the very top is consistent with stronger secret governance at large sites, whereas the mid-rank peak may reflect complex front-end stacks and third-party integrations whose governance has not caught up with that complexity. Prominence still compounds the consequence rather than the frequency: leaks on top-ranked domains carry well above the global average number of copies each, since larger user bases and deeper supply chains widen the reach of every exposed value.

D4. Separation of owner and exposer. The party that exposes a secret is often not the party that can rotate it, which means fixing a leaking site can be the wrong action entirely. Three patterns explain D1's cross-domain reuse and determine where the repair must route:

- *L1 passive reuse:* multiple sites copy the same configuration or template, and each operator can rotate its own copy independently.
- *L2 SDK and plugin propagation:* downstream sites load credentials packaged by an upstream component, so only the upstream provider can rotate the shared value.
- *L3 platform-level distribution:* a customer site exposes a platform-issued secret it has no authority to rotate, so remediation rests with the cloud project owner or SaaS platform.

In L2 and L3, a disclosure addressed only to the exposing site will not result in rotation.

Reuse radius and service capability compound. The reach

```
#Case I: SDK config(unauth GET .../settings)
{ "chains":[ { "name":"evm", "networks":[ {
    "chainName":"Ethereum Mainnet", "networkId":1,
    "rpcUrl":"https://eth-mainnet.g.alchemy.com/v2/
     <API-KEY>"
} ] } ], "kyc":[...], "general":{ "displayName":"..." }}
```

```
#Case II: Ghost CMS(unauth GET .../configuration/)
{ "configuration":[ {
    "blogUrl":"https://<press-subsite>/",
    "clientId":"ghost-frontend",
    "clientSecret":"<OAUTH-SECRET>",
    "fileStorage":true, "useGravatar":true } ] }
```

```
#Case III: bundle 2822.chunk.js(hardcoded Maps key)
var s = document.createElement("script");
s.src = "https://maps.googleapis.com/maps/api/js"
      + "?key=AIzaSy...<redacted>&libraries=places&
     callback=init";
document.head.appendChild(s);
```

Figure 5: Three confirmed high-risk leaks (values redacted).

measured in D1 counts sites; capability is what each of those sites can do with the shared value, and the two multiply rather than add. A value reused across many domains is already hard to contain, and when that same value is also callable on high-value services (the capability dimension we quantify in §7), it becomes a single upstream point of failure that no downstream site can close on its own. This is why fix routing, not detection alone, decides whether a leak is actually repaired.

6.6. Case Studies

We present three typical leakage cases (Figure 5), independently verified via HackerOne, Intigriti, and vendors, profiled along family, mechanism, impact, and responsible party.

Case I: upstream SDK propagation, one secret inherited by every embedding site (L2). *Family*: third-party SDK upstream credential. A widely used crypto-wallet SDK ships hard-coded node-access keys, the API keys an app uses to read from and write to a blockchain, for several chains at once, so every website that embeds the SDK inherits the same paid blockchain access. *Mechanism*: on initialization, the SDK fetches an unauthenticated settings endpoint whose per-chain configuration embeds each node URL with its API key in plaintext, with no Referer, Origin, or IP restriction. *Impact*: anonymous callers issue blockchain requests against the provider's paid quota, exhausting it and shifting the cost upstream; the same exposure is shared across every embedding site. *Party*: the SDK provider, since notifying any individual embedding site would not result in rotation.

Case II: site-owned exposure on public config paths (L1). *Family*: site-owned content credentials. A Tranco top-3K travel platform hosts a press subsite on Ghost [18], an open-source content management system (CMS). *Mechanism*: two OAuth-style `client_secret` values are returned by unauthenticated GET requests to the publicly reachable paths `/configuration/` and `/clients/slug/ghost-frontend`. *Impact*: paired with the Ghost Content API, the first secret exposes pre-publication multi-language drafts and the second enumerates 88 CMS staff accounts. *Party*: the site operator, since both rotation and endpoint restriction are within its control.

Case III: public-by-design misconfiguration, fanned out across storefronts (L3). *Family*: public-service key with a missing restriction. A Tranco top-200 consumer-electronics vendor ships the same Google Maps API key inside front-end JavaScript chunks across roughly 28 regional storefronts. *Mechanism*: the key is browser-visible by design, but the Google Cloud Platform (GCP) project carries no Referer or IP restriction. *Impact*: anonymous callers invoke the project's enabled, billable Geocoding and Places endpoints; sustained abuse exhausts the quota, shifts costs to the vendor, and can trigger provider-side throttling that degrades map features across all storefronts. *Party*: the cloud project owner; the fix is a Referer or IP restriction in the GCP console, not a change to the web front-end.

TABLE 11: Public-by-design secret verdicts and service breadth.

Per-key verdict (n = 2,684)			**Service breadth** (n = 1,010)		
Bucket	# Key	%	# Services per key	# Key	%
Directly usable	1,010	37.60	1	317	31.39
high-value	137	5.10	2	297	29.41
Conditional	600	22.40	3	49	4.85
Referer-only	475	17.74	4	49	4.85
IP-restricted	125	4.66	5	27	2.67
Unavailable	1,074	40.00	6–9	135	13.37
Service disabled	784	29.21	≥10	136	13.47
Key invalid	290	10.79	*mean/med/max*	*3.70/2/16*	

7. Service-Level Secret Validation

To further study the security risks of public-by-design secrets, following prior work [14], we select 31 Google API services for targeted validation and analyze their keys' exposure lifetime.

7.1. Service-Level Validation of Google API Keys

Probing Protocol. We test each of 2,684 deduplicated Google API keys against 31 documented Google service endpoints (Appendix E) spanning Maps/Geo, media, AI, and identity services. For each key we record a per-service outcome and collapse it to one of three verdicts ordered by how freely the key can be called: *directly usable*, *conditional*, or *unavailable*. Table 11 states the condition that defines each bucket. All probes follow the ethical constraints in Appendix F.

Verdict Distribution. The keys split cleanly rather than clustering in the conditional middle: 1,010 (37.60%) are directly usable, and the remainder are either fully unavailable or gated behind an origin or IP condition (Table 11, left). These directly usable keys are rarely confined to one service: the average such key reaches 3.7 services (Table 11, right). What governs risk, however, is not this headcount but which services a directly usable key can reach. Most reach only Maps/Geo, but a high-value tail of 137 keys reaches endpoints such as Gemini and Vision (Figure 6, Appendix E), where the consequence shifts from quota abuse to data extraction and AI-capability access.

TABLE 12: Re-checked results of exposed secrets.

Transition	# Key	Mechanism
Tightened or revoked (38)		
Maps-only → Conditional (Referer)	14	origin restriction added
High-value → Conditional (Referer)	8	origin restriction added
Conditional → Invalid	7	key revoked / expired
Maps-only → Invalid	4	key revoked / expired
High-value → Maps-only	3	high-value API disabled
Conditional → Disabled	1	service disabled
Disabled → Invalid	1	key revoked / expired
Relaxed (5)		
Conditional → Maps-only	4	restriction lifted / re-enabled
Conditional → High-value	1	high-value API enabled

7.2. Key-Level Verdicts and Capabilities

Service-level profiles are not evenly distributed across keys, and aggregating them surfaces two patterns a binary view hides: an exposed key is usually callable on several services at once, and the set of services it reaches can grow over time without any new leak.

Multi-service exposure is the norm. A single key rarely maps to a single service. The 1,010 directly usable keys span 3,736 key-service edges, and more than two thirds reach two or more services (Table 11, right). A key-level valid/invalid label collapses this surface, because the same key can be restricted on one service yet directly usable on another within the same GCP project.

A leaked key's attack surface grows as its GCP project expands. As Google API keys carry no API restriction by default [27], [26], [60], a key minted today for an embeddable Maps or Firebase front-end silently inherits access to many APIs its GCP project later enables, including Gemini, often without prompting the developer to revisit the key. We see this convergence in the data: 57 directly usable keys already carry both a Maps/Geo and a high-value AI or media capability, so the boundary between a low-stakes utility key and a high-value one is set by project configuration the developer may never revisit.

7.3. Secret Exposure Lifetime

To study the lifecycle of exposed keys, we re-checked the same set of keys three weeks after disclosure (see Appendix F).

Most exposed keys remain uncorrected. Exposure is overwhelmingly static over the window: 98.4% of keys hold their category, and only 1.4% move toward any repair (Table 12). The directly usable keys stay publicly callable on the same services, which suggests developer awareness of browser-visible key exposure remains low. Even the rare repairs are partial, usually adding an origin restriction rather than revoking the key, reducing the exposure without closing it.

Exposure can deepen without a new leak. A browser-visible value is fixed once published; what keeps changing is the GCP project behind it, and that configuration can widen the exposure on its own. Over the same window, 5 keys became more capable across 7 key-service pairs even though the public value never changed [60], as the project re-enabled a service, lifted a restriction, or revived an apparently dead key. One key crossed from conditional to high-value when its project enabled the YouTube Data endpoint. Browser-visible exposure is therefore durable and only weakly self-correcting, and an attacker who keeps a public key on hand may find it more useful later than at the moment of leak.

8. Discussion

8.1. Root Cause Analysis

While the mechanisms of secret leakage vary significantly, their root causes generally fall into the following categories.

(1) Framework mechanism and developer error: the client boundary, misjudged or ignored. One root cause, a developer who misjudges where the client boundary lies, surfaces in two ways. The first is mediated by framework defaults: bundlers expand `process.env`, server-side renderers serialize state into hydration HTML, and public-prefix variables (`NEXT_PUBLIC_`, `VITE_`, `REACT_APP_`) ship anything matching the prefix into the client bundle. These browser-visible everyday assets account for 90.74% of all 907 file-type occurrences in our data; a secret hard-coded under such a prefix leaks all the same, with at least 12 server-side secrets so marked. The second is direct misconfiguration: a private value placed on a publicly reachable path, whether a document root left open to dotfiles such as `.env`, a credential file under the public directory, or a private key hard-coded as a front-end constant.

(2) Upstream SDKs and platforms: one source, many inheritors. An upstream SDK, plugin, or platform can ship one secret to many downstream sites, by serializing build-time secrets into a published bundle, returning platform-level secrets in unified API responses, or embedding long-lived multi-tenant tokens in client components. The downstream site is the *exposer*, but only the upstream *owner* can rotate or restrict the value. Cross-domain reuse is a diagnostic signal: 15 of 234 private-by-design values (6.41%) span more than one registrable domain, and public-by-design values reach far wider, the broadest surfacing on 321 domains. Reuse at this scale is consistent with supply-chain propagation, though we cannot rule out independent copying.

(3) Neglect of public-by-design secrets: visible is not unrestricted. Some secrets are designed to appear in the browser (Google Maps keys and Mapbox public tokens, for example), and developers hard-code them into visible resources on purpose, following the provider's own documentation. Putting the value in the page is not the mistake. The mistake is what comes next: the value's security model rests not on confidentiality but on safeguards such as Referer/IP restrictions, API allowlists, quota policies, and console-side rotation, and these are routinely left unset after issuance. Absent them, the model reduces to value-only protection, which is no protection at all: anyone who reads the key off the page can call the service on the owner's quota. Because restrictions are scoped per service, the unit of risk is the key-service pair, not the key alone; our validation finds a

single key restricted on one service yet freely callable on another in the same project.

8.2. Mitigation

We propose good practices for each cause, taking the three in order.

(1) Framework and developer. For the framework path, frameworks should separate server-only from client-side configuration by default, warn when a server-looking secret is marked with a public prefix, and scan delivered assets, not just source, before release. For the direct path, deny dotfiles such as `.env` at the server and monitor for sensitive files that become publicly reachable, anchor the document root to the sanctioned public directory, and inject server-side secrets through a secret manager [3], [29], never the browser.

(2) Upstream owner. Owners should default to short-lived, per-tenant, minimum-scope secrets and replace shared secrets with presigned URLs or session tokens. Once cross-domain reuse is observed, disclosure should reach the upstream owner first under multi-stakeholder notification [62], since the exposing site cannot rotate the value.

(3) Key provider and consumer. "Restricted at creation" should be the provider default, and the console should surface, per key, which services it can call and under what restrictions. Consumers should treat public-by-design keys as policy-bound capability secrets, and detection tools should report a service-level capability profile rather than a single valid/invalid verdict.

8.3. Limitations

Although our method improves over previous work, several limitations still remain.

Crawl Coverage and Sampling Bias. Two limits bound what we observe. The first is crawl depth: although T1–T5 reach as many assets as possible, our link-depth and dictionary limits keep us from exhausting every page or path. The second is sampling: our results come from a single top 1M snapshot of 65,000 randomly sampled domains. That sample spans the full Tranco rank range yet still cannot fully represent the web at large. Under both limits, our results are best read as a lower bound on secret exposure across the web ecosystem.

Detection Coverage. Although KeySifter reaches 93.6% precision at 81.7% recall, some values are still missed, for two reasons. First, the filtering rules that suppress noise are deliberately strict, so a minority of genuine secrets are dropped along with the noise. Second, content that ships obfuscated may stay out of reach, though this edge is largely mitigated where a deobfuscation tool can recover the original structure [73].

Validation Scope. Ethical considerations bound what we validate. Private-by-design secrets that must never leak, many of them tied to per-user access control, cannot be probed directly. Beyond that, many secrets are designed for internal use, reachable only through specific non-public endpoints or under particular conditions, so devising a single automated method to validate them remains a challenge.

9. Related Work

Secret Leakage Detection. A large body of work studies secret leakage at the source-code level: early scanners relied on service-specific formats [53], large-scale measurements showed such leaks are pervasive on public GitHub [44], [68], [50], and later detectors target cases missed by format matching through deep learning [17], machine-learning filters [49], [42], [35], and language models for repositories and Android apps [7], [47], [2], [30], [45]. Basak et al. find that many industrial secret-detection tools still fall short [6], [5], [48]; developer studies explain why leaks happen and why warnings are bypassed [36], [34]. Detection has also expanded beyond repositories to mobile apps [75], [51], [69], [52], [67], mini-program front ends [72], container images [12], IPFS files [70], and VSCode extensions [41]. Each still exposes a bounded artifact or source-level context from which a detector can recover intent, and secret leakage across the browser-visible web remains largely unexamined. Browser-visible assets are different: leaked server-side secrets and public-by-design client keys can share the same surface form, and industry reports on Google API keys show that a key's risk can change once a project enables Gemini capabilities [60], [8].

Web Measurement. Crawling browser-visible assets at scale draws on prior web-measurement work, from research-oriented site rankings that define crawl targets [38] to high-fidelity browser-based crawling infrastructure [66], [25] and studies of how representative automated crawls are of real browsing [71]. Large-scale web studies provide the methodological base, including million-site browser instrumentation for tracking measurement [16], longitudinal measurements of insecure and outdated website software [13], and analysis of JavaScript bundling as a core feature of modern front-end delivery [46]. For web secret leakage measurement, Demir et al. [14] analyze HTTP Archive captures and study persistence and disclosure, focusing on landing pages and missing many assets and secrets that surface only at depth. Instead, KeySifter crawls each site to a bounded depth and probes framework-generated asset URLs to recover far more assets, and builds a context-aware knowledge base to reduce the resulting noise.

10. Conclusion

This paper studies browser-visible secret leakage and detection as a web-delivery problem. KeySifter combines framework-aware asset collection, minification-resilient extraction, and context-aware risk classification, achieving 93.6% precision and 81.7% recall on our evaluation set, where source-code-oriented rule scanners remain below 0.1% precision. Applied to 65,000 Tranco-sampled domains and 8,393,191 files, KeySifter identifies 1,244 secret leakage issues across 1,437 domains. The results show that risk comes from both hidden web assets and misconfigured public-by-design secrets: across 31 documented services, 37.60% of exposed keys remain directly usable on at least one service. Responsible disclosure has led to 53 confirmed

or remediated cases. Our root-cause analysis further shows that durable mitigation requires coordinated controls across the web delivery chain, from build-time exposure to service-side capability enforcement.

References

[1] Acorn contributors, "acorn: A small, fast, JavaScript-based JavaScript parser," https://github.com/acornjs/acorn, 2025, accessed: 2026-06-07.

[2] M. Alecci, J. Samhi, T. F. Bissyande, and J. Klein, "Evaluating large language models in detecting secrets in Android apps," https://arxiv.org/abs/2510.18601, 2025.

[3] Amazon Web Services, "Amazon AWS secrets manager," https://aws.amazon.com/secrets-manager/, accessed: 2024-10-29.

[4] Anthropic, "Claude opus 4.7 model card," https://www.anthropic.com/claude/opus, 2025, accessed: 2026-05-22.

[5] S. K. Basak, J. Cox, B. Reaves, and L. Williams, "A comparative study of software secrets reporting by secret detection tools," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2023.

[6] S. K. Basak, L. Neil, B. Reaves, and L. Williams, "Secretbench: A dataset of software secrets," in *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[7] C. Biringa and G. Kul, "Detecting hard-coded credentials in software repositories via LLMs," https://arxiv.org/abs/2506.13090, 2025.

[8] CloudSEK, "Hardcoded Google API keys in top Android apps now expose Gemini AI," https://www.cloudsek.com/blog/hardcoded-google-api-keys-in-top-android-apps-now-expose-gemini-ai, Apr. 2026, accessed: 2026-05-23.

[9] W. G. Cochran, *Sampling Techniques*, 3rd ed. John Wiley & Sons, 1977.

[10] Codecov, "Bash uploader security update," https://about.codecov.io/security-update/, 2021, accessed: 2026-05-31.

[11] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[12] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets revealed in container images: An internet-wide study on occurrence and impact," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2023.

[13] N. Demir, T. Urban, K. Wittek, and N. Pohlmann, "Our (in)secure web: Understanding update behavior of websites and its impact on security," in *Passive and Active Measurement Conference (PAM)*, 2021.

[14] N. Demir, Y. Vekaria, G. Smaragdakis, and Z. Durumeric, "Keys on Doormats: Exposed API Credentials on the Web," https://arxiv.org/abs/2603.12498, 2026.

[15] Y. Du, M. Tian, S. Ronanki, S. Rongali, S. Bodapati, A. Galstyan, A. Wells, R. Schwartz, E. A. Huerta, and H. Peng, "Context length alone hurts LLM performance despite perfect retrieval," in *Findings of the Association for Computational Linguistics: EMNLP*, 2025.

[16] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[17] R. Feng, Z. Yan, S. Peng, and Y. Zhang, "Automated detection of password leakage from public GitHub repositories," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[18] Ghost Foundation, "Ghost: The professional publishing platform," https://ghost.org, 2026, accessed: 2026-06-08.

[19] GitGuardian, "The State of Secrets Sprawl Report 2024," https://www.gitguardian.com/state-of-secrets-sprawl-report-2024, 2024, accessed: 2024-10-14.

[20] GitGuardian, "The state of secrets sprawl 2025," https://www.gitguardian.com/state-of-secrets-sprawl-report-2025, 2025, accessed: 2026-05-31.

[21] Gitleaks contributors, "Gitleaks: generic entropy rule," https://github.com/gitleaks/gitleaks/blob/48a2e0e3184d8d07275a22f24d3c96019df8cd4c/cmd/generate/config/rules/generic.go#L39, accessed: 2024-11-11.

[22] Gitleaks contributors, "Gitleaks: Protect and discover secrets," https://github.com/gitleaks/gitleaks, accessed: 2024-10-05.

[23] Gitleaks contributors, "Gitleaks README: files skipped," https://github.com/gitleaks/gitleaks/blob/5c2e8132f2092ba4a48e0886c38ae9828f26fb51/README.md?plain=1#L161, accessed: 2024-10-29.

[24] Gitleaks contributors, "Gitleaks rule: generic-api-key," https://github.com/gitleaks/gitleaks/blob/master/config/gitleaks.toml, accessed: 2024-10-15.

[25] A. Goel, J. Zhu, R. Netravali, and H. V. Madhyastha, "Sprinter: Speeding up high-fidelity crawling of the modern web," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024, pp. 893–906.

[26] Google, "Gemini API keys: secure unrestricted keys," https://ai.google.dev/gemini-api/docs/api-key#secure-unrestricted-keys, accessed: 2026-05-23.

[27] Google Cloud, "Authentication using API keys," https://docs.cloud.google.com/docs/authentication/api-keys, accessed: 2026-05-23.

[28] M. Günther, S. Sturua, M. K. Akram, I. Mohr, A. Ungureanu, B. Wang, S. Eslami, S. Martens, M. Werk, N. Wang, and H. Xiao, "jina-embeddings-v4: Universal embeddings for multimodal multilingual retrieval," https://arxiv.org/abs/2506.18902, 2025.

[29] HashiCorp, "Vault by HashiCorp," https://www.vaultproject.io, accessed: 2024-10-29.

[30] Y. Huang *et al.*, "Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.

[31] IBM, "Cost of a data breach report 2024," https://www.ibm.com/reports/data-breach, 2024, accessed: 2024-09-23.

[32] Internet Archive, "Heritrix: Internet archive's web crawler," https://github.com/internetarchive/heritrix3, accessed: 2026-05-26.

[33] E. Kenneally and D. Dittrich, "The menlo report: Ethical principles guiding information and communication technology research," *Available at SSRN 2445102*, 2012.

[34] J. H. Klemmer, A. Krause, Y. Acar, and S. Fahl, "Committed by accident: Studying prevention and remediation strategies against secret leakage in source code repositories," https://arxiv.org/abs/2211.06213, 2022.

[35] A. V. Konygin, A. V. Kopnin, I. P. Mezentsev, and A. A. Pankratov, "Using bigrams to detect leaked secrets in source code," in *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2023.

[36] A. Krause, J. H. Klemmer, N. Huaman, D. Wermke, Y. Acar, and S. Fahl, "Pushed by accident: A mixed-methods study on strategies of handling secret information in source code repositories," in *32nd USENIX Security Symposium*, 2023.

[37] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[38] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," in *Network and Distributed System Security Symposium (NDSS)*, 2019.

[39] F. Li and K. A. Roundy, "Who you gonna call? an empirical evaluation of website security.txt deployment," in *ACM Internet Measurement Conference (IMC)*, 2021.

[40] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[41] Y. Liu, C. Tantithamthavorn, and L. Li, "Protect your secrets: Understanding and measuring data exposure in VSCode extensions," https://arxiv.org/abs/2412.00707, 2024.

[42] S. Lounici, M. Rosa, C. M. Negri, S. Trabelsi, and M. Önen, "Optimizing leak detection in open-source platforms with machine learning techniques." in *ICISSP*, 2021, pp. 145–159.

[43] M. Maass, A. Stoever *et al.*, "Effective notification campaigns on the web: A matter of trust, framing, and support," in *30th USENIX Security Symposium*, 2021.

[44] M. Meli, M. R. McNiece, and B. Reaves, "How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories," in *NDSS*, 2019.

[45] L. Niu, S. Mirza *et al.*, "CodexLeaks: Privacy leaks from code generation language models in GitHub Copilot," in *32nd USENIX Security Symposium*, 2023.

[46] J. Rack and C.-A. Staicu, "Jack-in-the-box: An empirical study of JavaScript bundling on the web and its security implications," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[47] M. N. Rahman, S. Ahmed, Z. Wahab, S. M. Sohan, and R. Shahriyar, "Secret breach detection in source code with large language models," https://arxiv.org/abs/2504.18784, 2025.

[48] M. R. Rahman, N. Imtiaz, M.-A. Storey, and L. Williams, "Why secret detection tools are not enough: It's not just about false positives-an industrial case study," *Empirical Software Engineering*, vol. 27, no. 3, p. 59, 2022.

[49] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera, "Secrets in source code: Reducing false positives using machine learning," in *International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2020.

[50] E. Schiavone, L. Chiavetta, and M. Migliardi, "Tales from the git: Automating the detection of secrets on code and assessing developers' passwords choices," in *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2023.

[51] D. Schmidt, S. Schrittwieser, and E. Weippl, "Leaky apps: Large-scale analysis of secrets distributed in Android and iOS apps," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2025.

[52] S. Shi, X. Wang, K. Zeng, R. Yang, and W. C. Lau, "An empirical study on mobile payment credential leaks and their exploits," in *EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2021.

[53] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, "Detecting and mitigating secret-key leaks in source code repositories," in *IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, 2015.

[54] A. Stafeev and G. Pellegrino, "SoK: State of the krawlers – evaluating the effectiveness of crawling algorithms for web security measurements," in *33rd USENIX Security Symposium*, 2024.

[55] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow, "Towards more successful web vulnerability notifications," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[56] Symantec Threat Hunter Team, "Mobile supply chain: Hard-coded AWS credentials in popular apps," https://www.security.com/threat-intelligence/mobile-supply-chain-aws, 2022, accessed: 2026-05-31.

[57] TechCrunch, "Toyota customer data exposed after access key left public on GitHub for five years," https://techcrunch.com/2022/10/12/toyota-customer-email-addresses-exposed/, 2022, accessed: 2026-05-31.

[58] Truffle Security, "TruffleHog: Find and verify secrets," https://github.com/trufflesecurity/trufflehog, accessed: 2024-10-05.

[59] Truffle Security, "TruffleHog rule: generic-api-key," https://github.com/trufflesecurity/trufflehog/blob/34e443adcf1548e573a9d8f8496efb49b5d8a9c1/examples/generic.yml#L2, accessed: 2024-10-15.

[60] Truffle Security, "Google API keys weren't secrets. But then Gemini changed the rules," https://trufflesecurity.com/blog/google-api-keys-werent-secrets-but-then-gemini-changed-the-rules, Feb. 2026, accessed: 2026-05-23.

[61] U.S. Federal Trade Commission, "In the matter of Uber technologies, inc.: Revised complaint," https://www.ftc.gov/system/files/documents/cases/152_3054_c-4662_uber_technologies_revised_complaint.pdf, 2018, accessed: 2026-05-31.

[62] Y. Vekaria, R. Nithyanand, and Z. Shafiq, "Towards multi-stakeholder vulnerability notifications in the ad-tech supply chain," https://arxiv.org/abs/2406.06958, 2024.

[63] Verizon, "2024 data breach investigations report," https://www.verizon.com/business/resources/reports/dbir/, 2024, accessed: 2026-05-31.

[64] X. Wang, Y. Tan, Y. Du, Y. Zhang, and Z. Lin, "Credit karma: Understanding security implications of exposed cloud services through automated capability inference," in *32nd USENIX Security Symposium*, 2023.

[65] Wappalyzer, "Javascript frameworks," https://www.wappalyzer.com/technologies/javascript-frameworks/, 2026, accessed: 2026-09-21.

[66] Webrecorder, "Browsertrix Crawler: Browser-based, crawler-in-a-container for high-fidelity web archiving," https://github.com/webrecorder/browsertrix-crawler, accessed: 2026-05.

[67] L. Wei, H. Huang, S.-C. Cheung, and K. Li, "How far are app secrets from being stolen? a case study on Android," https://arxiv.org/abs/2501.07805, 2025.

[68] E. Wen, J. Wang, and J. Dietrich, "Secrethunter: A large-scale secret scanner for public git repositories," in *2022 IEEE 21st International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2022, pp. 123–130.

[69] H. Wen and Z. Lin, "An empirical study of SDK credential misuse in iOS apps," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2018.

[70] Z. Wu, B. Kondracki, N. Nikiforakis, and A. Balasubramanian, "Secrets are forever: Characterizing sensitive file leaks on IPFS," in *IFIP Networking Conference*, 2024.

[71] D. Zeber, S. Bird, C. Oliveira, W. Rudametkin, I. Segall, F. Wollsén, and M. Lopatka, "The representativeness of automated web crawls as a surrogate for human browsing," in *The Web Conference (WWW)*, 2020, pp. 167–178.

[72] Y. Zhang, Y. Yang, and Z. Lin, "Don't leak your keys: Understanding, measuring, and exploiting the AppSecret leaks in mini-programs," in *ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[73] J. Zhou, Z. Zhang, L. Ying, H. Chai, J. Cao, and H. Duan, "Hey, your secrets leaked! detecting and characterizing secret leakage in the wild," in *IEEE Symposium on Security and Privacy (S&P)*, 2025.

[74] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in Android apps," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[75] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

```
{
  "id":              "<uuid4>",
  "original_index": <int>,
  "vector":          [<2048 float32>],
  "text_content":    "Variable Name: ... .
                     Service: ... .
                     Semantic Signature: ... .
                     Features - Length: ..., Charset:
    ... .
                     Context Keywords: ..., ... .
                     Endpoint Patterns: ... .
                     Typical Variable Names: ..., ...",
  "payload": {
    "variable_name":      "...",
    "service":            "...",
    "length":             "...",
    "charset":            "...",
    "prefix":             [...],
    "context_keywords":   [...],
    "endpoint_patterns":  [...],
    "typical_var_names":  [...],
    "semantic_signature": "...",
    "original_full_json": { /* full record:
        risk_level, confidence, reason,
        filtering_rule, verification_strategy,
        kb_features, ... */ }
  }
}
```

Listing 1: Schema of one record in the risk-pattern knowledge base.

Appendix A. Details of Risk-Pattern Knowledge Base

The semantic matching stage in §4.4 adjudicates each candidate by retrieval against a pre-built RPKB snapshot. This appendix documents how that snapshot is built, stored, and isolated from the measurement data.

A.1. Structured Generation and Human Review

To mitigate category bias and align RPKB patterns with the Web ecosystem, we sampled 15,000 Tranco domains outside the 65,000-domain measurement set and expanded secret/noise patterns until no new features emerged. This process yielded 37,676 candidates before filtering and deduplication. Records cover private keys, database passwords, public front-end keys, AI/cloud/API credentials, and non-secret lookalikes such as hashes, checksums, and tracking IDs. Each accepted item becomes one record with structural and contextual features and a `filtering_rule` (Listing 1).

LLMs organize candidates, extract contextual features, and draft records for human review.

After we organize these keys together with their context and item-level features, two security researchers, each with over five years of experience, independently label each record as *accept*, *revise*, or *reject*, reaching Cohen's $\kappa = 0.95$ [11], [37]. Disagreements are resolved by discussion, or by a third reviewer when consensus fails. The accepted records form the 4,056-entry memory used by the semantic matching stage.

A.2. Memory-Bank Storage

After human review, each accepted JSON record is materialized as one row of the risk-pattern knowledge base, whose on-disk schema appears in Listing 1. Two layout choices matter beyond the field list. First, an entry's `text_content` is assembled under the same logic the query side uses, so an entry and a candidate are embedded by an identical recipe. Second, `payload` stores those same fields redundantly alongside the original record, so the retrieval logic, the private-by-design promotion guard, and the auditable provenance chain read fields directly without re-decoding the vector. The `original_index` field records each entry's position in the review pool, which supports resumable batch processing.

Each measurement round uses one fixed RPKB snapshot, keeping retrieval reproducible.

Appendix B. Structured Breakdown of Baseline Misses

§5.3 shows that both tools achieve precision below 0.1% on $\mathcal{E}$. This appendix examines why they also miss confirmed private-by-design values at their default settings.

B.1. Gitleaks: Miss Analysis on $\mathcal{E}$

Gitleaks misses 31.7% of confirmed values on $\mathcal{E}$. The misses concentrate in two areas:

- **Format-ambiguous generic space (dominant).** Plain alphanumeric tokens and opaque strings in `key=value` positions whose character class and entropy fall below the `generic-api-key` threshold [24], [21]. This is the same format-less space that holds most confirmed private-by-design values on $\mathcal{E}$.
- **Multi-line / PEM bodies.** Serialized `-----BEGIN PRIVATE KEY-----` blocks whose canonical form spans newlines; Gitleaks' single-line scan dispatch drops them.

B.2. TruffleHog: Miss Analysis on $\mathcal{E}$

- **TruffleHog-D.** The default mode misses 76.0% of confirmed values. Without the generic detector enabled, credentials outside a known service format have no matching rule and are skipped outright. Even within supported service families, some detectors do not fire on the in-page value form, leaving additional values unrecovered [59].
- **TruffleHog-G.** Enabling `detectors.Generic` raises recall only marginally, while precision collapses to 0.0015% (Table 5), a volume no human triage can process.

Shared failure mode. Format matching alone cannot adjudicate the format-less generic space, and widening rules to cover it trades recall for an output volume no human triage can process.

Appendix C. LLM Judge Prompt Template

The following template is the prompt sent to the LLM judge in §5.4, adapted from the GitGuardian technical report [19]. The {`filename`} and {`content`} placeholders

```
You are a security expert analyzing web
files for leaked secrets and API keys.

Analyze the following file content and
identify any hardcoded secrets, API keys,
tokens, or credentials that appear to be
real and sensitive (not placeholders, test
values, or public-facing tokens).

File: {filename}

```
{content}
```

Respond with JSON only (no markdown fences):
{
  "found_secrets": [
    {
      "value": "<the secret value>",
      "variable_name": "<variable name
                        if visible,
                        else null>",
      "service": "<service/provider name>",
      "confidence": "high|medium|low",
      "reason": "<why this is a real secret>"
    }
  ],
  "verdict": "contains_secrets|no_secrets",
  "summary": "<one sentence summary>"
}
```

Listing 2: Prompt template sent to the LLM judge (§5.4).

TABLE 13: Intra-site fanout for the top 10 most-affected domains (anonymized).

Site	Occurrences	Values	Files	Fanout pattern
Site A	53	11	21	Mixed reuse
Site B	49	40	1	One file, many values
Site C	41	1	41	One value, many files
Site D	37	1	37	One value, many files
Site E	30	1	30	One value, many files
Site F	30	1	30	One value, many files
Site G	28	6	1	One file, many values
Site H	25	3	9	Mixed reuse
Site I	18	2	6	Mixed reuse
Site J	17	1	17	One value, many files
Top-10 total	328	–	–	43.33% of all occurrences

are substituted at runtime with the target file name and its textual content (the ±16 KB window centered on the candidate offset in the *with location prior* setting, or the full file otherwise).

Appendix D. Intra-Site Fanout: Full Per-Domain Breakdown

This appendix expands the per-domain fanout for the top 10 private-by-design domains referenced in D2 (§6.5). Table 13 reports per-domain occurrences, unique values, and files, identified only by anonymized site IDs.

Each fanout pattern carries a different risk profile.

- *One value, many files.* The same secret is duplicated across route pages, SSR output, or repeated bundle artifacts. Exposure is observable from many entry points, and remediation tends to miss legacy pages, stale chunks, or source maps.
- *One file, many values.* A single dense configuration file or bundle artifact carries multiple distinct secrets, so one exposure breaches several service boundaries at once.
- *Mixed reuse.* Both the secret values and their propagation paths must be cleaned up together.

Appendix E. Google Service Endpoints and Capability Distribution

The service-level validation in §7.1 probes a fixed set of 31 Google service endpoints. Table 14 lists them in full so reviewers can audit the measurement scope and the way we attribute each endpoint to a service family.

Fifteen of the 31 endpoints belong to Maps Platform, split across three response shapes: seven use the legacy Maps web service shape, two are image APIs, and six use standard JSON APIs. The three shapes differ only at the parsing layer, and all fifteen draw on one Google Maps Platform billing quota under one service-family label.

Each probe is a single read-only request that retrieves or analyzes data without creating or modifying any account-side state. We fill each request with example values from the vendor's public documentation, so it writes nothing, causes no side effects, and stays within the free or lowest billing tier Google documents. §8.3 sets out the ethical boundary on third-party accounts, billing quotas, rate limiting, and single-shot probing.

TABLE 14: The 31 Google API endpoints used for validation.

Service family	Endpoints
Maps Platform (15)	`maps_places_nearby`, `maps_places_text`, `maps_directions`, `maps_distancematrix`, `maps_geocoding`, `maps_elevation`, `maps_timezone`, `maps_staticmap`, `solar`, `maps_streetview`, `geolocation`, `routes`, `addressvalidation`, `airquality`, `pollen`
YouTube Data	`youtube_videos`, `youtube_search`
Translate	`translate_languages`, `translate_text`
Vision	`vision_annotate`
Natural Language	`nlp_sentiment`
Text-to-Speech	`texttospeech`
Gemini / GLA	`gemini_models`, `gemini_generate`
Knowledge Graph	`kgsearch`
Custom Search	`customsearch`
Google Books	`books`
Web Fonts	`webfonts`
PageSpeed Insights	`pagespeed`
Civic Information	`civicinfo`
Fact Check Tools	`factcheck`

Profiling each directly usable key by the endpoints it reaches turns this scope into a capability map. Figure 6 reports the per-service distribution across the 1,010 directly usable keys, separating the low-risk Maps/Geo utilities from the high-value tail that reaches endpoints such as Gemini and Vision.

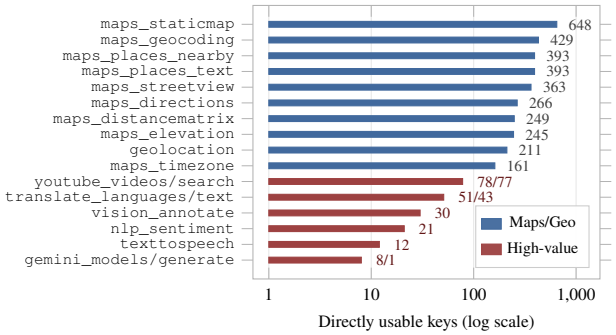


Figure 6: Service-level distribution of directly usable API secrets.

Appendix F. Ethics Considerations

Our research involves large-scale measurements of exposed secrets and security misconfigurations. Recognizing the inherently sensitive nature of this study and our interactions with live third-party systems, we strictly adhered to the principles outlined in the Menlo Report [33]. We designed our methodology to prioritize harm reduction and to affirmatively protect affected users and organizations.

Interactions with Live Systems and Probing Boundaries. To assess the security impact of the exposed secrets, it was necessary to interact with live systems owned by various service providers. Obtaining explicit prior permission from each system owner was practically infeasible because of the large scale of the measurement. Therefore, we implemented system-wide constraints to limit operational and billing impact. Our collection agents (T1–T5) issue anonymous `GET` requests only and never brute force enumerate directories, bypass access controls, or touch post-login APIs. Following the harmless-verification methodology of prior secret measurements [14], [73], [64], validation was restricted to documented, read-only Google API requests, using the free tier where available and the lowest-cost non-destructive variant otherwise (Appendix E). Each key–service pair was probed once initially and at most once three weeks later, for at most two requests. We excluded writes, deletions, authenticated sessions, privilege escalation, brute-force requests, and access to owner data; we retained only status codes and minimal metadata and did not store response bodies. Any cost would arise only under the conservative counterfactual that ordinary visitor traffic, the key owner's routine business use of the service, or other traffic had exhausted the allowance before our test. In other words, the website's ordinary service traffic may have consumed the allowance before we initiated validation. Our testing volume was equivalent to the minimum load generated by a normal visitor using the web provider's domain under its normal usage. Under that counterfactual, the maximum estimated cost across affected domains was $0.47, and 87.6% of domains were below $0.10. For our LLM baseline experiments, secret values are masked before any third-party model call, ensuring that no live secrets leave our environment in cleartext.

Data Handling and Sensitive Information (PII). The artifacts and secrets extracted during our measurements inherently pose risks if mishandled and may inadvertently contain Personally Identifiable Information (PII). To mitigate these risks, all collected data were encrypted immediately upon extraction and stored on secure offline servers equipped with strict access controls. We maintain a firm commitment that no raw datasets containing actionable secrets, sensitive configurations, or PII will be published or shared.

Vulnerability Disclosure Process. Given the extensive scale of the discovered exposures, yielding 8,947 disclosure records sent to 1,367 responsible parties (since one responsible party may own multiple affected domains), we initiated a coordinated responsible disclosure process. Following the minimum-information principle [55], [43], notices carry only the host, asset URL, secret type, and verification result; no raw values or matchable formats appear in any public artifact. To maximize reach, we attempted seven channels in priority order across all affected domains (e.g., `security.txt` [39], VDP directories, targeted crawls of contact pages, and WHOIS fallback), successfully yielding at least one usable contact for 73.6% of responsible parties. Of the 1,006 reachable responsible parties, 785 had an email endpoint. As of submission, we have completed all email notifications to these endpoints and submitted the corresponding platform reports, and 53 organizations have confirmed or remediated the exposure. We are actively continuing contact form submissions and are committed to finalizing the notification process no later than the rebuttal deadline. To strictly preserve double-blind anonymity, all case studies presented in this paper are anonymized: secret values are masked and the underlying service providers and affected organizations are not identifiable, and no specific CVE identifiers or identifiable vendor correspondence were included in this submission.

Dual-Use Concerns and Tool Release. We acknowledge that the automated secret-harvesting and validation pipeline developed for this research could be misappropriated by adversaries to conduct malicious reconnaissance attacks. To balance the academic necessity for reproducibility with our ethical obligation to prevent gratuitous harm, the open-source release of our artifact was deliberately constrained. We will release the core analytical framework and detection logic; however, we will intentionally omit the active probing payloads and validation modules to prevent the direct weaponization of our software.

Appendix G.
Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

G.1. Summary

This paper presents KeySifter, a pipeline for detecting and assessing browser-visible secrets, such as API keys and other credentials inadvertently exposed in web assets. The system combines probing rules derived from 10 web frameworks with AST-based backtracking to detect secrets hidden in minified code that may be missed by traditional scanners. KeySifter then applies a Risk-Pattern Knowledge Base (RPKB) to reduce false positives and prioritize high-risk leaks. The authors evaluate KeySifter on 65,000 domains randomly sampled from the Tranco Top 1M list, identify more than 1,200 instances of secret leakage, and conduct responsible disclosure with affected parties.

G.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

G.3. Reasons for Acceptance

1) The paper introduces a new tool that can enable future research. KeySifter is a framework-aware crawler and classifier capable of autonomously detecting publicly visible secrets in web assets. It introduces novel techniques, including AST-based call-site backtracking for minified code. The system achieves high precision and recall and substantially outperforms existing rule-based scanners in the authors' evaluation. The authors have also committed to releasing KeySifter to the research community, which could facilitate future work in this area.
2) The paper provides a valuable step forward in an established field. While secret scanning in source code is well studied, this work addresses the distinct challenge of detecting secrets in browser-visible assets, where minification, bundling, and other build artifacts complicate detection.