Static AST-Based Triage of Socially-Engineered Developer Lures: Proving Obfuscator Rotations Without Execution

Ilja Krasnjanski

Independent researcher

ORCID: 0009-0007-1039-5930

31 October 2026 · Preprint

Abstract

Developer recruitment attacks hide executable commands in verification pages, assessment repositories and editor extensions, and static inspection is difficult when clipboard handlers assemble commands through helper functions and obfuscators replace configuration strings with calls into encoded, rotated arrays. We built `unk-scanner` to recover these structures without executing the submitted code. It follows clipboard values through the abstract syntax tree and evaluates a restricted subset of expressions, and for recognised obfuscator loops, it recovers array order by reproducing the checksum condition in a restricted evaluator, with explicit assumptions governing the correctness of the recovered rotation. Detection rules combine indicators to distinguish suspicious execution paths from techniques also used by legitimate developer tools. The evaluation covers approximately 2.8 million files from software registries and a public malware dataset, retained UNK_DeadDrop artifacts, controlled obfuscation experiments and evasion fixtures. The isolated scanner processed 272 calibration files per second. Two error-level alerts among 19,976 recently changed npm packages were false positives. Controlled experiments across five obfuscator releases reached 99.7 to 99.8 percent string recovery with the revised engine; every decoded call site matched its original value, yet detection coverage remained limited: that engine warned on 128 of 2,122 public malware samples. Small source rewrites also exposed missed behaviours and an incorrect rotation assumption. Recovered strings made hidden code inspectable. Accurate recovery nevertheless coexisted with low malware coverage and easily evaded rules.

Keywords: static analysis; JavaScript deobfuscation; software supply chain; ClickFix; editor extensions; developer security

1. Introduction

Supply-chain defences commonly examine packages that execute malicious code at installation or import, including typosquats and compromised dependencies, while recruitment campaigns introduce another route: an assessment page persuades a developer to paste a verification command, or a take-home repository configures the editor to install an extension. The malicious code may run only after this interaction. Its purpose can be difficult to determine from a brief source review.

The studied ClickFix lures construct commands when the user copies them. Platform checks select helper functions that assemble the command; reading clipboard-call literals misses that construction, while obfuscated extensions hide configuration and module names behind decoder calls into rotated arrays.

Recovering those values is only part of the problem. Legitimate tools use the same techniques: language clients run helpers through the editor binary with `ELECTRON_RUN_AS_NODE`, and commercial extensions use string-array obfuscation.

Dynamic analysis can reveal execution behaviour, but running hostile code may leave platform-, session- or server-dependent branches unexplored; we ask how much command construction and hidden configuration static analysis can recover, and how well those structures distinguish malicious artifacts from ordinary developer software.

The scanner traces clipboard writes into local builders and caller arguments, and a bounded evaluator resolves supported expressions and solves string-array rotations from the obfuscator's checksum. Checksum-based recovery is also used by synchrony [8]. Here it shares an evaluator with the detection rules. Severity depends on combinations of techniques rather than obfuscation or Electron use alone. The implementation adds process isolation and SARIF output.

The evaluation measures alert volume, parser coverage, throughput and sensitivity on three registry-derived corpora, then tests recovery and bypasses using retained artifacts and inert variants; a controlled experiment with 7,200 obfuscated samples of known content measures whether recovered strings are correct, and compares recovery with three published deobfuscators. It shows both the practical value of exposing hidden commands and the fragility of structural rules: low error-level alert volume does not imply broad detection coverage.

2. Background and threat model

We consider JavaScript delivered to a developer through social engineering:

- ClickFix pages that substitute a terminal command on copy;
- repositories whose workspace configuration (for example a `runOn: folderOpen` task) runs or installs code;
- editor extensions, sideloaded or published;
- npm packages consumed as part of an assessment.

The UNK_DeadDrop campaign used all of the first three [1, 2]. Related activity is tracked publicly as Contagious Interview (MITRE ATT&CK G1052) [4], first reported as fake job interviews that ask developers to review an npm package [12].

The obfuscator replaces string literals with calls to a decoder, such as `_0x2bac03(0x530,'rkdk',…)`. Wrappers map their arguments to an index and, for the RC4 variant, a key. The decoder subtracts an offset, reads an element of a stored array, decodes it (plain, base64 with a file-specific alphabet, or RC4 over base64), and caches the result. The stored array is rotated. At start-up an immediately invoked function repeatedly decodes a few numeric-looking strings, combines them with `parseInt` into a checksum, and shifts the array until the checksum equals a constant embedded in the file. A wrong rotation does not fail. It silently returns other entries of the same array.

The analyst holds a file, an unpacked package or an extension, and wants to know what it is set up to do without running it, while the adversary controls the source and may target the analyser:

deep nesting, pathological sizes, code designed to exhaust the parser, or inputs crafted to make an evaluator perform side effects. The corpus evaluation measures existing code; the bypass study separately considers an adversary who knows the rules and changes the input to evade them. Out of scope are code fetched at run time that is not present in the file, native payloads (handled by file-content signatures), and proving reachability.

3. Design

The engine parses each file once and indexes parent links and function scopes (Figure 1). It recovers string arrays before applying the rules. Recovery and detection share a pure evaluator. Parsing, recovery and rules run inside an isolated process; only the report leaves it (Section 3.6). Baseline results use source hash `caf3f552`. Later hardening and decoder-recognition changes appear separately in Sections 5.6 and 5.7 and do not alter those baseline measurements.

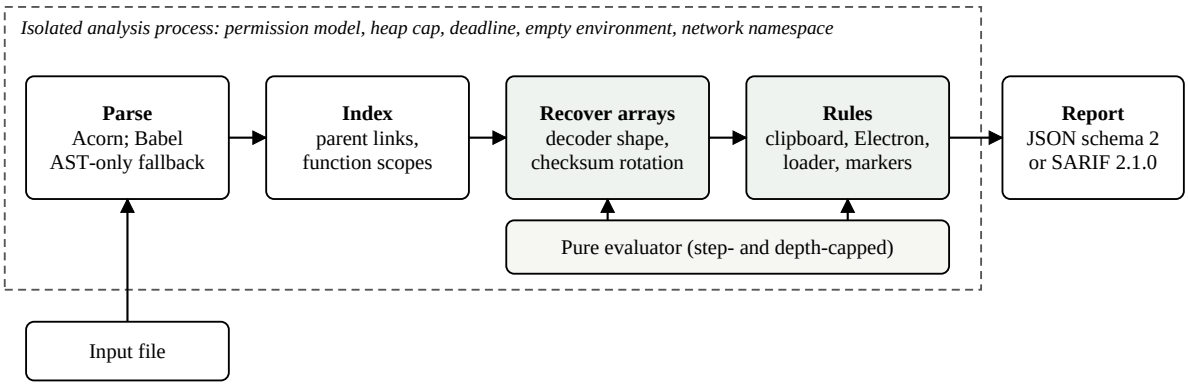


Figure 1. Analysis pipeline. A file is parsed and indexed; obfuscated string arrays are recovered before the rules run; recovery and rules share a step- and depth-capped evaluator that never calls into the input. The dashed boundary is the isolated analysis process; only the report leaves it.

3.1 Parsing

Files are parsed with Acorn as ECMAScript module, then as script, and when Acorn rejects a file, `@babel/parser` parses it in AST-only mode with the TypeScript and/or JSX plugins, chosen by file extension or tried in turn, and emits ESTree. The same rules therefore apply to TypeScript sources and JSX components. TypeScript-only wrappers (`as`, `satisfies`, non-null assertions) are transparent to evaluation and member resolution. Location objects are not materialised. Positions are computed from offsets only for reported nodes, which lowers peak memory on large bundles. All tree walks are iterative.

3.2 Pure partial evaluation

The evaluator interprets the following:

- literals and templates;
- arithmetic, bitwise, comparison and logical operators on primitives;
- `const` bindings and properties of literal objects;
- string and array indexing and `.length`;
- `parseInt`, `String.fromCharCode` and a fixed list of string methods;
- functions whose body is a single `return` of a supported expression, with parameters bound to argument values.

Any other construct yields *unknown*, which propagates. This includes calls on unknown values, `let`/`var`, assignments, loops, getters, spreads and regular expressions. Each evaluation is capped at 20,000 steps and depth 48, strings at 1 MiB and arrays at 10,000 elements. Scopes are function-level. A name declared twice in one scope becomes unknown. Unsupported operations remain unknown rather than being delegated to the JavaScript runtime. Input-supplied functions

never execute. Correct interpretation of the supported subset is a separate requirement, tested by the recovery experiments.

3.3 Proving the rotation of string arrays

An array source is a self-redefining function that holds a string-array literal, or an array variable that rotation code shifts; a decoder rebases its first parameter (`i = i - OFFSET`) before indexing the source, and the scanner selects the innermost function that performs these operations. The offset is evaluated. Candidate variants come from the decoder's syntax:

- a 65-character alphabet literal indicates base64;
- a used key parameter indicates RC4;
- otherwise the array is plain.

The scanner then decodes in its own code; the input's decoder is never run.

The rotation loop is found as an immediately invoked function that receives the array or fetches it by name and contains `x.push(x.shift())`; within its `try` block, the stopping condition is either `if (v === target) break` with `const v = EXPR`, or `if (TARGET == EXPR) break`. Loop parameters are bound to the call's arguments. For each candidate rotation `r = 0 … n−1`, the evaluator computes the checksum expression through the same wrappers and decoders, and takes the first `r` that equals the target. When several decoders share one array, as with a base64 decoder and an RC4 decoder over the same strings, all combinations of their candidate variants are tried under one shared rotation.

```
Algorithm 1. Rotation by checksum, without executing the input
input:  array source A of n strings; decoders D over A, each with an offset and
        candidate variants; the rotation call F(args), whose try block holds E == target
output: a rotation r, or none
bind F's parameters to args, the array parameter to A itself
t <- evaluate(target)                           -- must fold to a number, else none
for each assignment of variants to D (at most 16):
    for r in 0 .. n-1:
        decode(i) := variant(A[(i - offset + r) mod n])    -- reimplemented decoder
        if evaluate(E, decode) == t: return r              -- first match, as the loop
return none                                     -- consensus or unresolved, below
```

The proof requires a recognised decoder with the correct variant and offset, an evaluator that computes `E` and the target as JavaScript would, and one left rotation per failed iteration; under those conditions, Algorithm 1 and the loop evaluate the same values in the same order. The loop rotates the array after each mismatch, including thrown errors. The algorithm stops at the same first match. Rotations repeat modulo `n`; if no `r < n` matches, the loop never terminates and the algorithm returns none.

An unrecognised rotation step or unsupported expression breaks these assumptions; the evaluator treats an unknown expression as a mismatch, which can skip a rotation, and recovery then returns none unless a later rotation matches. A wrong rotation rarely yields a number: it places non-numeric strings where `parseInt` reads them, so `E` becomes NaN. The algorithm evaluates `E` at most `16·n` times, with each evaluation capped at 20,000 steps. In obfuscator output, `E` contains six to twelve decoder calls.

The input program imports `child_process` and prints a URL alongside the type of `spawn`:

```
const cp = require("child_process");
const url = "https://example.invalid/setup";
console.log(url, typeof cp.spawn);
```

The program never calls `spawn`. Its URL uses the reserved `.invalid` domain.

With rotation enabled, javascript-obfuscator 5.9.0 (seed 19, plain strings) replaces the four application strings with decoder calls. The output contains thirteen array entries. Nine are numeric-prefix strings used by the startup checksum.

The generated program is shown in two blocks below, with declarations placed before their callers. It is separate from the input program above.

```
function W() {
    const a = [
        "416VKoFbO", "68MVpWjx", "spawn",
        "3664340cHkggg", "272844EERGGb",
        "2935488qIidkn", "28811hhXjYz", "2767497Wyegbc",
        "https://example.invalid/setup",
        "3715170urlFCz", "5448065HFAxTT", "log", "child_process"
    ];
    W = function () {
        return a;
    };
    return W();
}

function f(G, d) {
    G = G - 0x107;
    const t = W();
    let E = t[G];
    return E;
}
```

`W()` allocates the array on its first call, then replaces itself with a closure that returns the same object. Subsequent calls share that array. The startup loop therefore mutates the array that `f()` reads.

The decoder subtracts `0x107` (263) from its first argument and returns the entry at that index. Its second parameter is unused. Plain-string decoding needs only the offset and the correct array order.

```
(function (G, d) {
    const t = G();
    while (!![]) {
        try {
            const E =
                -parseInt(f(0x110)) / 0x1
                    * (-parseInt(f(0x10b)) / 0x2)
                + parseInt(f(0x111)) / 0x3
                + parseInt(f(0x10d)) / 0x4
                + parseInt(f(0x113)) / 0x5
                + -parseInt(f(0x10f)) / 0x6
                + -parseInt(f(0x107)) / 0x7
                + parseInt(f(0x10a)) / 0x8
                    * (-parseInt(f(0x10e)) / 0x9);
            if (E === d) {
                break;
            } else {
                t["push"](t["shift"]());
            }
        } catch (L) {
            t["push"](t["shift"]());
        }
    }
}(W, 0xaf1a1));

const cp = require(f(0x109));
const url = f(0x112);
console[f(0x108)](url, typeof cp[f(0x10c)]);
```

The startup loop computes `E` and compares it with `0xaf1a1` (717,217). A mismatch moves the first entry to the end. The exception handler performs the same rotation. Once the checksum matches, the loop stops and the application calls use the resulting order.

Static recovery reproduces that search over candidate rotations. For a left rotation of `r` positions:

```
f(i) = stored_array[(i - 263 + r) mod 13]
```

The evaluator substitutes these values into the checksum expression. It never calls the submitted `W()` or `f()`.

Rotations 0 through 9 produce NaN: at least one `parseInt` operand resolves to an application string with no numeric prefix. Rotation 10 produces 717,217. No other rotation matches. A separate Python implementation of the checksum returns the same solution.

Decoder call	Value at rotation 10
`f(0x109)`	`child_process`
`f(0x112)`	`https://example.invalid/setup`
`f(0x108)`	`log`
`f(0x10c)`	`spawn`

All four values are printable even under the wrong rotation because every array entry is a plain string. Printability cannot select the order here. The checksum supplies the deciding constraint.

If checksum recovery fails, the next step depends on whether rotation code was recognised:

- *No rotation code:* the array is taken as stored ("unrotated").
- *Rotation code but no solvable checksum:* the scanner falls back to *consensus*. It accepts a rotation only if that rotation decodes at least 80% of up to 64 call sites to printable text, and the runner-up decodes at most half as many. A keyless base64 or plain array decodes to printable text under every rotation, so consensus cannot select one and the array is reported as unresolved.
- *Never assumed unrotated:* an array is never treated as unrotated when recognised rotation code exists (Section 5.4 shows an unrecognised rotator that defeats this in the evaluated engine; Section 5.6 describes the change that closes it). An earlier implementation missed a parameterless `for(;;)` loop and incorrectly decoded 22 commercial-bundle files under rotation 0. The decoded text looked plausible under the wrong rotation.

Call sites are calls to a decoder or to a helper that reaches one through single-return wrappers and `const` aliases. Calls inside wrappers are excluded as plumbing. Each site is evaluated. Arguments drawn from constant objects resolve. Recovered strings are substituted wherever rules read literal values, including concatenations of decoded fragments.

3.4 Tracing copied values

Clipboard writes (`navigator.clipboard.writeText`, `clipboardData.setData`) are analysed in three steps:

1. The written value is evaluated. When it resolves, the exact text is classified.
2. Otherwise, string fragments are harvested, not evaluated, from the value expression, from locally defined helper functions and objects of helpers (to a bounded depth and node budget), and, when the value is a parameter of a generic copy helper, from the arguments its callers pass.
3. The text is classified by indicator: pipes into a shell, download cradles, hidden or encoded PowerShell, `iex`, `mshta`, `cmd /c`, base64 decoding, detaching (`nohup`, `setsid`, `perl -MPOSIX`), `osascript`, quiet `msiexec` and `rundll32`.

A finding is an error when a strong indicator is present (hidden or encoded window, `mshta`, base64 decoding, detaching), and it is also an error when the write occurs inside a `copy`/`cut` handler or carries verification-lure wording. A plain `curl … | sh` behind a copy button is a warning, because installation pages do exactly that.

3.5 Severity by combination

Each rule has a default severity and explicit escalation conditions. The calibration principle is that a technique legitimate software uses routinely may not be an error on its own:

- *Electron launches.* Launching a process with `ELECTRON_RUN_AS_NODE` in its environment is a warning, and assigning the variable is a note. Either escalates when the launching function, or a local helper it calls, also hides execution through a hidden-window PowerShell invocation or a script launched from a temporary or `.vscode` directory.
- *Obfuscation.* A string array is a warning whose detail records whether it was recovered.
- *Encrypted loaders.* An encrypted loader is an error. It requires embedded base64, AES-GCM authentication and decoded text passed to `Module._compile`, linked through constant bindings.
- *Rules derived from the case.* Investigation-specific markers are notes.

3.6 Isolation

The command-line tool analyses each file in a pool of child processes. Each process runs as follows:

- under the Node.js permission model, with no file writes, child processes, workers or addons, and reads limited to the scanner, its parser dependencies and named rule files;
- with code generation from strings disabled;
- with a 512 MiB heap and an empty environment;
- with a per-file deadline that kills and replaces the process;
- on Linux, where unprivileged user namespaces are available, inside an empty network namespace with `prlimit` caps on CPU time, file size and open files.

The applied controls are recorded in the output. A hostile rule module that attempts a file write, a process launch and a network connection is refused all three in our tests. These controls limit resource use and access if analysis fails. The tests cover these access restrictions. Containment of arbitrary parser or engine exploits remains untested.

4. Implementation

`unk-scanner` runs on Node.js 22 or later. It pins Acorn 8.17.0 and `@babel/parser` 8.0.8, uses the MIT license, and is published on npm and PyPI. The Python wheel bundles the engine. The JavaScript implementation has about 2,500 lines across fifteen modules for parsing, evaluation, string-array recovery, loader recognition, isolation, the CLI, SARIF output and rules.

Each rule module defines an identifier, default severity, ATT&CK mapping and a `create(context)` function that returns visitors, following ESLint's model, and the engine dispatches visitors in one pass over the indexed tree. Its nine built-in rules cover clipboard use, Electron launches, obfuscation and loaders, and campaign markers, while custom rules receive the same context, including resolved values, harvested fragments and recovered strings, and load through `--rule` or the library interface.

The scanner uses explicit stacks and node budgets for its tree walks, so deeply nested input exhausts a budget instead of the call stack. The parsers still recurse. The engine reports parser stack overflow as a failed file. To reduce memory use, it computes positions on demand from a line index rather than storing Acorn location objects; that change lowered peak memory on a 7.9 MB bundle from 462 MB to 399 MB.

Each report records the accepted parser, every recovered array's variant, offset, rotation method and decoded versus total call sites, the isolation controls, and a fixed list of limitations. Output uses JSON schema 2 or SARIF 2.1.0 with `security-severity` properties for code-scanning alerts; the test suite has 69 cases, including the Section 5.4 bypass fixtures, and runs on Linux, macOS and Windows with Node.js 22 and 24. The companion repository records the evaluated source fingerprints, package metadata and tests [2].

5. Evaluation

The evaluation combines scans of popular software, recently changed npm packages and known-malicious samples, and together they measure alert volume and detection coverage; retained campaign artifacts and single-change variants test string recovery and resistance to evasion. Parser coverage and throughput establish the practical cost of analysis. A controlled experiment with known originals (Section 5.7) measures the correctness of string recovery and compares it with three deobfuscators.

5.1 Corpora

Tier	Corpus	Purpose
Calibration	2,500 npm (`npm-high-impact` 1.13.0 ranking) and 500 Open VSX extensions by downloads; 143,153 unique files	Used during development to set severities; not held out
1	Top 5,000 npm packages (`npm-high-impact` ranking; 4,992 acquired) and top 1,000 VS Code Marketplace extensions by installs (967 acquired), with the calibration Open VSX set; 332,054 unique files, 188,901 not in the calibration corpus	Alert volume beyond the initial calibration set; one rule adjustment informed by Tier 1
2	20,000 most recently changed npm packages from the registry changes feed (sequences 140,571,698–140,654,245, 29 October 2026); 19,976 acquired, 2,403,437 unique files; 84% scoped	Review of recently changed packages
3	Datadog malicious-packages dataset at commit `438afad` [3]: all 122 IDE-extension samples and a seeded random sample (seed 20261029) of 2,000 of 26,715 npm samples, 41,359 unique files; the 14 retained UNK_DeadDrop artifacts	Sensitivity and bypass analysis

Archives are read in memory under size caps; nothing is installed and no lifecycle script runs. Malicious samples remain encrypted at rest, and their members are written only as inert files for parsing.

5.2 Findings in popular software

On the calibration corpus no error-level finding occurred in 143,153 files. All 107 warnings were reviewed:

- 90 were commercial obfuscation, in five packages, all recovered or reported as unresolved;
- 16 were legitimate launches of the editor binary as Node, among them four bundled copies of TypeScript's typings installer;
- one was an installer's copy button.

Because severities were set on this corpus, it shows consistency rather than generalisation. Rescanned with the frozen engine on the evaluation server, the corpus again produced no error and 108 warnings; the additional warning is a TypeScript browser-installer helper in an AI coding extension that the earlier engine could not parse, and it is a legitimate launch. Tier 1 extends the measurement beyond the initial calibration set. Its 332,054 unique files include 188,901 that do not occur in the calibration corpus, drawn from 2,783 packages and extensions, and the engine was frozen before the final run (source hash recorded with the results). On those 188,901 additional files, the final engine raised no errors and 12 warnings; all warnings were reviewed:

- 11 were legitimate launches of the editor binary as Node: helper and port-forwarder launch templates in Microsoft Live Share and GitHub Codespaces, three bundled copies of TypeScript's typings installer, and a theme patcher's safety check and documented uninstall hook;

- one was a commercial theme's licence code, an RC4 string array fully decoded under the checksum-proven rotation, whose hidden strings are licence-shop and update URLs and favicon-based connectivity probes.

An initial Tier 1 run, before the engine was frozen, raised four errors on the Live Share and Codespaces files; the bundles also contain a copy of a privilege-elevation library that runs PowerShell with a hidden window, and the escalation then applied to the whole file. The rule now escalates only when the hidden execution occurs in the launching function or a local helper it calls; this was the only change made after examining Tier 1 results, and it preserved every retained artifact's previous severity. Consequently, the additional files are disjoint from the initial calibration set, but the final result is not an untouched held-out estimate.

5.3 Recently changed npm packages

Tier 2 includes every selected package whose registry document changed in a window of about 83,000 change sequences ending on 29 October 2026, most of them published that day. The scanner raised 2 error-level and 6,215 warning-level findings in 2,403,437 files from 19,976 packages:

Rule	Severity	Findings	Packages
`string-array-obfuscation`	warning	6,142	153
`electron-run-as-node`	warning	68	44
`clickfix-clipboard-command`	warning	3	1
`child-process-hidden-script`	warning	2	2
`electron-run-as-node`	error	1	1
`clickfix-clipboard-command`	error	1	1

Full review classified both error-level findings as false positives. An AI desktop application builds a hidden, encoded PowerShell command to set `ELECTRON_RUN_AS_NODE` for its helper. The helper contacts `127.0.0.1` and uses a local password. A remote-access tool copies a setup guide whose only `curl` call prints an HTTP status code. The word “verify” in the guide triggered the lure indicator.

We reviewed the code behind every clipboard and hidden-script warning. For Electron launches, we inspected five packages whose names did not suggest developer tooling and triaged 39 others from their recorded launch target and environment, and their contexts fit helper relays, licence checks, launcher shims, desktop-app smoke tests and bundled TypeScript.

We screened recovered strings from 153 obfuscated packages for process, shell, exfiltration-service, wallet and credential-store indicators, and seven packages matched; their code served a Telegram and Discord bot, an OAuth workspace connector or agent CLIs. Nine packages retained unresolved arrays, eight from one commercial product. The review classified no package as malicious and produced no registry report. Unresolved arrays leave some package behavior unknown.

Weekly downloads were retrieved in bulk for unscoped names and individually for every flagged scoped package and a seeded sample of 500 other scoped packages. Flagged packages are not concentrated in the long tail: of the 168 flagged scoped packages, 7 had fewer than 100 weekly downloads and 65 had 10,000 or more, against 80 and 87 in the 500-package sample. These counts show that alerts also arise in widely used tooling. At error level, 2 of 19,976 packages (0.01%) required review. This is the observed package-level review rate, not a population false-positive estimate.

5.4 Detection coverage, recovery and bypasses

On the retained UNK_DeadDrop artifacts, all three ClickFix lure generations, three encrypted loaders, three decrypted collectors and the Windows Node/Python pipeline reach error level, while the two obfuscated extension files are recovered and reported as warnings. The Windows Node agent produces only notes, a miss: its collection code carries no launch, clipboard or loader structure the rules key on. The login interface, which is context-only by design, produces none. These artifacts informed the rules, so they establish coverage, not generalisation.

The checksum method gives rotation 391 for the extension and 261 for its bootstrap, the values established manually during the investigation by a printable-output comparison, and decodes 2,139 of 2,140 and 15 of 17 call sites.

Across the calibration corpus, 486,208 of 486,601 decoder call sites in 90 arrays were decoded; of those arrays, 71 were solved by checksum (69 plain, 1 RC4, 1 shared base64 and RC4) and 19 were left unresolved. The unresolved arrays were reached through reassignable `var` aliases, which the evaluator does not follow. The artifact and calibration figures were reproduced exactly by the frozen engine on the evaluation server.

Tier 3 measures sensitivity on malware outside the campaign. The result bounds the scope of the approach. A sample counts as detected when any of its files raises a warning or error:

Sample set	Samples	Error	Warning	Not detected	No JavaScript
IDE extensions	122	0	4	113	5
npm, malicious intent	1,884	0	44	1,550	290
npm, compromised library	116	0	14	102	0

String-array obfuscation produced 55 detections and hidden-script launches produced seven. No sample reached error level. None used the ClickFix, Electron or encrypted-loader structures covered by the higher-severity rules. Parser failures account for 27 of the misses.

We screened the remaining 1,738 samples for common static traits. These traits also appear in benign code, so the counts describe the misses rather than explain them. Among missed npm samples, 900 read environment variables or home-directory files, 873 use `child_process`, 871 make network requests and 397 reference a webhook or messaging exfiltration service; among missed IDE extensions, 81 use `child_process` and 21 contain variation-selector code points that can hide text.

Credential access and install-time behaviors often fall outside the clipboard, Electron and encrypted-loader patterns modeled here. The retained UNK_DeadDrop artifacts helped shape the rules. Their expected outputs, including the missed Windows agent and the context-only login interface, measure coverage of the development inputs.

To measure how easily the rules are evaded by an adversary who knows them, we wrote inert variants of the demonstration fixtures and of the loader test fixture, each changing one thing, and scanned them with the CLI on the evaluation server:

Technique (baseline)	Variant	Result
ClickFix copy (error)	builder through a `var` alias; builder named via `eval`; command in base64 decoded with `atob`; command as character codes spread into `String.fromCharCode`; command in a mutable variable set elsewhere; copy through a hidden textarea and `execCommand`; command fetched at copy time	all missed
Electron launch (error)	variable name split into constant fragments; name base64-decoded at run time	missed

Technique (baseline)	Variant	Result
Electron launch (error)	editor binary spawned directly with `windowsHide`; hidden window via encoded PowerShell	warning
Encrypted loader (error)	same steps inside a function; `vm.runInThisContext` instead of `_compile`; AES-CBC without tag; `let` bindings	all missed
String array (warning, recovered)	wrapper reached through a `var` alias or an object property	warning, but the aliased call sites (the URL and module name) not decoded
String array (warning, recovered)	checksum target not foldable	recovered by printable consensus
String array (warning, recovered)	rotation moves elements with `splice` instead of `push(shift())`	incorrect: rotator not recognised, decoded under rotation 0

Eighteen of nineteen variants evade the targeted rule, lower its severity or defeat string recovery. The consensus fallback handles only the unfoldable-checksum target. The clipboard, Electron and loader rules recognise observed structures but remain sensitive to small rewrites, and one variant exposed a defect: the rotation rule assumes the scanner recognises the loop. An unknown rotator triggers stored-order decoding, which the rule was designed to avoid.

These results use the unchanged engine. Section 5.6 evaluates a hardened engine against the same variants and corpora. Because the scanner reports partial recovery, analysts must compare decoded and total call sites.

5.5 Throughput and parser coverage

We ran the measurements on one eight-vCPU virtual server with 15 GiB of memory; unless noted, each run used the frozen engine and four analysis workers. On the calibration corpus, the isolated CLI processed 272 files per second (7.4 MB/s); in-process workers processed 264. The isolated run was slightly faster in this comparison. Repeated trials would be needed to estimate the difference and its variance. Median analysis time per parsed file was 1.6 ms, the 99th percentile was 207 ms, and the maximum was 12.5 seconds for an 8 MB bundle.

The isolated path processed malicious Tier 3 samples at 325 files per second, while Tier 2 reached 848 files per second with six workers, reflecting its many small files. Before TypeScript support, Acorn rejected 31,964 calibration files; the Babel fallback reduced that count to 352.

Tier 1 parsed 330,473 of 332,054 files (99.5%): Acorn parsed 251,593, TypeScript 77,521, TSX 668, Flow with JSX 572, and JSX 119; of 1,576 failures, 1,305 were TypeScript files, including legacy `module X {}` namespace syntax. None parsed under the independent Babel configuration we tested. Compatibility with the original toolchains remains untested. Five files were not valid UTF-8. Tier 2 parsed 2,398,405 of 2,403,437 files (99.8%).

5.6 Hardening after the evaluation

After freezing the baseline measurements, we extended the engine to recognise the structures exposed by the bypass tests, and we rescanned every corpus and the retained artifacts on the same server. The results record each engine's source hash separately. The hardened engine adds these analysis paths:

- the evaluator treats `let` and `var` bindings as constants when no assignment in the file resolves to them, and evaluates `atob`, `Buffer.from(...).toString(...)`, spread arguments of known arrays and `eval` of a constant identifier name (other `eval` text is still not interpreted);

- clipboard tracing harvests every value stored into a mutable variable, recognises copy handlers registered as object methods, and treats `document.execCommand('copy')` as a write of the field filled before it;
- the Electron rule resolves script text whose variable name is assembled or decoded, counts encoded PowerShell as hidden execution, and escalates a detached, windowless launch of a binary other than the editor's own runtime;
- loader recognition links an embedded encoded payload, a recognised cipher in any mode and a code-compiling sink (`Module._compile`, `vm`, `eval`, `Function`) through bindings in any scope;
- rotation recognition accepts equivalent steps (`push(splice(0, 1)[0])`, `unshift(pop())`), treats any other reordering in the rotation function as rotation code, accepts the unrotated order only when at least 80% of sampled call sites decode to printable text, and follows call sites through aliases and object members.

With these changes, 18 of the 19 variants reach the baseline's severity and full string recovery; the splice-rotated array now decodes to the baseline strings under the checksum-proven rotation. The exception is the command fetched at copy time, which is not in the file. The retained artifacts again match their 14 expected outputs, and the test suite now includes every variant.

Corpus	Files	New errors	Files with new warnings	Change in string recovery
Calibration	143,153	0	2 (1 package)	90 arrays (19 unresolved) to 96, all solved by checksum; 660,545 of 660,678 call sites decoded
Tier 1	332,054	0	2 (the same package; none in the 188,901 additional files)	19 unresolved arrays to none
Tier 2	2,403,437	0	3 (2 packages)	1,387 unresolved arrays to 16; packages with no resolved array from 9 to 0
Tier 3	41,359	0	3 (2 samples)	detected samples from 62 to 64, none at error level

The new warnings are a commercial extension's genuine obfuscator.io arrays and an AI scripting CLI's string arrays, both declared with `var` and previously skipped, and an installer's `curl … | bash` copy button; the newly decoded Tier 2 strings were screened as in Section 5.3. Two packages newly revealed a `child_process` import: a commercial SAP CAP plugin whose licence code downloads an encrypted per-customer configuration and parses it as JSON, and a commercial developer CLI whose launches list files with `git` and locate the global npm directory. Static reading found no code-execution sink for downloaded data. Neither package was classified as malicious. The Univer Pro modules and the agent library whose arrays had stayed unresolved decode to product-licence and service endpoints and public DNS-over-HTTPS resolvers.

The hardening also reproduced the calibration problem in Section 5.2; a first version escalated any detached Electron launch without a window or stdio, which raised an error on an AI coding extension that relaunches its own runtime (`process.execPath`) as a detached helper server. The rule now escalates only a named binary other than the extension's own runtime; a malicious extension using the same relaunch therefore remains a warning: the syntax does not distinguish them. On the calibration corpus with four in-process workers, the hardened engine processed 281 files/s (median 1.2 ms, p99 210 ms per file), against 264 files/s for the frozen engine in the same configuration, a single-run comparison with no measured slowdown.

The fixes closed the tested bypasses without adding error-level findings. We wrote the variants and designed the hardening against them. Resistance to an adaptive adversary remains unmeasured: another rewrite could reassign an alias, use an unsupported sink or move the command outside the file.

5.7 Controlled string-array recovery

The corpus counts decoded call sites, but real obfuscated code has no known answer key. We built a controlled test from 50 benign calibration files, one per package; each file was a 3 to 30 KB script with at least 15 distinct strings and no findings. We obfuscated each file under 72 string-array configurations: three encodings (none, base64 and RC4), rotation on or off, no wrapper or two variable or function wrappers, calls transform on or off, and index shift on or off. Each sample used its own seed. The test used javascript-obfuscator 5.9.0, current at the time, and 4.1.1, current when the newest baseline was updated. We enabled only string-array transforms, so every hidden string has a known original.

We scored every tool the same way. A string counted as recovered when it appeared in output or, for the scanner, among decoded call-site values, and we counted output as unexplained when neither the original nor obfuscated input contained it whole or as a concatenation; deobfuscators can fold `'a' + 'b'`. Set-level scoring cannot detect a wrong rotation of a plain or base64 array because each call site would decode to another real string from the same program. We therefore compared call sites in source order. The obfuscator replaces each occurrence in place, so the decoded sequence should match the original. We counted sites outside the longest common subsequence as misplaced. The check passed on 25 unencoded samples. Shifting each decoded value one array position misplaced 4,240 of 4,903 sites.

The frozen, hardened and revised scanner engines processed all 7,200 samples. We compared three deobfuscators on a stratified subset of 1,440 samples: ten inputs for each configuration and release. Each tool ran in a separate process with a 60-second limit; the tools were webcrack 2.16.0 [9], which runs the decoder in a sandbox, and the static tools synchrony 2.4.6 [8] and obfuscator-io-deobfuscator 1.0.6 [10].

Tool (samples)	Recall, 5.9.0	Recall, 4.1.1	Samples with unexplained strings
`unk-scanner`, frozen (7,200)	68.3%	0%	0
`unk-scanner`, hardened (7,200)	87.2%	0%	0
`unk-scanner`, hardened (subset, 1,440)	90.3%	0%	0
`unk-scanner`, revised (7,200)	99.8%	99.8%	0
`unk-scanner`, revised (subset, 1,440)	99.96%	99.96%	0
webcrack (1,440)	98.1%	98.1%	144, one input
synchrony (1,440)	0%	98.2%	144, one input
obfuscator-io-deobfuscator (1,440; 144 timed out)	0%	98.0%	144, one input

No scanner engine produced an unexplained string, and every decoded call site passed the site-level check. The hardened engine decoded 638,574 call sites in source order; the frozen engine decoded 492,474. Both left misses unresolved rather than inventing values.

On version 5.9.0 output, the hardened engine solved rotation by checksum in all 1,800 rotated samples, compared with 1,344 for the frozen engine. With rotation code disabled, it accepted stored order in 1,602 of 1,800 samples and left the rest unresolved. Recall reached at least 99.7% in every configuration except one: function wrappers combined with the calls transform fell to 24%. Each wrapper declared a local offset object before returning, outside the single-return helper model in Section 3.2, so the scanner left those call sites unresolved.

The frozen engine's lower recall came from wrapper aliases declared with `var` (64%) and rotations it could not solve. Neither engine recovered strings from version 4.1.1. That release

replaces its decoder with an inner function, which rebases the index on a different parameter, and the engines did not recognise that decoder and reported no array.

The revised engine (source hash prefix `ba4d1468`) addresses both gaps. A helper can now declare side-effect-free locals before its `return`; the evaluator resolves a local only when the returned expression reads it. When a function assigns its own name to an inner function with index rebasing, the engine uses the inner function's offset and key parameters.

On the original 7,200 samples, the revised engine recovered 99.8% of hidden strings from both releases, and it solved every rotated sample by checksum and accepted the stored order of every unrotated sample. Because the fixes were designed against this set, we generated a held-out set after changing the code. It contains 20 new inputs, the same 72 configurations and five obfuscator releases: 5.9.0, 4.1.1, 5.0.0, 4.2.2 and 3.2.7. Three releases were unseen during development. The set adds 7,200 samples.

The revised engine recovered 99.7% of hidden strings in each held-out release, while the hardened engine recovered 85.0% on 5.x and 4.2.2, and none on 4.1.1 or 3.2.7. Across both sets, the revised engine decoded 3,265,560 call sites, all aligned with their original programs, and produced no unexplained string. The held-out set also revealed a gap in our reference sequence: it lacked class-field names. After we added them, every engine had zero misplaced sites.

We reran the revised engine on every corpus on the evaluation server and compared it with the hardened engine in Section 5.6; the calibration corpus produced 15 additional warnings in two commercial extensions. Tier 1 produced 19 additional warnings; four came from three extensions outside the calibration set. Tier 2 produced 917 additional warnings in five packages, including a Salesforce DevOps tool and a browser-testing SDK.

We screened the newly decoded Tier 2 strings as in Section 5.3. Four packages revealed a `child_process` import: the testing SDK, the DevOps tool, a design company's CLI and a self-hosted AI gateway. The first three used their vendors' service endpoints. The gateway contained the bundler's standard stub for an external module; none of its 162 decoded chunks launched a process, and its hidden strings named a licence server and update check. We classified none of the packages as malicious. In these benign corpora, every added warning came from an array solved by checksum or accepted in stored order after the readability check. The engine added no error-level findings. Results on retained artifacts and bypass fixtures stayed the same.

On the public malware dataset, warning detections doubled from 64 to 128 of 2,122 samples; the revised engine warned on 101 npm samples with malicious intent, up from 46, and 12 IDE extensions, up from four. None reached error level. The older decoder shape had hidden many obfuscated samples from the earlier engine.

The three deobfuscators failed on different shapes. On version 5.9.0 output, synchrony found the array but reported “no decoder”; obfuscator-io-deobfuscator returned the array in place, yet both recovered more than 98% of version 4.1.1 output. webcrack recovered 98% from both releases by executing the decoder rather than matching its shape.

webcrack's remaining misses involved rewrites: strings folded into longer literals, a constant `JSON.parse` replaced by an object literal, and an empty string. Each baseline's unexplained strings came from one input. webcrack returned the keys of the rewritten JSON object. synchrony turned an obfuscator variable name into a property name. obfuscator-io-deobfuscator propagated a variable as `undefined` and folded it into an error message, changing the program's output. We did not inspect every missed string from synchrony and obfuscator-io-deobfuscator.

Median time per sample, including parsing and scoring, ranged from 30 to 61 ms for the scanner engines. The last runs shared a server with other benchmarks. The baselines took 290 to 1,045 ms and also regenerated code.

Every scanner engine placed its recovered strings correctly in this controlled experiment. Coverage depended on recognising decoder shapes: two shapes caused major losses, and the fixes came from examining these failures. The held-out releases tested those fixes across further versions

of the same obfuscator. webcrack covered both original releases by executing input-derived decoder code in a sandbox and rewriting the program.

The experiment used benign inputs and one obfuscator family. Control-flow flattening and dead-code injection were disabled. Custom obfuscators and these additional transforms remain outside the measurement.

6. Case study: UNK_DeadDrop

The retained UNK_DeadDrop ClickFix lures assembled detached `curl` and `perl -MPOSIX` commands at copy time. In one generation, the clipboard write appeared inside a generic `copyText(text)` helper; tracing its callers exposed the command builder. The extension instead concealed module names, including `child_process`, and key material in an RC4 string array. Recovering the array made those values available for inspection. The Windows branch generated hidden PowerShell that launched collection code through the signed editor binary with `ELECTRON_RUN_AS_NODE` [1].

The frozen and hardened engines produce identical outputs on all 14 retained artifacts:

Retained artifact	Expected	Result	Rule
ClickFix lure bundles and script (3 generations)	error	error	`clickfix-clipboard-command`
Encrypted stage-2 loaders (3)	error	error	`encrypted-loader`
Decrypted stage-3 collectors (3)	error	error	`electron-run-as-node`
Windows Node/Python pipeline	error	error	`electron-run-as-node`
VSIX extension entry point and bootstrap	warning	warning	`string-array-obfuscation`, recovered
Windows Node agent	note	note	none: a known miss
Panel login interface	none	none	context only

These structures motivated clipboard tracing, decoder recovery and combined launch indicators, and they also explain the evaluation's limits: the rules recognise particular implementations recovered from the case, while the Windows agent's collection code remains below warning level and small rewrites defeat several detectors. Analysts can use recovered strings and call sites to guide examination, but an absence of findings does not clear an artifact.

7. Discussion and limitations

Static analysis exposes code structure and configured actions, not execution or impact; this scanner does not prove reachability, reconstruct the browser's displayed text or recover code fetched at run time. Its evaluator handles only the documented subset; reassignable aliases, dynamic code and unsupported encodings can prevent recovery. Even within obfuscator.io-style code, an unrecognised rotation loop can produce incorrect strings rather than an explicit failure. Partial recovery must therefore be assessed against the number of unresolved call sites.

The corpora answer different questions. Popular packages measure noise in widely distributed software, but are a convenience sample with correlated files; Tier 1 also informed one rule adjustment before the final scan, so its final result is not an independent validation of that rule. The new-package stream measures review burden over one registry window, while the public malware dataset reveals poor sensitivity outside the observed campaign structures. None establishes a deployment-wide false-positive rate.

Manual review was complete for error-level findings. Warning review used code inspection, recorded launch context and recovered-string screening as described in Section 5.3; it was not a comprehensive audit of every flagged package. Unresolved arrays and metadata-based triage leave

uncertainty about those packages. The bypass study further shows that low alert volume can coexist with easy evasion. Closing the evaluated variants (Section 5.6) did not add error-level alerts, but it was tested only against variants known in advance.

Samples were parsed without execution or installation, and acquisition did not run package lifecycle scripts; no Tier 2 package was classified as malicious, so no registry report resulted from that review. Public artifacts provide manifests, hashes and review records without redistributing the source corpus.

8. Related work

GuardDog [5] combines Semgrep and YARA rules over package source with metadata heuristics across ecosystems, including npm and VS Code extensions; Amalfi [6] classifies npm packages using code, metadata and change features. Ohm et al. [7] catalogue software supply-chain attacks and their injection and execution techniques. These approaches address behaviours prominent in the public malware corpus, where the rules evaluated here have low sensitivity; combining delivery-specific inspection with install-script and credential-access analysis is therefore more plausible than treating either as complete coverage.

webcrack [9] regenerates the sample's array, decoder and rotation code, then runs it in an `isolated-vm` sandbox. synchrony [8] and obfuscator-io-deobfuscator [10] match decoder templates and reimplement them. synchrony also evaluates the `parseInt` checksum chain to determine rotation.

Our scanner follows the static approach. It shares a restricted evaluator between detection rules and presents recovered values as triage evidence instead of rewriting the file. Section 5.7 compares the four tools on the same samples. Each static tool recognises different decoder shapes; webcrack's sandbox covered both tested releases.

The scanner produced no unexplained strings, but its recognition limits matter. Section 5.4 shows incorrect recovery when a rotator falls outside the supported syntax, and Section 5.7 shows decoder shapes the evaluated engines missed and the later revision recognised.

Proofpoint [11] and Microsoft Threat Intelligence [13] describe ClickFix lures that induce users to paste commands into a terminal or run dialog; Unit 42 [12] reported developer targeting through fake interviews and npm-based tasks under Contagious Interview, now tracked as ATT&CK G1052 [4]. The present work studies static inspection of related delivery mechanisms; it does not resolve the attribution of UNK_DeadDrop to that group.

9. Conclusion

`unk-scanner` recovers commands and configuration from recognised JavaScript structures without executing the input; its bounded evaluator supports clipboard tracing and checksum-based string-array recovery, while combined indicators reduce error-level alerts on ordinary developer tooling. The evaluated CLI processed 272 calibration files per second with process isolation.

The scanner warned on 62 of 2,122 public malware samples, and 18 of 19 single-change variants evaded a rule, lowered severity or defeated recovery. One variant returned incorrect strings. In the known-answer test, every string the scanner decoded belonged to the original program and appeared at the correct call site. The evaluated engines recognised decoders from one of two obfuscator releases; the revised engine recognised all five releases tested and warned on 128 malware samples.

The hardened engine handled 18 of the 19 variants without adding error-level findings across 2.8 million tested files; the variants came from the authors and do not measure resistance to an adaptive adversary. The scanner exposes evidence for further inspection alongside package and credential-access analysis. A clean scan reports only that the evaluated patterns were absent or missed. Published manifests, review records and bypass fixtures let others reproduce the stated limits and test future changes.

Artifact availability

Source, rules, inert fixtures, benchmark harness, acquisition manifests, review ledgers and the controlled recovery experiment (`benchmark/rotation`, with pinned tool versions, sample manifests and per-sample results) are in the companion repository [2]. Corpus source is not redistributed; selection manifests and hashes allow it to be reacquired.

References

[1] Ilja Krasnjanski. UNK_DeadDrop: An Artifact-Based Study of Recruitment Malware, Credential Collection, and Operator Infrastructure. Zenodo preprint. https://doi.org/10.5281/zenodo.23652297

[2] KrasForge. `unk-deaddrop-companion` and `unk-scanner`. https://github.com/KrasForge/unk-deaddrop-companion

[3] Datadog Security Labs. Open-Source Dataset of Malicious Software Packages. https://github.com/DataDog/malicious-software-packages-dataset (Apache-2.0).

[4] MITRE ATT&CK. Contagious Interview, G1052. The MITRE Corporation, 2025. https://attack.mitre.org/groups/G1052/

[5] DataDog. GuardDog. GitHub, Apache-2.0. https://github.com/DataDog/guarddog

[6] Adriana Sejfia and Max Schäfer. Practical Automated Detection of Malicious npm Packages. In Proceedings of the 44th International Conference on Software Engineering (ICSE '22), ACM, 2022, pp. 1681–1692. https://doi.org/10.1145/3510003.3510104

[7] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2020), LNCS 12223, Springer, 2020, pp. 23–43. https://doi.org/10.1007/978-3-030-52683-2_2

[8] relative. synchrony: javascript-obfuscator cleaner and deobfuscator. GitHub, GPL-3.0. https://github.com/relative/synchrony

[9] j4k0xb. webcrack. GitHub, MIT. https://github.com/j4k0xb/webcrack

[10] ben-sb. obfuscator-io-deobfuscator. GitHub, Apache-2.0. https://github.com/ben-sb/obfuscator-io-deobfuscator

[11] Tommy Madjar, Selena Larson, and the Proofpoint Threat Research Team. Security Brief: ClickFix Social Engineering Technique Floods Threat Landscape. Proofpoint, 18 November 2024. https://www.proofpoint.com/us/blog/threat-insight/security-brief-clickfix-social-engineering-technique-floods-threat-landscape

[12] Unit 42. Hacking Employers and Seeking Employment: Two Job-Related Campaigns Bear Hallmarks of North Korean Threat Actors. Palo Alto Networks, 21 November 2023. https://unit42.paloaltonetworks.com/two-campaigns-by-north-korea-bad-actors-target-job-hunters/

[13] Microsoft Threat Intelligence and Microsoft Defender Experts. Think before you Click(Fix): Analyzing the ClickFix social engineering technique. Microsoft Security Blog, 21 August 2025. https://www.microsoft.com/en-us/security/blog/2025/08/21/think-before-you-clickfix-analyzing-the-clickfix-social-engineering-technique/