skarn

RESEARCH / MEASUREMENT

How precise is secret detection? We measured it.

0.0%

false-positive rate over a labeled corpus of 40 benign, credential-shaped values.

Zero false positives, 100% recall on the 111 formats tested across 73 services, and the 10 formats we do not yet catch published in full.

Executive summary

Skarn's secret scanner raises 0 false positives over a labeled corpus of 40 benign, credential-shaped values: a measured 0.0% false-positive rate. On the same corpus, 111 current credential formats across 73 services are all detected - 100% recall on the tested corpus - and the 10 current formats Skarn does not yet match with a dedicated rule are published as an explicit gap list rather than quietly omitted.

Nearly every fixture is grounded in a cited public source - a scanner detector definition, an official provider document, an RFC or specification, or a documented scanner false positive - and every credential in the corpus is synthetic and non-functional. The false-positive rate is the headline because it is the number that decides whether a tool you run against your own work every day is one you trust or one you mute.

The figures at a glance: 0 false positives over 40 non-secret decoys (0.0%); 111 current formats across 73 services detected (100% recall on the tested set, held against regression); 10 formats published as known gaps; 5 documented benign false-positive families; 161 labeled secret-track fixtures, 158 of them carrying a cited public source; 189 synthetic secrets exercised, none ever shown unredacted in any output.

This report gives the full method, the complete corpus breakdown, and a short checklist for measuring your own exposure with the same tool.

What we measured, and why the false-positive number is the one that matters

Every secret scanner claims low false positives. Few show their work. A scanner that flags your git commit hash, your UUID request IDs, the base64 image in your README, and the example key in your quickstart is a scanner whose alerts you learn to ignore - and an ignored scanner misses the real leak. So we did what a detector should always be made to do: we built a labeled corpus of the benign values that look like credentials, measured the false-positive rate, and - unusually - we also published the list of formats we do not yet detect.

A precision claim is only worth as much as the corpus behind it. The credible bar on the secrets surface was set by published benchmarks with a named dataset, a stated method, and exact precision and recall figures. We follow that bar: a defined corpus, a method published in full, and a number that is the output of one command - reproducible and falsifiable, not a marketing figure.

The false-positive rate is the number that matters for a tool you run against your own work every day. The hard part of secret detection is not matching `AKIA...`; it is staying silent on the forty other things that look like `AKIA...`. A 40-character hex string is a git SHA, a content checksum, a request ID, or an AWS secret key, depending only on context. Skarn's value is that it tells them apart.

Where this sits against the other local scanner

Skarn is not the only scanner that reads this surface locally. GitGuardian's Developer Endpoint Protection, announced 2026-06-16, scans AI tool directories and log files on the endpoint, and states that only structured metadata leaves it - so the honest axis of comparison is not local versus cloud. It is what gets correlated.

The documented scope of that product is a credential inventory, honeytokens, and forwarding to a SIEM; as of that announcement it documents no attack-chain, behavioral, or MITRE ATLAS correlation across the session. A credential inventory answers "is there a key here". Skarn measures precision on that question and on the harder one next to it - did this session behave like an attack - and publishes the false-positive rate for each, with the command that produces it. The companion attack-chain report is the other half of that pair.

How the corpus is built: real scenarios, synthetic secrets

Two rules govern every fixture.

Real scenarios. Every credential format is grounded in its authoritative source - an official provider doc, a published detector definition, or a format spec. Almost every benign value is grounded in the spec or the documented scanner false positive that establishes it as a non-secret: the UUID and JWT RFCs, the ULID, nanoid, and KSUID specifications, the W3C trace-context header, the Subresource Integrity spec, the npm and Go module checksum formats, and real scanner false-positive reports. 158 of the 161 secret-track fixtures carry that citation in their label metadata. The other 3 are grounded in something narrower but no weaker: they are regression guards for false positives this benchmark caught in Skarn itself, which we then fixed.

Synthetic secrets. Not one real credential appears anywhere in the corpus. Every token is synthetic, format-valid, and non-live - it matches a detector's shape, and its checksum where one applies (such as a GitHub token's CRC), but unlocks nothing. The two values that look most real are deliberately the universal public documentation constants - the example JWTs printed in the spec and on jwt.io, and a payment provider's shared sample test key - included as decoys precisely to prove the scanner stays quiet on documentation. An automated invariant extracts every secret-shaped token from the corpus and proves none of them ever appears, unredacted, in any output Skarn produces; 189 synthetic secrets are pushed through that gate and it passes.

Extended methodology

What the numbers measure, precisely

Two distinct measurements run over the credential track of the benchmark corpus.

Precision, the hero number: of 40 benign decoys - values that look credential-shaped but are not secrets - how many produce any finding. A decoy counts as a false positive if it produces any incident at all, from

any rule, structural detector, or generic heuristic. "0 false positives over 40 non-secret decoys" is therefore exact: nothing fires on any of the 40.

Coverage, held against regression: of 111 positives - real, current credential formats represented by synthetic non-live tokens - how many fire their expected rule. All 111 fire. A positive that stops firing is a recall regression.

This is a precision measurement over curated false-positive families plus a coverage measurement over current formats. It is not a base-rate prevalence claim about how often these values occur in the wild.

Corpus composition

The credential track holds 161 labeled fixtures: 111 current-format positives across 73 distinct service formats, 40 benign decoys across the 5 documented false-positive families, and 10 known-gap fixtures for current formats with no dedicated rule yet. It sits inside a larger labeled benchmark of 261 fixtures that also covers the behavioral, supply-chain, lockfile, and insecure-code tracks. 158 of the credential-track fixtures carry, in their label metadata, the real public URL that establishes the format or the non-secret status; the remaining 3 are regression guards for defects this benchmark found in Skarn.

Grounding and verification

Fixtures are not invented. They come from a structured research pass across the coverage families (current provider key formats) and the false-positive families (non-secrets mistaken for keys), followed by an adversarial fetch-and-confirm step against each cited source. Crucially, Skarn's own scan - not the research prose - decided coverage: each candidate format was generated as a synthetic token and scanned, and labeled a positive only if its expected rule actually fired, otherwise a tracked gap. Several research claims that a service was "uncovered" turned out to be wrong; the rules existed and the tokens fired. Ground truth came from the scanner, not from a writeup.

Building the corpus found real bugs

Constructing an adversarial precision corpus is supposed to find false positives, and it did: genuine ones in pre-existing detectors - among them the canonical example JWTs from the spec and jwt.io, a payment provider's shared sample test key, public-key configuration assignments that are published by design, a public analytics beacon token embedded in page HTML, the value of a field whose own name says it holds a digest, and angle-bracket template placeholders. Each was fixed with an additive suppression and a regression test, and no real detector was weakened and no existing suppression removed. A test-mode key, or a longer real key that merely shares a documentation prefix, still fires. Finding and closing these is the point of the exercise, not a footnote to it - and it is why a handful of the decoys cite this benchmark rather than an outside source: the defect is their provenance.

Honest limitations

Stated plainly so the number survives scrutiny rather than inviting it.

It is precision, not prevalence. The number is a false-positive measurement over a curated corpus, not a field study. It says Skarn does not false-fire on these 40 real benign values - not that the false-positive rate in the wild is exactly zero. A corpus sampled from real traffic would strengthen it further; this one is built adversarially from the documented false-positive families, which is the right first bar.

Recall is reported, not hidden. All 111 tested formats fire - 100% recall on the tested set, held against regression - and it is stated alongside the false-positive rate. The 111 positives are current-format recall guards across 73 services, not a claim of completeness. The 10-entry gap register is the explicit incompleteness statement, and several of those gaps are still caught generically, so the true miss set is smaller than 10 and the register is a conservative upper bound.

Some lookalikes are flagged on purpose. A test-mode key and the canonical cloud-docs example access key are intentionally still treated as secrets; only the universally-shared documentation constants are suppressed. A test-mode key is a real, if low-blast-radius, secret, so environment classification down-ranks it rather than hiding it.

The benign families we test against

These are the values that fool a naive secret scanner. Skarn stays silent on all 40, across five families.

Identifiers (12). Public, deliberately-shared correlation keys: UUIDs, ULIDs, nanoids, KSUIDs, distributed-tracing identifiers, error-tracker event IDs, and the object and request IDs that payment and cloud APIs return on every call.

Content hashes (8). Digests fingerprint public artifacts, not credentials: git SHA-1 and SHA-256 object names, Docker image digests, Cargo and Go module checksums, Subresource Integrity attributes, npm lockfile integrity fields, and the value of a field whose own name says it holds a digest.

Example and placeholder keys (8). Documentation and tool output, not live credentials: sample keys, publishable client-side keys, asterisk-masked scanner output, `.env.example` placeholders, angle-bracket template placeholders, and a licence token quoted in prose or carrying a checksum that does not validate.

Code and config identifiers (7). Symbol references, names, and version pins, never values: dotted code references, assembly public-key tokens, environment-variable names referenced in deploy scripts, public-key config fields, a public analytics beacon token embedded in page HTML, and version pins that hold a commit SHA.

Encoded data (5). Presentation and ciphertext blobs, plus canonical public example tokens: inline base64 images and fonts, sealed and encrypted ciphertext, and the example JWTs from specs and docs.

The corpus is built to be hard, not easy. It includes the boundary cases - a publishable key versus a secret key, a variable's name versus its value, a 40-hex commit SHA versus a 40-hex secret, a licence token whose checksum validates versus one whose checksum does not - precisely the places a sloppy scanner turns a config file into a false alarm. Each one trips a different suppression and stays silent, and each is a regression guard that fails the build if it ever starts firing.

The formats we detect

The same corpus proves the scanner did not go quiet to win on precision. The 111 positives confirm current-format detection across 73 services in five groups.

AI and LLM providers: OpenAI (project, service-account, and admin key shapes), Anthropic, Groq, DeepSeek, xAI, Perplexity, Replicate, Hugging Face, Cohere, Mistral, Together, Fireworks, NVIDIA, OpenRouter, Voyage AI, Jina, Langfuse, Mem0, LlamaCloud, Cursor, Composio, RunPod, MiniMax, Friendli, Zhipu, Dify, Baseten, Ollama, and Deepgram.

Cloud: AWS, including both Bedrock key lifetimes, plus Azure, DigitalOcean, Fly.io, and Render.

Version control and CI/CD: GitHub and GitLab in their token variants, Bitbucket, npm, PyPI, RubyGems, Docker Hub, Buildkite, CircleCI, Vercel, Netlify, Jenkins, AppVeyor, and TeamCity.

Payments and communications: Stripe, Twilio, SendGrid, Mailgun, Slack, Discord, Microsoft Teams, Telegram, Postmark, Square, and PayPal.

Data and infrastructure: Supabase, Neon, MongoDB, Redis, Vault, Doppler, Dynatrace, Grafana, 1Password, connection strings, JWTs, SSH and PEM private keys, and Skarn's own licence tokens.

Each positive is generated to match the live rule, so a miss is a real recall regression rather than a silent gap.

What we do not yet detect

10 current formats have no dedicated rule yet: a handful of newer AI providers, including one whose newest key shape is still rolling out; a cloud storage shared-access-signature token; several unprefixed cloud and VPS provider tokens whose shape is too generic to match safely; a package-registry token; a managed-Redis token; and one managed-database password prefix. Some are still caught by generic high-entropy detection, with less precision and no service attribution.

We track them as a published gap list rather than add aggressive rules that would regress precision elsewhere - because on a tool you run daily, a quiet, honest scanner beats a loud, overreaching one. The list shrinks over time; the honesty does not. A scanner's candour about what it misses is part of its trustworthiness, so we publish the gap register next to the precision number rather than behind it.

Reproducibility

The number is not a slide. It is the output of one command - so here is the command, the corpus it runs over, and the scorecard it prints.

The benchmark and the labeled corpus ship with the tool. One command scores the whole corpus and gates the build: it fails the moment one new benign value trips a detector, one current-format positive stops firing, or a number published in this report stops matching the corpus. A second command proves the corpus holds no live credential. A drift guard checks the figures surfaced in the binary itself against the corpus, so they cannot quietly diverge, and `skarn taxonomies` prints the same figures.

```
zig build -Dtest-mode benchmark
zig build -Dtest-mode redaction-check
```

What the first command scores, on the secret track: 161 labeled fixtures, in the four label kinds the corpus uses.

- 111 positives. Current credential formats across 73 services, each a format-valid synthetic token generated to match the live rule. Each must fire its expected rule; a miss is a recall regression and fails the build.
- 40 decoys. Benign, credential-shaped values across the 5 false-positive families: 12 identifiers, 8 content hashes, 8 example and placeholder keys, 7 code and config identifiers, 5 encoded blobs. Each must produce no finding at all; one finding is a false positive and fails the build.
- 10 known gaps. Current formats with no dedicated rule yet. Tracked, published in this report, and deliberately gate-neutral: they are the register, not a failure.
- 0 known bugs. The fourth kind, for a rule that exists but is defective. That register is currently empty.

Every token in the corpus is synthetic and non-live, and that is not an assurance but a second gate: `zig build -Dtest-mode redaction-check` extracts all 189 secret-shaped tokens from the corpus and asserts that none of them appears unredacted in any output Skarn produces - every format, the baseline file, the audit log, the export, the local scan API, and the guard verdicts.

The scorecard the gate prints:

```
FALSE-POSITIVE RATE (benign fixtures that fired any rule)
  all decoys:        0/108  (0.0%)
  behavioral decoys: 0/59  (0.0%)   <-- attack-pattern / chain precision (L1)
  secret decoys:     0/40  (0.0%)   <-- leaked-credential precision
  secret recall:     111/111  (100.0%)   current-format coverage (held against
regression)

GATE: PASS
```

The raw corpus is not published as a download, and the reason is the same one that governs the rest of this report. The fixture bodies, the generator, and the scoring rule map together spell out the exact shape each detector keys on and the precise boundary of every suppression - a tuning guide for evading the scanner. What we miss is not the withheld part: the gap register is published in full, above. What is withheld is the map of how each rule decides. The manifest, the command, and the expected output are the falsifiable part - they state exactly what is measured and what result would contradict it.

How to assess your own exposure

The corpus shows the shape of the problem. Measuring yours takes a few minutes and never leaves your machine.

1. Install Skarn. It is a single static binary for macOS, Windows, and Linux, on both Intel and ARM.
2. Run `skarn check`. It scans your AI coding-assistant sessions locally - Claude Code, Gemini CLI, Codex CLI, Cursor, and GitHub Copilot - with no network calls by default. Nothing is uploaded.

3. Read the redacted report. Secrets are never displayed in full: each finding is masked to a fixed form that does not disclose the value's length, so the report is safe to keep and to share with whoever owns remediation.
4. Act on the findings, and trust the silence. When Skarn raises a finding, the precision measured here is what makes it worth investigating rather than dismissing. When Skarn stays quiet on your commit hashes, request IDs, and example keys, that silence is measured, not assumed. Rotate what it finds; Skarn surfaces exposure, it does not change your credentials for you.

Sources

The credential formats are grounded in scanner detector definitions and official specifications; the benign families cite the same specifications and the documented scanner false positives that establish each value as a non-secret. Every source below was re-verified on 2026-07-12, and the corpus counts in this report were re-measured against the live gate the same day.

- gitleaks default configuration, the detector definitions the credential formats are grounded against: https://github.com/gitleaks/gitleaks/blob/master/config/gitleaks.toml
- trufflehog detectors, provider key formats and shapes: https://github.com/trufflesecurity/trufflehog
- gitleaks issue 1578, generic-api-key false positives on `public_key=` and similar names: https://github.com/gitleaks/gitleaks/issues/1578
- gitleaks issue 1775, generic-api-key false positives on git commit hashes in build config: https://github.com/gitleaks/gitleaks/issues/1775
- gitleaks issue 1728, Kubernetes SealedSecret ciphertext detected as a generic API key: https://github.com/gitleaks/gitleaks/issues/1728
- gitleaks issue 1830, entropy detection firing on plaintext variable names and placeholders: https://github.com/gitleaks/gitleaks/issues/1830
- trufflehog issue 4631, detector false positives on checksums, encrypted data, and service names: https://github.com/trufflesecurity/trufflehog/issues/4631
- detect-secrets issue 384, base64 images in notebooks flagged as high-entropy secrets: https://github.com/Yelp/detect-secrets/issues/384
- RFC 7519, JSON Web Token: https://datatracker.ietf.org/doc/html/rfc7519
- RFC 4122, Universally Unique Identifier (UUID): https://datatracker.ietf.org/doc/html/rfc4122
- W3C Trace Context, the traceparent header: https://www.w3.org/TR/trace-context/
- W3C Subresource Integrity, the integrity attribute hash form (also used by the npm lockfile): https://www.w3.org/TR/SRI/
- ULID specification: https://github.com/ulid/spec
- nanoid: https://github.com/ai/nanoid
- KSUID: https://github.com/segmentio/ksuid
- Go modules reference, the go.sum checksum format: https://go.dev/ref/mod

- .NET strong-named assemblies, the public-key-token format: https://learn.microsoft.com/en-us/dotnet/standard/assembly/strong-named
- GitGuardian, extending our mission with Developer Endpoint Protection (2026-06-16), the source for the endpoint-scanner scope described above: https://blog.gitguardian.com/extending-our-mission-with-developer-endpoint-protection/

Provider key formats for the 73 covered services are grounded in the gitleaks and trufflehog detector definitions above and the respective official provider documentation.