

# Parsers of Babel: A Large-Scale Study of URL Parsing Differentials

Michael Blunt, Sohom Datta, Andres Modval, William Enck and Alexandros Kapravelos  
*North Carolina State University*

**Abstract**—Modern web infrastructure relies on URL parsing to identify and interact with resources. When two parsers’ interpretations of a URL diverge, an attacker can exploit the gap to bypass security checks. Prior work analyzing URL parser disagreement lacks the depth and coverage needed to accurately characterize the modern ecosystem. To address these gaps, we develop TURLMOIL, a grammar-based, feedback-driven fuzzer for large-scale differential testing of URL parsers. We evaluate TURLMOIL against the largest corpus of URL parsing libraries assembled to date: 96 parsers spanning 21 language ecosystems. Across 512,424 generated URLs, we find that 99.96% of distinct parser pairs exhibit at least one differential, and 97.92% diverge on every parsed component of a URL. We taxonomize existing parser differentials and identify previously unknown techniques enabling open redirect, server-side request forgery, and logic bypasses. We show that regardless of specification or language, differing URL parsers provide no assurances of security or interoperability. We further demonstrate that parser discrepancies can be used to fingerprint parser implementations on black-box systems, and we apply our dataset of differentials against real-world systems to uncover high-impact vulnerabilities.

## 1. Introduction

URL parsers are a foundational component of modern computing, embedded in web browsers, HTTP clients, reverse proxies, content delivery networks, and backend microservices. Running at the boundaries of web infrastructure, they perform the first step of nearly every web interaction. URL parsers decompose a raw URL string into distinct components over which downstream logic makes security decisions such as routing, caching, access control, and request filtering.

A single user-supplied URL rarely encounters only one parser. As the URL traverses a web system, it passes through a chain of independently implemented parsers, each of which re-derives its own components. The security of that chain rests on an implicit assumption: a URL means the same thing to every parser that reads it. When two parsers disagree, an attacker can craft a URL that satisfies a check at one layer while resolving to a different, attacker-chosen target at the next. This disagreement is categorized as a URL parsing differential. In this study, we consider any differences between two parsed components to be a URL parsing differential. This class has resulted in server-side request forgery, open redirect, and access-control bypass in production systems [1], [2], [3], [4].

Despite a decade of such attacks, the prevailing understanding of the URL parsing ecosystem leaves four gaps. *Scale*: prior corpora are small relative to the deployed ecosystem, with the largest prior measurement covering only 15 parsers [3] and manual studies covering fewer [2], [5]. *Coverage*: prior measurement concentrates on a narrow slice of the component space, over which parsers sharing a standard appear to form largely interoperable clusters [3]. Whether that interoperability holds across the rest of the URL has not been examined at scale. *Input space*: prior automated work generates inputs either by substituting one to four characters of a known valid URL [3] or by mutating a small seed corpus [6]. Neither reaches the delimiter, encoding, and structural interactions that underlie many severe parser bugs [7], [8], [9]. *Applicability*: prior approaches presume the parsers in the request path are already known [3], whereas an attacker in practice faces an opaque stack and cannot tell which parsers a backend runs. No existing approach measures the ecosystem across its full structure, identifies the parsers in a deployed stack, and produces an input that distinguishes them in a single pipeline. We address all four.

We present TURLMOIL (Testing URL Multi-parser Output Inconsistencies at Large-scale), a methodology and accompanying framework for building a large-scale URL parser differential dataset and applying it against opaque systems. TURLMOIL assembles 96 production parsers across 21 language ecosystems with harnesses to extract all 11 URL components. To reach the differentials that arise among delimiters, encodings, and structural boundaries, and to scale across heterogeneous runtimes without per-target instrumentation, TURLMOIL generates inputs from a probabilistic context-free grammar (PCFG) over the full URL structure and uses a feedback loop that re-weights the grammar at runtime toward rare, security-relevant disagreement. Finally, TURLMOIL uses the resulting dataset to fingerprint an opaque stack, identifying the parser at each layer from the outside and returning an input that distinguishes it from the one expected downstream.

We organize the study around four research questions:

- **RQ1**: How prevalent are URL parser differentials across the ecosystem?
- **RQ2**: What attacker techniques produce security-relevant URL parsing differentials?
- **RQ3**: How do attacker techniques interact with URL parser pairs that share standards?
- **RQ4**: Can an attacker identify URL parsers in a black-box stack from the outside and turn the result into a targeted exploit?



Figure 1: URL component diagram.

Standards do not eliminate URL parser differentials. TURLMOIL discovers that 99.96% of pairs diverge on at least one of the 11 URL components for some input, and 97.92% diverge on every component for some input. Disagreement persists even between parsers that claim the same standard and that ship in the same language, breaking common assumptions about both interoperability and security. We show that safe parsing requires the *same* parser, not a same-standard one.

We distill the discovered differentials into 21 techniques across 5 families, and our fingerprinting method identifies layered parsers from the outside in roughly  $\log_2 N$  probes per layer, followed by a dataset lookup. We therefore reduce black-box exploitation from an unbounded search to a bounded sequence of probes and a query. Applying this method, we uncover and responsibly disclose high-impact vulnerabilities in deployed systems, including an OAuth account takeover in MediaWiki [10], open redirects in Jenkins [11] and CTFd [12], and bypasses in Google Safe Browsing [13] and VirusTotal [14].

Our contributions are as follows:

- We present *TURLMOIL*, a PCFG-based, feedback-driven framework for large-scale differential testing of URL parsers. Our tool generates security-relevant URL inputs, extracts component-level parser outputs, and records concrete disagreements across heterogeneous parser implementations.
- We conduct the largest empirical study of URL parser differentials to date, covering 96 parsers across 21 language ecosystems and all 11 URL components. We release a queryable dataset that, for any diverging parser pair and component, returns a concrete distinguishing input and the corresponding parses.
- We show that neither shared standards nor shared implementation languages guarantee interoperable URL parsing. Our measurements include the first empirical evidence of intra-standard divergence, contradicting prior claims that same-standard parsers cluster safely.
- We introduce a black-box parser fingerprinting technique that identifies URL parsers in opaque systems using parser differentials, and we use it to uncover six responsibly disclosed vulnerabilities in widely deployed systems, spanning OAuth account takeover, open redirect, and security-gate fail-open bugs.

Overall, we build a 96-parser differential dataset, fingerprint black-box parsing stacks from the outside, and discover that shared standards do not prevent security-relevant divergence—yielding six disclosed vulnerabilities

in production systems. The source of TURLMOIL and associated analysis scripts can be found at <https://github.com/wspr-ncsu/TURLMOIL>. The full differential dataset produced by TURLMOIL can be found at <https://doi.org/10.7910/DVN/W32MSS>.

## 2. Background and Related Work

Modern web architectures often use Uniform Resource Locators (URLs) as security-critical identifiers, leveraging them to transmit credentials, establish identity, and orchestrate complex routing. Syntactically, URLs comprise explicit hierarchical components and sub-components including: scheme, authority (username, password, host, port), path, query, and fragment. These components are categorized in Figure 1.

**URL Standards:** Two specifications govern URL syntax. The IETF was the first to standardize the URL with RFC 1738 [15], which later evolved into RFC 3986 in 2005 [16]. The RFC standards did not achieve interoperability in practice, and themselves acknowledge grammatical ambiguities while leaving error handling to the implementer. As browsers accumulated parsing quirks to accommodate real-world input, browser vendors codified their behavior into the Web Hypertext Application Technology Working Group (WHATWG) URL Standard in 2012 [17], which accepts more input than the RFCs and defines its own semantics for delimiters, slashes, and Unicode that the RFCs do not share. The two standards split the implementation landscape. Backend frameworks and command-line tooling largely retained RFC semantics [18], [19], browsers and client-facing software adopted the WHATWG algorithm [17], and some libraries implement a hybrid of both. Because both standards leave certain edge cases underspecified, the gap between any two interpretations in a request path becomes an attack surface [2], [3], [4], [20], [21], [22], [23], [24], [25], [26]. Researchers long treated such inconsistencies as interoperability quirks [27] until practitioners weaponized them: Tsai’s *A New Era of SSRF* [1] defeated SSRF defenses with parser differentials, and follow-on work extended the technique to NGINX, the Spring Framework, and Ruby on Rails [7], [28]. URL parser differentials have since been exploited widely [4], [8], [22], [23], [29].

**Measuring the URL Parsing Ecosystem:** Two manual studies anchor the measurement literature. Moshe et al. [2] audited 16 parser implementations and distilled 5 attack classes. Ajmani et al. [5] cross-tested 8 parsers across 5 language ecosystems by extracting 1,445 URLs from the parsers’ own test suites, reported 4,262 inconsistencies

TABLE 1: TURLMOIL vs prior work.

| Property                         | TURLMOIL<br>(This work) | Reynolds<br>et al. [3] | Kallus<br>et al. [6] | Ajmani<br>et al. [5] | Moshe<br>et al. [2] |
|----------------------------------|-------------------------|------------------------|----------------------|----------------------|---------------------|
| Automated                        | ✓                       | ✓                      | ✓                    | ✗                    | ✗                   |
| Parser Coverage                  | 96                      | 15                     | 6                    | 8                    | 16                  |
| Language Coverage                | 21                      | 8                      | 1                    | 5                    | 10                  |
| Scheme                           | ✓                       | ✗                      | ✓                    | ✓                    | ✓                   |
| Authority                        | ✓                       | ✗                      | ✗                    | ✗                    | ○                   |
| Userinfo                         | ✓                       | ✗                      | ✗                    | ✗                    | ✗                   |
| Username                         | ✓                       | ✗                      | ✗                    | ✗                    | ✗                   |
| Password                         | ✓                       | ✗                      | ✗                    | ✗                    | ✗                   |
| Host                             | ✓                       | ✗                      | ✓                    | ✓                    | ✓                   |
| Port                             | ✓                       | ✗                      | ✓                    | ✓                    | ○                   |
| Path                             | ✓                       | ✗                      | ✓                    | ✓                    | ○                   |
| Query                            | ✓                       | ✓                      | ✗                    | ✗                    | ○                   |
| Query Dictionary                 | ✓                       | ✗                      | ✗                    | ✗                    | ✗                   |
| Fragment                         | ✓                       | ✗                      | ✗                    | ✗                    | ○                   |
| URLs Tested                      | 512,424                 | 98,445                 | N/A                  | 40                   | N/A                 |
| Testing Methodology              | PCFG                    | Mutation               | Mutation             | Tests                | Manual              |
| Intra-spec Divergence Measured   | ✓                       | ○                      | ○                    | ✓                    | ✗                   |
| Black-box Parser Fingerprinting  | ✓                       | ✗                      | ✗                    | ✗                    | ✗                   |
| Code Coverage Feedback           | ✗                       | ✗                      | ✓                    | ✗                    | ✗                   |
| Open Source Dataset              | ✓                       | ✗                      | ✗                    | ✓                    | ○                   |
| Applied to Black-box Deployments | ✓                       | ✓                      | ✗                    | ✗                    | ✗                   |

✓ yes, ○ partial, ✗ no, N/A not applicable.

prioritized by same-origin-policy relevance, and grouped them into 7 categories spanning hostname, scheme, control-character, backslash, slash, path, and encoding confusion. Both demonstrate the two-parser composition we define in Section 3, but manual auditing bound their scale.

Automated studies address some of the limitations posed by manual testing. Kallus et al. [6] applied grammar-aware, coverage-guided differential fuzzing to 6 Python parsers, seeding mutation from 200 WebKit test URLs and flagging both exit-status and output differentials; however, coverage guidance requires per-target instrumentation that does not generalize across heterogeneous runtimes and compiled binaries. The largest prior automated study is Reynolds et al. [3], who evaluated 15 parsers (2 of them custom specification-reference implementations the authors wrote for RFC 3986 and WHATWG) on 98,445 inputs produced by inserting one to four bytes into the hostname of a single valid URL. Reynolds compared parsers both against the two references and pairwise across all sampled pairs, and reported two high-agreement clusters: a strict-RFC group and a more permissive group that accepts invalid characters without erroring. From the disagreements, Reynolds distilled seven hostname pitfalls and evaded the Google Safe Browsing and VirusTotal classifiers.

Another set of studies examine the two-parser pattern inside a single application. Zhang et al. [26] show that an inconsistency between a Java framework’s routing normalization and its URL-based authentication check allows a crafted path (for example, `/login/./admin`) to reach a protected handler. Their static analyzer found 94 such bypasses across 72 applications, confirming that the two-parser configuration is real and high-impact. Zhang et al. examine only one language ecosystem, limited to the path component, through white-box static analysis of source rather than measurement of parser disagreement. Wang et al. [30] similarly automate SSRF discovery in PHP applications, but neither characterizes the parsing ecosystem itself. Balduzzi et al. [31] automate the discovery of HTTP parameter pollution across live sites, but measure disagreement over duplicate query parameters rather than over the full URL structure.

**Limitations of Prior Approaches:** Prior measurements are

bounded along four axes: scale, coverage, input space, and applicability. Table 1 summarizes which axis each study addresses.

*Scale:* The corpora are small relative to the ecosystem. Reynolds et al. [3] evaluate 15 parsers, while the manual studies of Moshe et al. (16 [2]) and Ajmani et al. (8 [5]) are bounded by human effort. Kallus et al. [6] evaluate 6 parsers confined to a single language.

*Coverage:* No prior work systematically measures and classifies disagreement across all components of the URL. Reynolds et al. [3] report agreement on five segments but only mutate and systematize the hostname. Kallus et al. [6] surface scheme, host, port, and path bugs but present no results for userinfo, username, password, query, or fragment. Ajmani et al. [5] and Moshe et al. [2] inspect varied components, but opportunistically rather than at scale.

*Input Space:* How a study generates inputs bounds the differentials it can surface. Reynolds et al. [3] substitute one to four hostname characters with random code points, which exposes single-character discrepancies but underapproximates the delimiter, encoding, and structural interaction bugs behind many severe vulnerabilities [7], [8], [9]. Both Kallus et al. [6] and Ajmani et al. [5] draw inputs from the parsers’ existing test suites, bounding their input space by the cases maintainers already thought to test. Moshe et al. [2] used 3 experts to manually test for parser differentials, bounding their corpus.

*Applicability:* Prior work stops at characterization and cannot be pointed at a real, opaque target. Two capabilities are missing. First, none identify an unknown parser stack from the outside: Reynolds’ threat model explicitly presumes an adversary who can already anticipate the victim’s parsers, and the studies that reached live services [2], [3] did so against parsing behavior an expert assumed or inferred rather than deterministically fingerprinted. Second, none provide a queryable artifact that returns a concrete distinguishing input for an *identified* pair of parsers and components. Prior work’s output is a taxonomy, a category label, or a bug report. Ajmani et al. [5] release a framework and a set of inconsistent URLs, but it requires the analyst to supply and run the parsers. Zhang et al. [26] reach real targets, but only through white-box static analysis of application source. Consequently, turning any prior catalog into an exploit against a specific deployment still requires the attacker to already know the stack.

These limitations leave a central question open. Reynolds et al. [3] report that same-standard implementations form high-agreement clusters and read the residual differences as minor variations in strictness. That observation was drawn from agreement measured on the hostname, under single-character perturbation, with an equality relation that abstracts away case, delimiter inclusion, and error type. Additionally, only one of the 13 production parsers Reynolds tested follows WHATWG. By definition, one parser cannot be used to assess intra-standard divergence. Whether the reported interoperability reflects the ecosystem or the surface on which it was measured therefore remains untested across the rest of the URL and under inputs that exercise delimiter, encoding,

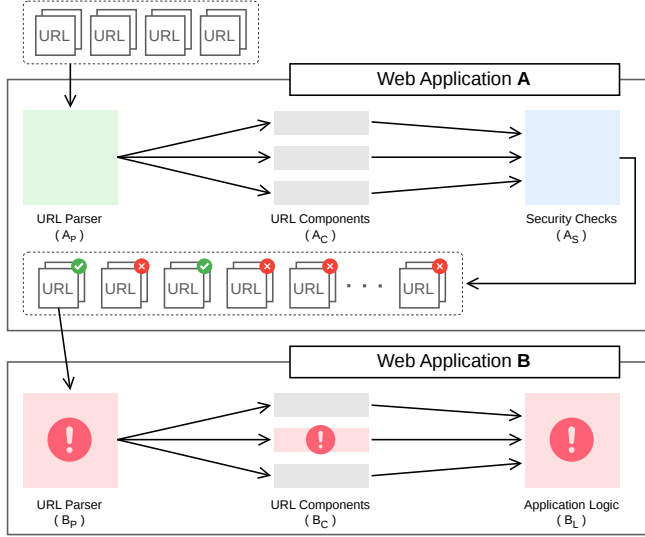


Figure 2: Threat model.

and structural interactions. We answer this question directly in Section 5.

To overcome these limitations, we develop a methodology for constructing a large-scale URL parser differential dataset, realized in a PCFG-based, feedback-driven fuzzer, that addresses all four axes. For **scale**, we evaluate 96 parsers across 21 language ecosystems against 512,424 generated URLs (Section 4). For **coverage**, we measure differentials across all components of the URL as shown in Figure 1 (Section 5). For **input space**, the grammar generates over the full URL structure, sampling the delimiter, encoding, and structural interactions that fixed mutations and test-suite seeds do not reach (Section 4). For **applicability**, a black-box fingerprinting method identifies the parser at each layer of an opaque stack, and the dataset returns a concrete distinguishing input for any identified pair (Section 4.5).

### 3. Threat Model

We detail our threat model in Figure 2. We assume an unprivileged, remote attacker who can send arbitrary requests to the validating layer *A* and observe its responses, but cannot reach the downstream layer *B* directly. The attacker sees the downstream layer’s behavior only as it surfaces through *A*, whether as an error, a missing fetch, a redirect target, a link preview, or some other oracle. The attacker cannot read either layer’s source, does not know which parser libraries are deployed, holds no man-in-the-middle position, and cannot modify the servers or other users’ requests. We represent this threat model as two parsers arranged in sequence, each independently parsing the input URL.

**Architecture:** A web request is rarely parsed once. A user-supplied URL normally passes through a chain of independently implemented systems, each of which parses the URL with its own library and acts on its own interpretation. We abstract this chain into its essential pair (Figure 2): a

*validating* application *A* that parses an incoming URL with parser  $A_P$  into components  $A_C$  and applies a security check  $A_S$ . When the check passes, a downstream *acting* application *B* parses the same URL with parser  $B_P$  into components  $B_C$  and uses the result in application logic  $B_L$ , such as fetching a resource or issuing a redirect. Under this model, a bypass reduces to a pairwise differential: the attacker seeks an input URL for which  $A_S$  passes under  $A_C$  but  $B_L$  acts on a different interpretation  $B_C$ .

The two-parser configuration is not specific to any one system. An OAuth provider validates a redirect URI and then redirects the browser to it. A login flow validates a `next` parameter before redirecting. A webhook endpoint validates and stores a URL that a delivery worker later fetches. A reputation service such as Google Safe Browsing or VirusTotal classifies a URL that the user’s browser independently visits. In each case, two parsers consume the same URL, and the system’s security rests on their agreeing. The validating layer is trusted to guard the system precisely because it is assumed to interpret the URL the same way the acting layer will. That assumption is the attack surface, and it holds only if the two parsers never disagree on the part of the URL the check inspects.

The simplest and most common case is *transparent passing*, where the validating layer *A* forwards the URL untouched and the acting layer *B* parses exactly the string the attacker sent. OAuth, redirect, webhook, and reputation flows all behave this way. Some intermediaries instead *re-serialize*: a proxy, CDN, or firewall parses the URL, rewrites it into its own canonical form, and forwards that. The downstream layer then parses the rewritten string rather than the original, so the relevant disagreement is between the intermediary’s rewriting and the downstream parse. Our fuzzing methodology and our fingerprinting still apply, but the attacker must account for the rewriting rather than reading a single pairwise differential from the dataset. We therefore treat transparent passing as the primary case and re-serialization as a generalization. A related effect arises within a single parser whose own canonicalization is not idempotent, so that re-parsing its output yields different components. These self-differentials compound the cross-parser ones we study and can introduce new vulnerabilities. We leave self-differentials to future work.

**Exploitation:** To exploit the configuration, the attacker crafts a single URL that *A* reads one way and *B* reads another, so that  $B_L$  operates on a different, attacker-chosen target. The disagreement must fall on the component the check inspects and the downstream logic uses. Mounting this attack blind is hard, because the attacker does not know which parsers sit at either layer and useful disagreements are rare. Our work removes that obstacle. The attacker first fingerprints the stack (Section 4.5), identifying each layer’s parser from the outside by observing how the system responds to chosen probes, and then queries our dataset for a URL on which the identified pair diverges. Exploitation becomes a lookup rather than a search. The described fingerprinting method works only when the downstream layer’s behavior produces an observable signal. A layer that returns uniform responses and no side effect cannot be fingerprinted or attacked this

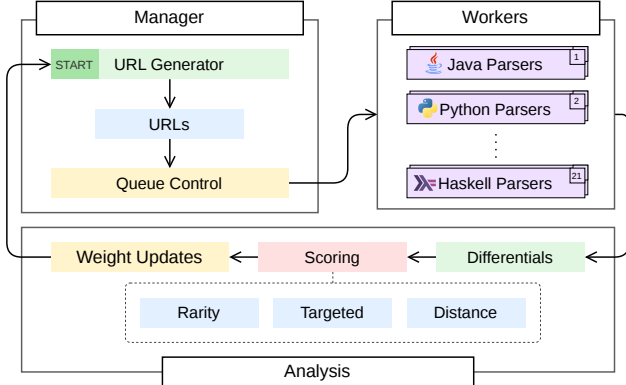


Figure 3: System diagram.

way.

In 2019, a researcher mounted a server-side request forgery against Dropbox [9] by exploiting exactly the two-parser split detailed in Figure 2: the backend validated a URL as pointing to an allowlisted Dropbox host, while the downstream cURL client resolved the same string to a loopback address and reached an internal service. The same check-then-act pattern underlies a broad class of deployed bugs whose consequences span internal-architecture disclosure, open redirect, SSRF, and unauthorized access. Section 6 examines several of these vulnerabilities in deployed systems.

## 4. TURLMOIL

TURLMOIL measures parser disagreement at scale, steers generation toward security-relevant disagreement, and records every differential in a queryable dataset. Three problems make this hard. The input space is large, and the security-relevant differentials within it are rare, so undirected generation spends most of its effort on inputs that every parser handles alike. The parsers span many languages and runtimes, so one comparison harness must drive code that shares no common interface. The volume of comparisons is large, because every input is parsed by every parser and compared component by component.

Figure 3 shows how TURLMOIL addresses these problems. A central manager generates candidate URLs and manages the task queue. Each candidate becomes a URL job, which the manager fans out into one library job per parser. Language workers, one per ecosystem, receive a library job, invoke the parser on the input URL, and return its parsed components. A database stores every parse. Analysis workers then compare the parses for each URL, flag the components on which any two parsers disagree, and score the URL by how useful its differentials are. Those scores update the generation weights, so the next round of generation favors the inputs that produced useful disagreement. Every layer is horizontally scalable. The number of generators, language workers, and analysis workers can be raised independently without changing the core logic.

### 4.1. Parser Selection

We began from the 25 most common programming languages in the Stack Overflow Developer Survey [32]. For each language ecosystem, we surveyed open-source URL parsing implementations, using GitHub stars, package download counts, and inclusion in the standard library as signals of adoption. We excluded closed-source parsers, whose behavior we could not reliably extract. Two authors with web security experience then reviewed the remaining code and removed parsers that were thin wrappers around another implementation. For each remaining parser, we also recorded the URL specification it claims to follow, using the parser documentation as the source of these labels. We use these labels later to partition parser pairs by claimed standard and measure divergence within each group. The result is 96 parsers across 21 language ecosystems, the most comprehensive cross-language corpus assembled for URL differential analysis to date. Appendix C lists every parser with its claimed specification, language, version, and usage metrics.

A harness around each parser extracts the 11 URL components defined in Section 2. Not every library exposes every component. When a library provides no method to extract a field, the harness marks that field `EXCLUDE` and omits it from comparison. When a component field is marked as `EXCLUDE`, the *library client* is forced to parse this field independently if they wish to use its value. As we demonstrate in Section 5, parsing is very difficult to do correctly, and ad hoc parsers implemented by library clients are almost certain to have differentials with other established parsing libraries. As such, our exclusion of missing components ensures our study represents an accurate lower bound on the total number of differentials present in the ecosystem.

Parsers embedded in compiled tools required extra work. For tools such as cURL, we wrapped the binary and invoked it with flags that surface only its URL parsing. For other compiled parsers, we recompiled the tool with an added interface that exposes the parsed components. For the remaining libraries, the tool harness calls the public API directly. To amortize interpreter and runtime startup, each language ecosystem runs in a single container that hosts all of its parsers, which also pins versions and isolates dependencies per ecosystem. When a container receives a library job, a lightweight process invokes the parser, and the parsed components are serialized to the database.

### 4.2. URL Generation

TURLMOIL generates inputs from a probabilistic context-free grammar (PCFG) built with the `nltk` library [33]. The grammar abstracts the structure of a URL and produces both valid and invalid inputs. It represents each of the 11 components separately, generates them independently, and concatenates them into a complete URL. Every component has two weighted productions: a valid path and an invalid path. The valid path emits characters

```

FRAGMENT -> VALID_FRAG [p_valid]
          | INVALID_FRAG [p_invalid]
          | NULL [p_null]

VALID_FRAG -> FCHAR FTAIL [1,0]
FTAIL -> FCHAR FTAIL [recurse]
        | NULL [stop]

INVALID_FRAG -> ICHAR ITAIL [1,0]
ITAIL -> ICHAR ITAIL [recurse]
        | ASCII_WS [ws]
        | NULL [stop]

FCHAR -> ASCII_LETTERS [w1] | ASCII_DIGITS [w2]
        | ASCII_SPECIAL [w3]

ICHAR -> ASCII_CONTROL [w1]
        | UNICODE_VALID [w2]
        | UNICODE_DISALLOWED [w3]
        | UNICODE_UNASSIGNED [w4]
        | UNICODE_RESERVED [w5]

```

Figure 4: PCFG for the URL fragment component. Bracketed weights are updated at runtime by the feedback loop.

and structures the WHATWG standard permits. The invalid path inserts disallowed characters and breaks the structural constraints the standards impose, which probes how each parser behaves near its boundaries. This differs from prior generation strategies such as that of Reynolds et al. [3], whose generator inserts random code points into a valid URL.

Terminal values are drawn from 10 character groupings that together cover the input space: ASCII\_DIGITS, ASCII\_LETTERS, ASCII\_WHITESPACE, ASCII\_SPECIAL, ASCII\_CONTROL, HEXDIGIT, UNICODE\_VALID, UNICODE\_DISALLOWED, UNICODE\_UNASSIGNED, and UNICODE\_RESERVED. The Unicode groupings follow the IANA IDNA tables [34], and UNICODE\_DISALLOWED is taken per component from the WHATWG standard. Each grouping carries a weight, and a weighted right-recursive production selects terminals one character at a time, which allows variable-length output. Figure 4 gives the grammar for the fragment component as a representative example.

The generation weights are not fixed. TURLMOIL records the grammar path that produced each URL, and the feedback loop updates the weights on that path after the URL is scored. The generator reads the updated weights at the start of the next cycle and adjusts its sampling, so generation shifts over the run toward the regions of the grammar that produce useful differentials.

### 4.3. Feedback Loop

When a URL produces at least one differential, TURLMOIL scores each differing component from 0 to 100. The score combines three metrics that contribute equally.

*Levenshtein distance* measures how far apart the two parses are, which filters superficial differentials. Some parsers strip the fragment separator # while others keep it, for instance, and such cosmetic differences are unlikely to carry

security weight. Identical outputs score zero, and completely different outputs score the maximum.

*Differential rarity* keeps the fuzzer from fixating on a few permissive parsers. A lenient parser disagrees with nearly every other parser, so without correction the fuzzer would over-sample those pairs and miss disagreement between stricter libraries. TURLMOIL tracks how often each library pair and component produces a differential and scores rare pairs higher.

*Differential targetedness* rewards inputs that separate only a small subset of pairs. A highly targeted differential is more likely to reflect distinct parser behavior, making it more useful for secondary tasks such as fingerprinting. URLs generating differentials only affecting one parser pair score higher than URLs that affect a large number of parser pairs.

After scoring, TURLMOIL raises the weights on grammar paths that produced high-scoring differentials and lowers the weights on paths that produced low-scoring output. Later iterations therefore favor the productions associated with security-relevant disagreement.

### 4.4. Attack Technique Identification

The raw dataset (detailed in Section 5.1) records the parse each parser produced for each input URL. To distill security-relevant attack techniques, we index every differential by its (parser\_a, parser\_b, component) key, then query the index for the differentials on a given component between a given pair of parsers.

For each recurring behavior, we isolate the root cause by minimization. We take a complex URL that triggers the differential and minimize it until only the underlying disagreement remains. We then express the isolated root cause as a set of SQL filters over the dataset, remove every matching case from the set of unexplained differentials, and repeat. We continue until no further recurring root cause remains in the unexplained set.

The procedure produced 21 techniques, which we group into five families. (1) *Encoding* covers disagreement over how percent-encoded octets, encoded host labels, and non-ASCII characters are interpreted. (2) *Sanitization* covers whether invalid characters, control characters, whitespace, and null bytes are stripped, retained, normalized, or rejected. (3) *Normalization* covers whether different representations are treated as equivalent and when that equivalence is applied. (4) *Delimiter confusion* covers disagreement over which characters mark the boundaries between components. (5) *Structural* covers how repeated, empty, or degenerate inputs become application-facing data. Section 5 analyzes how each family distributes across parsers and components. The families are consistent with the differential classes reported by Reynolds et al. [3], Moshe et al. [2], and Ajmani et al. [5], and expose additional subtechniques present in our data.

### 4.5. Fingerprinting URL Parsers

TURLMOIL enables fingerprinting the URL parsing stack of a black-box system. Given a URL input field

exposed by a remote service, an attacker can identify the specific parser running behind it using a small number of carefully chosen inputs. The technique relies on the same disagreements we measure throughout the paper. If parser *A* and parser *B* behave differently on some input, then the response of the black-box service to that input reveals which of the two it runs.

**Algorithm:** The core algorithm is an adaptive binary search over the candidate parser set, keyed on error behavior. Given  $N$  candidates, the attacker selects a probe URL that splits the candidate set as evenly as possible. The selected URL must error on roughly half the candidates and parse successfully on the other half. The response from the black-box service eliminates approximately half the candidates in a single probe. The attacker selects the next probe based on the remaining set and repeats. When a clean error-based split exists at every step,  $\log_2 N$  probes are sufficient to uniquely identify the parser. In practice, the number of probes required is variable because usable differentials often split the candidate set into uneven groups.

The primary signal is parser errors. An error response narrows the candidate set to parsers that reject the input, while a successful response narrows it to those that accept. Additionally, when the service exposes the parsed value, the output itself becomes a discriminator. This visibility is not required, but it shortens the procedure when available.

Many production systems layer several parsers, with one parser validating input before another acts on it. The procedure above identifies only the outermost parser, whose error behavior is surfaced to the caller. Errors are especially useful for peeling off subsequent layers. Once the outer parser is identified, the attacker restricts probes to inputs that parser is known to *accept*. Any error observed in response then originates from a parser further down the stack, and the same binary search applies against the remaining candidates. The procedure repeats one layer at a time, for as many layers as the service contains.

A successful fingerprint reduces the attack surface to a known parser, against which the dataset records every differential the parser exhibits with every other parser in the corpus. An attacker who has identified one layer of the stack can then craft an input that diverges from any specific candidate parser expected downstream, turning a generic class of parser confusion attacks into a targeted exploit.

**Constraints:** The target system must surface differential behavior to the caller, whether through an HTTP error, a missing fetch, a visible redirect target, or a preview. Systems that normalize error responses, return generic failures, or run their parsing behind a canonicalization layer cannot be fingerprinted by this method. Parsers that are thin wrappers around a shared engine may not be distinguishable from one another. Parsers outside our corpus can be narrowed to a residual candidate set but not identified.

## 5. Evaluation

This section answers the following research questions.

- RQ1** *How prevalent are URL parser differentials across the ecosystem?*
- RQ2** *What attacker techniques produce security-relevant URL parsing differentials?*
- RQ3** *How do attacker techniques interact with URL parser pairs that share standards?*
- RQ4** *Can an attacker identify URL parsers in a black-box stack from the outside and turn the result into a targeted exploit?*

### 5.1. Experimental Setup

**Implementation:** TURLMOIL runs as a fully containerized, distributed, and horizontally scalable system. TURLMOIL uses Celery [35], an open-source distributed asynchronous task queue written in Python, to implement our task queue. Celery handles task distribution, status tracking, retries, and result collection. All workers are implemented as Kubernetes pods, which register their task types with the manager at startup. Results are written to a PostgreSQL database backed by a Kubernetes `PersistentVolumeClaim`, which preserves them across restarts. The cluster runs on five nodes, which together provide roughly 416 GB of memory and 326 CPU cores. Experiments took 42 days, producing an 822 GB dataset.

**Differential Dataset:** TURLMOIL generated 512,424 input URLs and parsed each with all 96 parsers, recording the 11 extracted components per parse. Comparing these outputs across every distinct pair of parsers yields 4,560 parser pairings, across which we recorded 5,011,835,830 individual component-level differentials. Every differential is stored as a tuple keyed by `(parser_a, parser_b, component)` together with the input URL and both parses.

The dataset is a reusable artifact whose structure directly answers two questions. First, for any pair of parsers in the corpus, it reports whether they diverge and on which components. This is the question that matters when composing parsers in a request path, and it is one that conformance to a specification cannot answer. Second, every differential is stored with the input URL and both parses. This ensures that the dataset provides a concrete proof-of-concept input for any diverging pair and component.

### 5.2. RQ1: Prevalence of URL Parser Differentials

Across the corpus, parser disagreement is the norm. 99.96% of the 4,560 distinct parser pairs in our corpus diverge on at least one of the 11 URL components, and 97.92% diverge on every parsed component of a URL. Figure 5 shows the per-pair breakdown.

Only four pairs agree on every input we tested. On closer inspection, these parser pairs are not independent, but thin wrappers around a shared parser. We attempted to filter for this case in our parser selection, but some parsers did not clearly advertise this relationship. Though we did not intentionally include thin wrappers, TURLMOIL discovered real differentials between the wrappers and the core library, demonstrating the depth of probing reached.

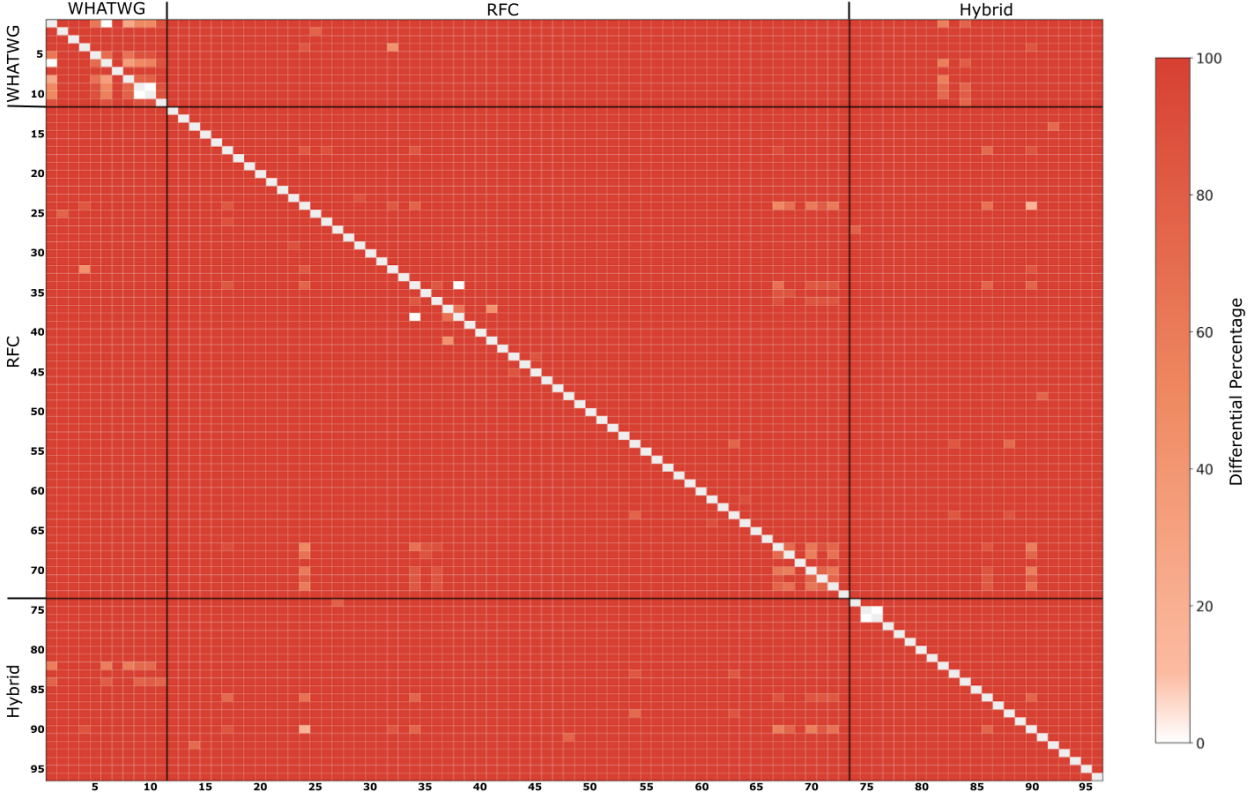


Figure 5: Percentage of differing outputs for each parser pair, over the components both parsers extract. Only inputs that both parsers parsed successfully are considered.

Language bindings enable different libraries to invoke the same underlying implementation, or fixed mode variants of the same library. We do not interpret these pairs as evidence of safe composition, only as evidence that a shared implementation cannot disagree with itself by construction.

**Standard Compliance:** A natural defense against URL parser differentials is to standardize on a single specification. If every parser in a request path claims conformance to the same standard, a developer might reasonably expect those parsers to interpret a URL identically. Prior work reinforces this intuition: Reynolds et al. [3] report that same-standard implementations form high-agreement clusters and interpret the remaining differences as minor variations in strictness.

Our results show that this intuition is unsafe. In our corpus, 96.36% of parser pairs in which both parsers claim the WHATWG URL Standard diverge on at least one URL component, and 99.95% of pairs in which both parsers claim RFC also diverge. Figure 5 provides an overview of parser differentials by specification.

This contradicts the idea that same-standard composition is a meaningful security boundary. The disagreement is not simply a matter of obviously faulty implementations failing to follow the rules. Both the WHATWG URL Standard and RFC 3986 leave behavior undefined or implementation-dependent in security-relevant edge cases. As a result, two

parsers can each plausibly claim conformance while still resolving the same input differently wherever the standard is silent. A developer composing same-standard parsers is therefore not composing one behavior; they are composing multiple behaviors that only agree on the portion of the input space the specification fully determines.

The practical implication is that “use a standard parser” is insufficient as a defense against parser differentials. Safe composition requires either using the same parser implementation throughout the request path, or enforcing an explicit canonicalization and validation contract whose behavior is tested across every parser that will consume the URL. Merely selecting parsers that claim the same standard does not provide this guarantee.

The clearest exception in our corpus is the pair `rust-url` [36] and `cpp-ada-url` [37], the parsers shipped in Firefox and Node.js, respectively. Their agreement should not be read as evidence that independent same-standard parsers naturally converge. Rather, it is the product of sustained manual testing, cross-implementation review, and deliberate coordination by the maintainers of both projects [38]. This is the level of engineering effort required to make independently implemented parsers behave interoperably in practice.

**Takeaway:** *Parser disagreement is pervasive across the ecosystem. Standards do not eliminate URL parser differentials. Even parsers that claim the same specification frequently diverge on security-relevant components, contradicting the assumption that same-standard parsers compose safely.*

### 5.3. RQ2: Attack Techniques

We distill the 512,424 URLs generated by TURLMOIL into 21 broad techniques of security-relevant parsing differentials, and test each bucket against the 96 parsers in our dataset. Figure 7 shows the fraction of parser pairs that disagree for each technique and URL component. These techniques represent five broad families of attack techniques, which we describe in detail.

*Delimiter confusion* produces the largest differentials in authority fields. The family captures disagreement over which characters define the boundaries between URL components. While several families produce high disagreement across the aggregate heatmap, delimiter confusion is distinctive because its largest effects occur in authority-bearing fields: duplicate delimiter confusion reaches 95% pairwise disagreement on the host, and also produces high disagreement on authority, userinfo, username, and password.

When parsers disagree on delimiter placement, they assign different substrings to userinfo and host, and therefore differently answer the question of who is being authenticated and where the request is going. The family also includes ambiguous separators such as `;` and `,`, which target query parsing, and square brackets, which cause significant differentials across all components when in the host.

For example, on inputs of the form `http://a@b@c-@sub.example.com/`: 41 parsers apply the last-`@` rule and return host `sub.example.com`, 16 apply the first-`@` rule and return `b@c@sub.example.com`, and 3 return `b`. If a validator interprets `http://allowed.-example.com@attacker.tld/` using the first `@` while a client uses the last `@`, the two disagree on the destination host.

*Structural differentials* are narrow and typically target queries. The family captures disagreement over how parsers assemble ambiguous, repeated, or degenerate inputs into application-facing data structures. This family includes duplicate query keys, empty query keys or values, key mangling, and empty components. Unlike delimiter confusion or normalization, structural differentials are concentrated primarily in the query, the field that carries application data rather than destination or trust information.

For example, on `?a[]=1&a[]=2&a[]=3`, parsers split between array construction and last-value-wins behavior. On inputs of the form `?a%20b=1&a+b=2&a b=3`, 22 parsers treat the three keys as distinct while 20 collapse `+` and `%20` to the same key under form-encoding rules. Across the heatmap, this family is concentrated in the query columns. Key mangling disagrees on the raw query for 62% of parser pairs and on the query dictionary for 40%, while empty key/value dropping reaches 47% on the query dictionary. This

makes the surface more contained than authority-oriented families. Structural differentials are still security-relevant, but primarily affect what data the application sees, not which principal or network endpoint the application trusts.

*Encoding* affects all components because it operates on the byte or character stream before, or independently of, the final component interpretation. The family captures disagreement over how non-ASCII characters, encoded host labels, and percent-encoded octets are interpreted. For example, on a mixed Unicode/Punycode host such as `Bücher.xn--bcher-kva.com`, 28 parsers preserve the literal label, 16 re-encode it to the double-Punycode `xn--bcher-kva.xn--bcher-kva.com`, 12 case-fold only, and the rest consider the URL invalid. Similarly, for percent-encoded UTF-8 hostnames such as `sub%C3%B6example.c%C3%B6m`, 37 parsers keep the non-resolvable encoded form, 14 emit Punycode, and 11 decode to the literal IDN.

*Sanitization* similarly affects all components and captures parser disagreement over whether invalid Unicode, control characters, whitespace, and null bytes are stripped, retained, normalized, or rejected. For example, on a control-character input such as `http://us\ner:...@sub.\nexample.com/` with an embedded newline, parsers split across several behaviors: 18 retain the raw newline in the host string, 18 strip it to `sub.example.com`, and nine treat it as an authority terminator and return host `us`. This creates both a host differential and a potential injection primitive if the parsed value is later emitted into a request line, `Host` header, log entry, or redirect target. This is the mechanism behind the tab-stripping open redirects discussed in our case studies, where sanitization changes the value that the downstream redirect logic receives.

*Normalization* can also affect all components of a parsed URL. The family captures disagreement over whether different character representations should be treated as equivalent, and when that equivalence is applied during parsing. Unlike encoding and sanitization, normalization is often rule-specific, which lets an attacker aim it at a particular security boundary. Dot-segment normalization, for example, confuses parsers on inputs like `/a/%2E%2E/b/%2E/c`: 50 parsers leave the encoded dots opaque as `/a/%2E%2E/b/%2E/c`, 20 decode them without resolving the resulting segments as `/a/./b/./c`, and 16 both decode and resolve the path to `/b/c`. A path-prefix check using the first behavior and a file server using the third behavior can thus authorize and serve different resources for the same request.

URL encode/normalize ordering is another example: whether a parser decodes before or after splitting determines whether encoded delimiters remain data or become URL syntax. In the heatmap, encode/normalize ordering produces 84% disagreement on the host and 82% on the password. IP normalization similarly concentrates on the host, with 70% disagreement as some parsers canonicalize octal, hexadecimal, or integer IP forms to addresses such as `127.0.0.1`, while others pass them through. Slash normalization affects whether backslashes and slash-like characters are treated as

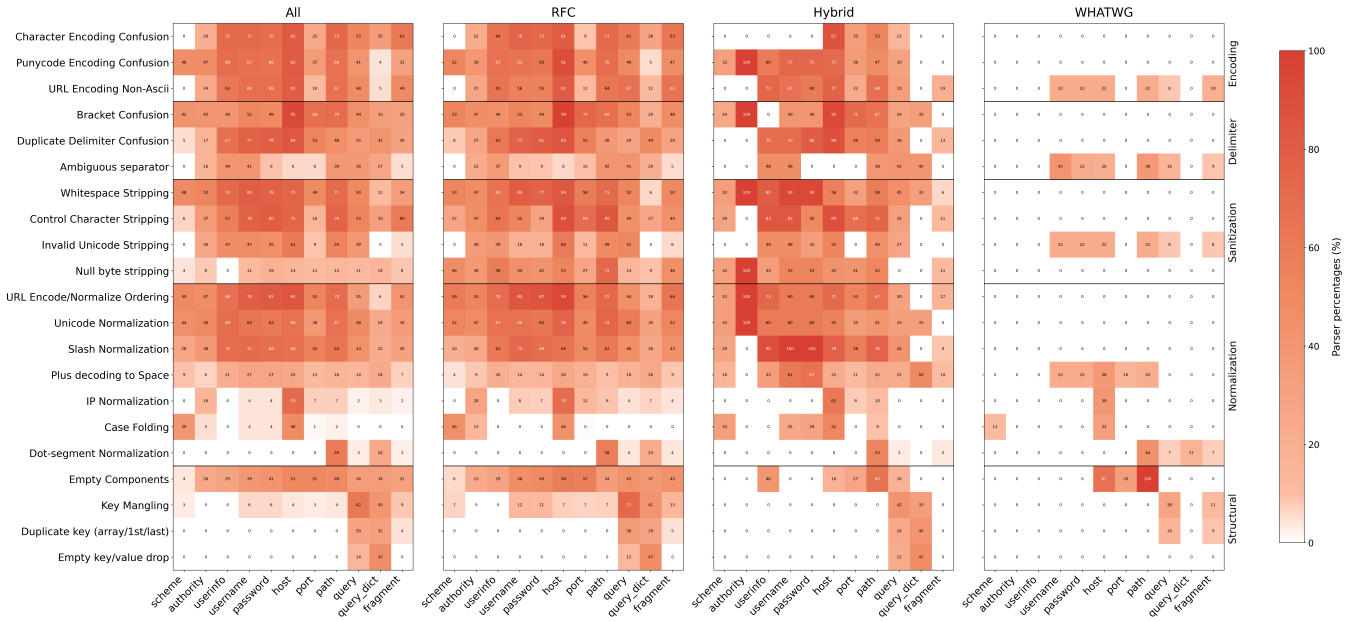


Figure 6: Heatmap of parser differential techniques across all parsers, RFC-based parsers, WHATWG-based parsers, and hybrid parsers.

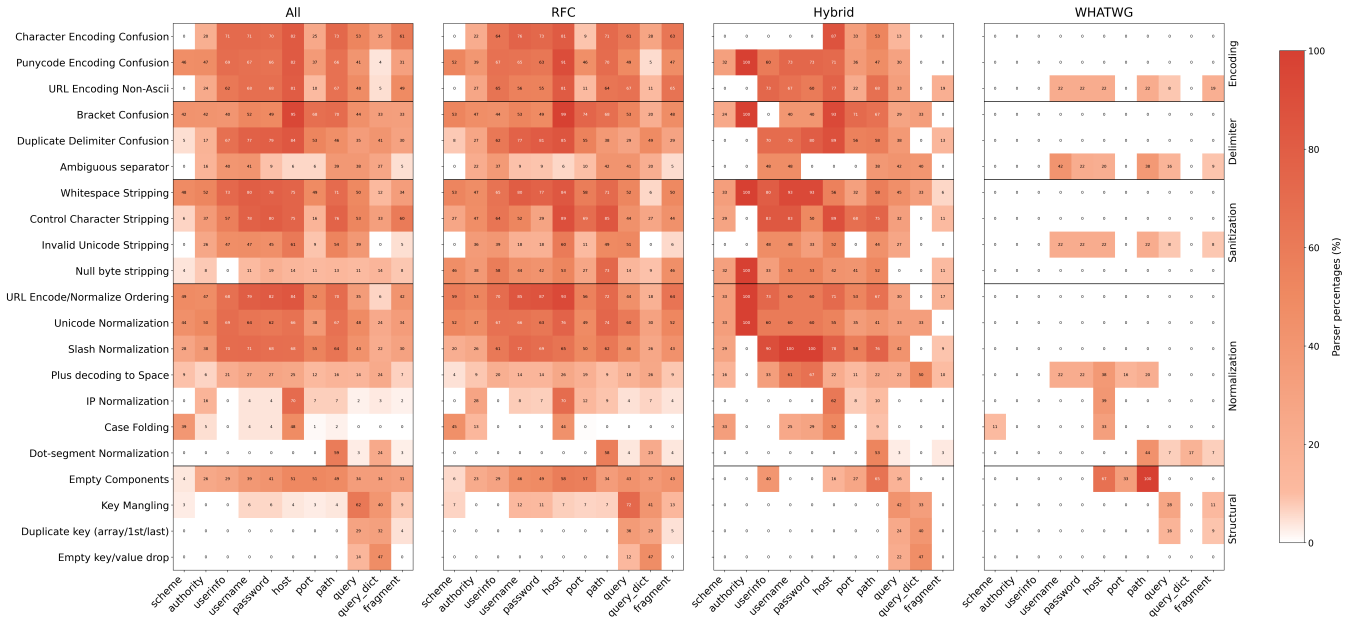


Figure 7: Heatmap of parser differential techniques across all parsers, RFC-based parsers, WHATWG-based parsers, and hybrid parsers.

path separators or authority delimiters, and allows an attacker to confuse parsers on where the host is.

**Takeaway:** *We identify five broad families of attack techniques that produce security-relevant URL parser differentials. Delimiter confusion disproportionately disrupts authority boundaries, while structural differentials concentrate in query parsing. Encoding, sanitization, and normalization induce disagreement across many components and can therefore be aimed at whichever component a security check relies on.*

#### 5.4. RQ3: Techniques Across Specifications

We next examine how these technique families interact with parser specification targets. The results show that specification choice changes where disagreement appears, but does not eliminate disagreement altogether. In particular, RFC-style parsers diverge broadly, hybrid parsers relocate disagreement into the most security-sensitive components, and WHATWG parsers are the most consistent but still retain disagreement on degenerate structural inputs.

*RFC-style parsers diverge across all families.* Parsers aligned with RFC-style URL handling show disagreement across every technique family: normalization, encoding, delimiter confusion, sanitization, and structural handling all produce observable splits. This suggests that RFC conformance alone is not a strong predictor of uniform behavior in deployed parsers. Parsers take significant freedom in error handling, normalization order, invalid-input recovery, and component representation, so implementations can still disagree on security-relevant inputs. For a security check, this means that choosing an RFC-style parser does not guarantee that a downstream client, browser, or framework will interpret the same URL in the same way.

*Hybrid parsers relocate disagreement rather than removing it.* Hybrid parsers, that is, parsers that do not conform to a specific standard reduce disagreement in some application-data fields, especially on the scheme, query, and fragment side. This cleanup comes with more disagreement in userinfo, authority, and credential-bearing components. Across encoding, normalization, and sanitization, the hybrid group collapses into near-total or total disagreement across multiple techniques in the authority fields. The result is thus not a uniform improvement over RFC behavior. Rather, disagreement is relocated from lower-risk fields into the fields most often used by security checks.

This pattern is especially concerning because these are the components that determine callback validation, SSRF filtering, credential extraction, host allowlisting, and origin-like trust decisions. Hybrid behavior also exhibits a slash-normalization spike consistent with protocol relativity and backslash handling: a parser may treat slash-like characters as ordinary path data, while another treats them as syntax that changes the authority boundary. Thus, the hybrid class diverges maximally in precisely the place where disagreement is most dangerous. It appears cleaner at the edge of the URL, but less reliable at the trust boundary.

TABLE 2: Fingerprinting probe sequence applied to Slack’s link preview pipeline to identify `libcurl`.

| Probe                                                    | Fetched | Remaining |
|----------------------------------------------------------|---------|-----------|
| Vertical tab ( <code>\x0B</code> )                       | No      | 73/96     |
| Invalid bytes ( <code>\xFF\xFF</code> )                  | Yes     | 43/96     |
| Special Unicode ( <code>U+FF61</code> )                  | No      | 38/96     |
| Backslash in path ( <code>\</code> )                     | Yes     | 29/96     |
| Dot segments + escaped bytes ( <code>/a\xffb/./</code> ) | Yes     | 3/96      |
| Special scheme ( <code>ws://</code> )                    | No      | 1/96      |

*WHATWG parsers are the cleanest, but still have structural disagreements.* WHATWG-oriented parsers show the strongest convergence. Most families exhibit near-total agreement, indicating that the algorithmic parsing model substantially reduces the scope of differences compared with RFC-style parsing. However, the remaining disagreement is not zero. The largest residue appears in empty-component handling, including complete disagreement on the path and substantial disagreement on the host. This suggests that the disagreements that survive are not simply implementation sloppiness, but places where malformed inputs, empty fields, or representation choices remain underspecified or difficult to align in practice.

This result also echoes the observation from Section 5.2: even among parsers that aim at the same modern specification, agreement is hard-won rather than automatic. The relative consistency of parsers such as `rust-url` [36] and `cpp-ada-url` [37] comes from closely following the WHATWG algorithm, not from URL parsing being intrinsically unambiguous. WHATWG therefore provides the strongest foundation for cross-parser consistency, but security-sensitive systems still cannot assume that all parsers claiming modern behavior agree on empty-component inputs.

**Takeaway:** *No specification family eliminates security-relevant parser differentials. RFC-style parsers diverge broadly, hybrid parsers concentrate disagreement near authority and userinfo boundaries, and WHATWG parsers converge the most but disagree on malformed or empty cases.*

#### 5.5. RQ4: Fingerprinting Using Parser Differentials

Our dataset establishes the precondition for parser fingerprinting as every recorded parser disagreement is also a distinguishing probe. Across many such probes, each parser exposes an observable behavioral signature through accept/reject behavior or through downstream effects such as redirects, fetches, and previews. Thus, the dataset shows that URL parsers are fingerprintable in principle.

The fingerprinting algorithm, however, must be evaluated on black-box systems because that is the setting it targets. In real deployments, the parser identity is unknown, and application logic may transform or hide parser behavior. Testing only against parsers in our harness would show that the dataset contains distinguishing inputs, but not that those inputs remain useful when mediated by real services.

To evaluate whether parser differentials can fingerprint real systems, we tested our technique against 4 black-box URL-processing systems, including messaging applications and malware-detection platforms. Our probes narrowed these systems to a small set of candidate backend parsers, and revealed one additional system whose behavior did not match any parser in our corpus. We now walk through these cases in detail.

**Messaging platforms:** We applied the technique to Slack [39], Discord [40], and Microsoft Teams [41], communication platforms that fetch URLs server-side to generate link previews. For all of them we were able to narrow it down to a single parser. For example, a user sends a message containing a URL; Slack issues a GET request to the link from its own servers. Using 6 manually crafted probes against this pipeline as shown in Table 2, we identified the backend parser as `libcurl` [42]. The probes for Discord and Microsoft Teams are listed in Appendix A. The work was conducted within the published scope of Slack’s bug bounty program.

**Google Safe Browsing:** We applied the fingerprinting technique to Safe Browsing [13] and discovered that the backend URL parser was not cataloged in our dataset. While fingerprinting, we were able to detect that the parser implemented the WHATWG spec. Combining these two facts, we narrowed the candidate parsers down to GURL, a URL parser tightly coupled with the Chromium browser. Additionally, we applied case folding and Unicode techniques, to identify that Safe Browsing uses an older revision of Google’s GURL library, distinct from the version shipped in Chrome. The version skew between the analyzer’s parser and the browser’s parser is itself the attack surface. This older revision of GURL is not in our corpus because the Chrome build is hard to isolate, but we were still able to detect it using the fingerprinting techniques described in Section 4.5.

The two GURL revisions diverge over the Greek lowercase final sigma ( $\Upsilon+03C2$ ,  $\varsigma$ ). The older revision used by Safe Browsing applies full Unicode case folding to host labels, normalizing  $\varsigma$  to the standard lowercase sigma  $\sigma$  ( $\Upsilon+03C3$ ), while the browser preserves  $\varsigma$ . Therefore, the two parsers disagree on the canonical hostname for any URL whose host contains  $\varsigma$ . Safe Browsing checks the reputation of  $\sigma.example$ , but the browser navigates to  $\varsigma.example$ . As a result, no domain or subdomain containing  $\varsigma$  can be flagged by Safe Browsing as malicious, because the host being checked ( $\sigma.example$ ) is not the host the browser will reach ( $\varsigma.example$ ).

## 6. Exploiting Parser Differentials

To demonstrate the real-world security impact of our findings, we used the dataset to identify vulnerabilities across open-source software and closed-source platforms. For both closed and open-source systems, we used the fingerprinting technique from Section 4.5 to identify the parsers in each system, then queried the dataset for a distinguishing input. Overall, we found six vulnerabilities, five of which we present in detail.

### 6.1. OAuth Bypass

OAuth callback validation is a natural target for parser differentials because the callback URL determines where authorization credentials are delivered. OAuth providers commonly validate a callback URL before redirecting the user’s browser to it with temporary credentials attached. If the provider-side parser and the browser disagree on the callback’s authority, the provider may validate the URL as belonging to the legitimate application while the browser delivers the credentials to an attacker-controlled origin.

**MediaWiki/Wikipedia:** We identified a vulnerability in the OAuth implementation of MediaWiki [10], the software powering Wikipedia and other Wikimedia projects (over 24 billion monthly page views) [43]. The vulnerability allows an attacker to take over user accounts in affected third-party applications. If the victim is an appropriately privileged administrator, the attacker can further escalate to cross-site scripting by phishing and abusing their ability to modify technical configuration rendered to any reader [44].

MediaWiki allows third-party applications to authenticate users through a custom OAuth 1.0a flow [45]. During this flow, the application may supply a post-authorization callback URL. To prevent credential leakage, MediaWiki validates this callback against an out-of-band callback prefix registered by the third-party application. This validation is intended to ensure that OAuth credentials are only returned to callbacks controlled by the registered application. However, we found that this guarantee can be broken when a particular library maintained by Wikimedia is used to build third-party apps. The library includes a login-helper functionality that forwards a user-controlled callback parameter into the OAuth flow. As a result, an attacker can cause a third-party application to initiate a legitimate OAuth flow while supplying an attacker-chosen callback URL to MediaWiki’s OAuth provider.

In isolation, the user-controlled callback would not be a problem. However, MediaWiki’s OAuth provider and the browser disagree on how to interpret backslashes in authority-bearing URLs. In particular, the provider-side callback validation accepts URLs of the form `https://attacker.example\@victim.example/callback`. MediaWiki’s server-side PHP `parse_url`-based parser [46] interprets this URL as belonging to `victim.example`, treating `attacker.example\` as `userinfo`. However, when the same URL is later followed by the browser, the browser normalizes the backslash to a forward slash and navigates to `attacker.example`. This creates a mismatch between the origin validated by the OAuth provider and the origin that receives the redirected browser request.

An attacker can exploit this mismatch by sending the victim to a login URL for the third-party application that contains the crafted callback. The victim is shown the expected authorization page for the legitimate application. After the victim approves the request, MediaWiki’s OAuth provider redirects the browser to the callback URL and appends the OAuth temporary credentials, including the `oauth_token` and `oauth_verifier`. Because the browser resolves the

callback to the attacker-controlled origin, these credentials are delivered to the attacker. The attacker can replay the credentials against the legitimate application’s callback endpoint from a separate browser session, completing the OAuth flow and obtaining an authenticated session in the affected application as the victim. We responsibly disclosed this vulnerability to the Wikimedia Foundation, the organization maintaining MediaWiki, and it has since been confirmed and patched.

## 6.2. Malware Analysis Platforms

Another class of system that consumes URL parsing for security decisions is malware analysis platforms. Google Safe Browsing [13], VirusTotal [14], and similar services classify URLs as malicious or benign, and other systems treat their verdicts as a gate. When these services fail, they fail open, meaning that a failure to parse does not result in a malicious verdict. This is a structural property that interacts poorly with parser differentials. Additionally, downstream consumers typically only trigger alerts on identified positives to avoid alert fatigue; a parser differential that causes the analyzer to error, or to interpret the URL differently from the browser, converts the analyzer’s strictness into a bypass.

**Google Safe Browsing Bypass:** Safe Browsing’s parser rejects URLs whose scheme delimiter contains anything other than exactly two slashes. The browser normalizes the delimiter so that any number of slashes greater than or equal to 1 after `http:` are all accepted and resolved to the same destination. Submitting `http://malware.wicar.org` to the Safe Browsing API returns `Request` contains an invalid argument, while the browser navigates to `http://malware.wicar.org`, a flagged malicious host. Any service that uses Safe Browsing as a gate and treats a parse error as a non-malicious verdict can be bypassed by submitting a URL with anything other than two scheme slashes.

**VirusTotal:** VirusTotal exhibits the same fail-open behavior. We identified a set of URLs that the browser accepts and follows but VirusTotal rejects with a parse error. The browser collapses repeated slashes in a URL path or scheme. This is a core behavior of the WHATWG specification. Notably, VirusTotal rejects URLs containing anything aside from exactly two slashes separating the authority and scheme. This means that URLs such as `https://malware.wicar.org` and `https:///malware.wicar.org` will be missed by VirusTotal but navigated to by the browser.

Each of these inputs traverses the same path as a normal URL through any system that calls VirusTotal as a reputation gate. The input URL is treated as benign by the consumer because the parse failure looks like a missing verdict rather than a malicious one. The defect is not in VirusTotal’s classifier; it is in the difference between what the classifier can parse and what the browser will fetch. We responsibly disclosed these vulnerabilities to Google.

## 6.3. Abusing Open Redirects

As a direct consequence of the differentials in our dataset, we were able to trivially exploit multiple open redirects against login redirection flows. The targeted applications accept a user-supplied URL as the destination of a redirect, validate it against an allowlist or relative path check, and then issue a 3xx response pointing to the validated URL. The validator and the browser that follows the redirect form the canonical two-parser configuration described in our threat model. Any input on which they disagree is exploitable. If the validator deems the URL safe but the browser parses the same URL as a redirect to an attacker-controlled host, the validator is bypassed.

We identified open redirects in multiple widely deployed applications. Two of these, in Jenkins [11] and CTFd [12], share a single root cause. The WHATWG URL Standard specifies that ASCII tab (`\x09`), line feed (`\x0A`), and carriage return (`\x0D`) are stripped from a URL during parsing in the browser. Neither of the two validators applies the same stripping. An attacker can insert a URL-encoded tab into a payload that would otherwise be rejected. The validator preserves the tab and parses the input as safe, but the browser strips the tab and parses the result as a redirect.

**Jenkins:** Jenkins’s redirect validator, `Util.isSafeToRedirectTo` [47], accepts a URI when the URI is not an absolute URI, does not begin with a backslash, and does not begin with `//`. The third condition is intended to reject protocol-relative URLs. We exploit Jenkins with the URL `/%09/%09/example.com`. After the servlet container percent-decodes the parameter, the validator sees `/\t/\t/example.com`, which satisfies all three conditions. The URL is marked safe and Jenkins emits a 302 to the same string. The browser strips the two tabs per WHATWG and follows the result, `///example.com`, which it interprets as protocol-relative against the base URL. Navigation proceeds to the attacker-controlled host. We responsibly disclosed this vulnerability to the Jenkins security tracker, and it has since been confirmed, patched, and assigned a CVE.

**CTFd:** CTFd’s validator, `_is_safe_url` [48], is a port of Django’s allowlist check and explicitly anticipates WHATWG slash semantics. The function first rejects any input whose raw string begins with `///`. If the prefix check passes, the function calls Python’s `urlsplit` and accepts the input when the resulting netloc is empty.

This defense is insufficient. The exploit URL `/%09-//example.com` is decoded and passes the prefix check for `///` because of the tab. The input then proceeds to `urlsplit`, which operates on `///example.com`. `urlsplit` reports a relative path of `/example.com` with an empty netloc. Based on this data, the validator concludes the URL is relative and allows the redirect. The browser performs the same tab strip and resolves `///example.com` as a protocol-relative URL, navigating to the attacker-controlled host. The vulnerability is an instance of the check-then-act pattern described in Section 3.

The shared failure mode is a recurring pattern. Developers of CTFd were aware of WHATWG slash semantics and wrote checks against them. They did not account for `urlsplit` silently applying a different behavior than the browser and returning a parsed result that contradicted their prefix check. The inability of CTFd’s developers to prevent this vulnerability is not due to a lack of effort or security awareness, but rather due to the hidden complexities of URL parsing. We responsibly disclosed this vulnerability to the CTFd maintainers, and it has since been confirmed and patched.

## 7. Discussion

**Impact:** URL validation cannot be treated as a local property of a single parser. Security checks for redirects, callback URLs, webhooks, link previews, malware verdicts, and outbound fetches depend on agreement among all parsers that later consume the URL. Our results show that this agreement rarely holds, even within the same claimed standard or language ecosystem.

**Developer Guidance:** The safest design is to use the same parser for validation and action, pass structured components rather than raw URL strings across trust boundaries, and reject inputs that require downstream reinterpretation. When multiple parsers are unavoidable, the most practical defense is to shrink the input space until the remaining disagreement cannot be reached. Where the set of acceptable URLs is known in advance, applications should accept only an explicit allowlist, and otherwise constrain input with the most restrictive pattern the application can tolerate. An input that never exercises an ambiguous construct should not cause differentials, regardless of how many parsers consume it.

**Specification Guidance:** Standards should resolve the ambiguities our results expose rather than defer them to implementers. The WHATWG URL Standard, for example, defines a query only as “either null or an ASCII string,” defers the parsing of key and value pairs to `application/x-www-form-urlencoded`, and leaves duplicate keys and array syntax (`a[]=1`) to implementations. Underspecification of this kind creates the structural differentials Section 5.3 measures. Both standards should make an effort to clarify ambiguities, and should mark the choices they intentionally leave open rather than leaving them silent.

**Single-Parser Correctness:** TURLMOIL measures disagreement between parsers, not the correctness of any one parser. Its oracle is pairwise comparison, so it cannot tell which of two disagreeing parsers is wrong, and a fault that every parser shares produces no differential at all. Single-parser vulnerabilities, such as crashes, memory-safety bugs, or deviations from the specification a parser claims to follow, therefore fall outside our measurement. Future work could evaluate correctness directly. Our grammar and corpus transfer to that setting unchanged, and could be paired with reference implementations of RFC 3986 and the WHATWG URL Standard as a conformance oracle, as Reynolds et al. [3] did

for the hostname, or with sanitizer-instrumented builds of the native parsers to surface memory-safety faults.

**Limitations:** Our results show that URL parser differentials are pervasive, but two limitations are worth highlighting. First, TURLMOIL measures the 96 parsers in our corpus, not every parser, version, fork, or deployment-specific configuration. The dataset should therefore be read as a broad empirical lower bound on ecosystem divergence rather than a complete enumeration of URL parsing behavior.

Second, TURLMOIL focuses on cross-parser differentials rather than self-differentials or full re-serialization chains. Parsers may also become unsafe when their serialized output is reparsed into different components, or when repeated normalization is not idempotent. We leave a systematic study of these cases, especially in proxy, CDN, and middleware deployments, to future work.

## 8. Conclusion

URL parsers sit at the trust boundary of modern web applications, yet request paths increasingly compose multiple parsers across browsers, application logic, OAuth providers, and reputation services. In this paper, we presented the largest empirical study of URL parser differentials to date. Using TURLMOIL, a grammar-based, feedback-driven differential fuzzer, we showed that parser disagreement is pervasive across a corpus of 96 parsers: 99.96% of parser pairs diverge on at least one URL component, and 97.92% diverge on every component. We further showed that these differentials persist even among parsers that claim the same standard, undermining the assumption that same-standard parser composition is safe. Finally, we demonstrated that these disagreements have security consequences, enabling bypasses of URL-based security checks in real systems. Together, our results show that URL parser differentials are a systemic risk in modern web infrastructure.

## Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions. We would also like to acknowledge Alex Nahapetyan and a colleague who has chosen to remain anonymous for their feedback and support through the paper-writing process. This work is supported in part by the National Science Foundation (NSF) under grant numbers CNS-2047260 and CNS-2207008. The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF.

## References

- [1] O. Tsai, “A new era of SSRF — exploiting URL parser in trending programming languages,” BlackHat USA, 2017. [Online]. Available: <https://blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf>
- [2] N. Moshe, S. Brizinov, R. Onitza-Klugman, and K. Efimov, “Exploiting URL parsers: The good, bad, and inconsistent,” 2022, claroty Team82 / Snyk Whitepaper. [Online]. Available: <https://web-assets.claroty.com/exploiting-url-parsing-confusion.pdf>

- [3] J. Reynolds, A. Bates, and M. Bailey, "Equivocal URLs: Understanding the fragmented space of URL parser implementations," in *Proceedings of the 27th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2022. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-3-031-17143-7\\_9](https://link.springer.com/chapter/10.1007/978-3-031-17143-7_9)
- [4] Z. Sims, "When URL parsers disagree (CVE-2023-38633)," 2023, blog post. [Online]. Available: <https://www.canva.dev/blog/engineering/when-url-parsers-disagree-cve-2023-38633/>
- [5] D. K. Ajmani, I. Koishybayev, and A. Kapravelos, "yoU aRe a Liar://A unified framework for cross-testing URL parsers," in *Proceedings of the IEEE Security and Privacy Workshops (SecWeb)*, 2022. [Online]. Available: <https://www.kapravelos.com/publications/youarealiar-secweb22.pdf>
- [6] B. Kallus and S. W. Smith, "dippy\_gram: Grammar-aware, coverage-guided differential fuzzing of URL parsers," in *Proceedings of the Language-Theoretic Security Workshop (LangSec)*, 2023. [Online]. Available: [https://langsec.org/spw23/papers/Kallus\\_LangSec23.pdf](https://langsec.org/spw23/papers/Kallus_LangSec23.pdf)
- [7] O. Tsai, "How I chained 4 vulnerabilities on GitHub enterprise, from SSRF execution chain to RCE!" 2017, blog post. [Online]. Available: <https://blog.orange.tw/posts/2017-07-how-i-chained-4-vulnerabilities-on/>
- [8] Z. Fedotkin, "Introducing the URL validation bypass cheat sheet," 2024, blog post. [Online]. Available: <https://portswigger.net/research/introducing-the-url-validation-bypass-cheat-sheet>
- [9] P.-N. Tseng, "Defending against SSRF attacks (with help from our bug bounty program)," 2022, blog post. [Online]. Available: <https://dropbox.tech/security/bug-bounty-program-ssrf-attack>
- [10] MediaWiki. MediaWiki. [Online]. Available: <https://www.mediawiki.org/wiki/MediaWiki>
- [11] Jenkins. Jenkins. [Online]. Available: <https://www.jenkins.io/>
- [12] K. C. . C. LLC. CTFd : The Easiest Capture The Flag Framework. CTFd LLC. [Online]. Available: <https://ctfd.io/>
- [13] Google Safe Browsing. [Online]. Available: <https://safebrowsing.google.com>
- [14] "VirusTotal API v3 Overview," <https://docs.virustotal.com/reference/overview>.
- [15] RFC Editors, "RFC 1738: Uniform resource locators," <https://www.rfc-editor.org/rfc/rfc1738>, RFC Editor, Tech. Rep. RFC 1738, 1994.
- [16] T. Berners-Lee, R. Fielding, and L. Masinter, "RFC 3986: Uniform resource identifier (URI): Generic syntax," <https://www.rfc-editor.org/rfc/rfc3986>, RFC Editor, Tech. Rep. RFC 3986, 2005.
- [17] WHATWG, "URL living standard," <https://url.spec.whatwg.org>, 2026.
- [18] Y. Nizipli and D. Lemire, "Parsing millions of URLs per second," *Software: Practice and Experience*, 2023. [Online]. Available: <https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.3296>
- [19] A. Costello, "Punycode: A bootstring encoding of Unicode for internationalized domain names in applications (IDNA)," <http://rfc-editor.org/rfc/rfc3492.txt>, RFC Editor, Tech. Rep. RFC 3492, 2003.
- [20] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: Multiple host ambiguities in HTTP implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016. [Online]. Available: <https://dl.acm.org/doi/pdf/10.1145/2976749.2978394>
- [21] T. Chauchefoin, "Security implications of URL parsing differentials," 2022, blog post. [Online]. Available: <https://www.sonarsource.com/blog/security-implications-of-url-parsing-differentials/>
- [22] J. Leitschuh and D. Steinberg, "SSRF via maliciously crafted URL due to host confusion," 2019, hackerOne Bug Report. [Online]. Available: <https://hackerone.com/reports/704621>
- [23] junan 98, "CVE-2025-56200: validator.js bypass," 2025, cVE. [Online]. Available: <https://nvd.nist.gov/vuln/detail/CVE-2025-56200>
- [24] S. Khodayari, K. Glauber, and G. Pellegrino, "Do (not) follow the white rabbit: Challenging the myth of harmless open redirection," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025. [Online]. Available: <https://www.ndss-symposium.org/wp-content/uploads/2025-523-paper.pdf>
- [25] M. Doyhenard, "Gotta cache 'em all: bending the rules of web cache exploitation," 2024, dEFCOn/BlackHat. [Online]. Available: <https://portswigger.net/research/gotta-cache-em-all>
- [26] Q. Zhang, F. Liu, Z. Lin, and Y. Zhang, "Be aware of what you let pass: Demystifying URL-based authentication bypass vulnerability in Java web applications," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2025. [Online]. Available: <https://dl.acm.org/doi/pdf/10.1145/3719027.3765199>
- [27] Q. Yang, D. Damopoulos, and G. Portokalidis, "WYSISNWIIV: What you scan is not what I visit," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, 2015. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-3-319-26362-5\\_15](https://link.springer.com/chapter/10.1007/978-3-319-26362-5_15)
- [28] O. Tsai, "Breaking parser logic! take your path normalization off and pop 0days out," <https://i.blackhat.com/us-18/Wed-August-8/us-18-Orange-Tsai-Breaking-Parser-Logic-Take-Your-Path-Normalization-Off-And-Pop-0days-Out-2.pdf>, 2018, blackHat USA.
- [29] slicingmelon, "URL parser bypass tool," 2026, gitHub repository. [Online]. Available: <https://github.com/slicingmelon/gobypass403>
- [30] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang, "Where URLs become weapons: Automated discovery of SSRF vulnerabilities in web applications," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2024. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10646755>
- [31] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda, "Automated discovery of parameter pollution vulnerabilities in web applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011. [Online]. Available: <https://seclab.nu/static/publications/ndss2011hpp.pdf>
- [32] Stack Overflow, "Stack overflow developer survey 2024," 2024, accessed 2025. [Online]. Available: <https://survey.stackoverflow.co/2024/>
- [33] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.
- [34] IANA, "IDNA tables — Unicode 6.3.0," 2019, internet Assigned Numbers Authority. [Online]. Available: <https://www.iana.org/assignments/idna-tables-6.3.0/idna-tables-6.3.0.xhtml>
- [35] Celery - Distributed Task Queue — Celery 5.6.3 documentation. [Online]. Available: <https://docs.celeryq.dev/en/stable/>
- [36] "Url - Rust," <https://docs.rs/url/latest/url/>.
- [37] "Ada-url/ada," Ada, Jun. 2026.
- [38] Mozilla, "Mozilla's URL parser testing suite," 2026. [Online]. Available: <https://hg-edge.mozilla.org/mozilla-central/file/tip/browser/components/urlbar/tests/browser>
- [39] Slack, "About Us," <https://slack.com/about>.
- [40] "About Discord | Our Mission and Story," <https://discord.com/comp any>.
- [41] "What is Microsoft Teams? | Microsoft Support," <https://support.microsoft.com/en-us/teams/platform/what-is-microsoft-teams>.
- [42] "Libcurl - your network transfer library," <https://curl.se/libcurl/>.
- [43] Wikimedia Statistics - All wikis - Total page views. [Online]. Available: <https://stats.wikimedia.org/#/all-projects/reading/total-page-views/normalbar12-year1-totallmonthly>
- [44] "Project:Interface administrators," [https://www.mediawiki.org/wiki/Project:Interface\\_administrators](https://www.mediawiki.org/wiki/Project:Interface_administrators).

- [45] "OAuth/For Developers," [https://www.mediawiki.org/wiki/OAuth/For\\_Developers](https://www.mediawiki.org/wiki/OAuth/For_Developers).
- [46] MediaWiki: Includes/Utils/UrlUtils.php Source File. [Online]. Available: [https://doc.wikimedia.org/mediawiki-core/master/php/UrlUtils\\_8php\\_source.html#00427](https://doc.wikimedia.org/mediawiki-core/master/php/UrlUtils_8php_source.html#00427)
- [47] "Jenkins/core/src/main/java/hudson/Util.java at 063b6a7cdc81c590b2910fa8b692daf739d6e4cb · jenkinsci/jenkins," <https://github.com/jenkinsci/jenkins/blob/063b6a7cdc81c590b2910fa8b692daf739d6e4cb/core/src/main/java/hudson/Util.java>.
- [48] CTFd/CTFd/utils/validators/\_\_init\_\_.py at master · CTFd/CTFd. GitHub. [Online]. Available: [https://github.com/CTFd/CTFd/blob/master/CTFd/utils/validators/\\_\\_init\\_\\_.py](https://github.com/CTFd/CTFd/blob/master/CTFd/utils/validators/__init__.py)

## Appendix A. Fingerprinting Probes

TABLE 3: Fingerprinting probe sequence applied to Discord's link preview pipeline to identify rust-url.

| Probe                                              | Fetches | Remaining |
|----------------------------------------------------|---------|-----------|
| Dot segment + backslash (/a/./b\c)                 | Yes     | 15/96     |
| Encoded slash + dot segment (/a%2faa../)           | Yes     | 13/96     |
| Caret in path (/^)                                 | Yes     | 5/96      |
| Right brace in path (/})                           | Yes     | 4/96      |
| Backslash after non-special scheme (dict://host:\) | No      | 3/96      |
| All use rust-url                                   | —       | 1/96      |

TABLE 4: Fingerprinting probe sequence applied to Microsoft Teams' link preview pipeline to identify csharp-systemuri.

| Probe                                                  | Fetches | Remaining |
|--------------------------------------------------------|---------|-----------|
| Dot segment + backslash (/a/./b\c)                     | Yes     | 15/96     |
| Punycode label + empty query/fragment (xn-.../?#)      | Yes     | 11/96     |
| Asterisk in host label (...*...?#)                     | No      | 2/96      |
| Repeated slashes after authority (http://host////////) | Yes     | 1/96      |

## Appendix B. Algorithms

Figure 8 states the feedback-driven generation loop of Section 4, and Figure 9 states the fingerprinting procedure of Section 4.5.

```

Input : grammar G, logits w, parsers P, budget N
Output: differential dataset D, tuned logits w
D ← {}
for i = 1 to N:
  u, ch ← Sample(G, softmax(w)) # one per component
  for p in P:
    R[p] ← Parse(p, u) # EXCLUDE if unsupported
  diffs ← []
  for c in Components:
    for (a, b) in Pairs(P):
      if EXCLUDE in (R[a][c], R[b][c]): continue
      if R[a][c] != R[b][c]:
        D ← D + (a, b, c, u, R[a][c], R[b][c])
        diffs ← diffs + (a, b, c)
  for d in diffs:
    # each metric 0-100
    s[d] ← (Delta(d) + Rare(d) + Targeted(d)) / 3
  job ← max(s) if diffs else 0 # no differential: 0
  for c in Components with a differential:
    comp[c] ← max(s[d] : d on c)
  b ← mean(score of last B jobs) # baseline
  for k, (chosen, g) in ch: # k: component
    R ← comp[Field(k)] if Field(k) in comp else job
    A ← (R - b) / 100 # scores are 0-100
    π ← softmax(w[k][g]) # g: group of chosen
    for j in w[k][g]: # REINFORCE on logits
      w[k][j] += lr * A * ((j == chosen) - π[j])

```

Figure 8: Feedback-driven generation.

```

Input : target T, candidates C, dataset D
Output: identified layers L, residual candidates C
def SelectProbe(D, C, L, U):
  # unused probes every layer in L accepts, so
  # an error can only come from below L
  X ← {x in D - U : no q in L errors on x}
  if X is empty: return None
  # the probe that comes closest to halving C
  return argmin over x in X of
    abs(|{p in C : Errors(p, x)}| - |C|/2)
L ← []; U ← {}; C0 ← C; err ← false
while |C| > 1:
  x ← SelectProbe(D, C, L, U)
  if x is None: break # no usable split left
  U ← U + x
  r ← Send(T, x)
  if IsError(r):
    err ← true
    C ← {p in C : Errors(p, x)}
  else:
    C ← {p in C : not Errors(p, x)}
  if |C| == 1 and (L is empty or err):
    L ← L + C # peel to the next layer
    C ← C0 - L; U ← {}; err ← false
return L, C

```

Figure 9: Layered fingerprinting algorithm.

## Appendix C.

### Corpus

TABLE 5: Complete corpus of URL parsers evaluated. The **Signal** column gives the primary inclusion rationale: standard library status, notable usage, download volume, GitHub star count, an alternate implementation in the language, or strict specification conformance.

| No. | Parser                 | Language | Version  | Spec   | Signal                    |
|-----|------------------------|----------|----------|--------|---------------------------|
| 1   | cpp-ada-url            | C++      | 3.4.4    | WHATWG | Used in major browsers    |
| 2   | cpp-url-c-whatwg       | C++      | 6e87cbce | WHATWG | Reference implementation  |
| 3   | csharp-anglesharp      | C#       | 1.4.0    | WHATWG | 5.5k stars                |
| 4   | elixir-ex_url          | Elixir   | 2.0.2    | WHATWG | 314k downloads            |
| 5   | javascript-deno        | JS       | 2.7.8    | WHATWG | Standard library          |
| 6   | javascript-whatwg      | JS       | 25.8.2   | WHATWG | Standard library          |
| 7   | php-league             | PHP      | 7.8.1    | WHATWG | 230M downloads            |
| 8   | python-ada-url         | Python   | 1.31.0   | WHATWG | 2.3M monthly downloads    |
| 9   | rust-reqwest           | Rust     | 0.13.2   | WHATWG | 512M downloads            |
| 10  | rust-url               | Rust     | 2.5.8    | WHATWG | 664M downloads            |
| 11  | swift-weburl           | Swift    | 0.4.2    | WHATWG | Alternate implementation  |
| 12  | ada-uri-mime           | Ada      | 1.4.0    | RFC    | Standard library          |
| 13  | bin-aria2              | Binary   | 1.37.0   | RFC    | 41k stars                 |
| 14  | bin-trurl              | Binary   | 0.14.0   | RFC    | curl.se tooling           |
| 15  | bin-wget               | Binary   | 1.24.5   | RFC    | GNU tooling               |
| 16  | clojure-uri            | Clojure  | 1.19.155 | RFC    | 252 stars                 |
| 17  | cpp-boost-url          | C++      | 1.83.0   | RFC    | Boost C++ Libraries       |
| 18  | cpp-guri               | C++      | 2.80.0   | RFC    | GNOME GLib                |
| 19  | cpp-http-parser        | C++      | 2.9.4    | RFC    | 6.4k stars                |
| 20  | cpp-libcurl            | C++      | 8.19.0   | RFC    | Bundled in libcurl        |
| 21  | cpp-poco-uri           | C++      | 1.15.2   | RFC    | 9.4k stars                |
| 22  | cpp-qurl-strict        | C++      | 6.11.0   | RFC    | Qt framework              |
| 23  | cpp-qurl-tolerant      | C++      | 6.11.0   | RFC    | Qt framework              |
| 24  | cpp-uriparser          | C++      | 1.0.0    | RFC    | 408 stars                 |
| 25  | cpp-url-c-rfc          | C++      | 6e87cbce | RFC    | Reference implementation  |
| 26  | crystal-uri            | Crystal  | 1.20.1   | RFC    | Standard library          |
| 27  | csharp-flurl           | C#       | 4.0.0    | RFC    | 162M downloads            |
| 28  | dart-core              | Dart     | 3.11.5   | RFC    | Standard library          |
| 29  | elixir-uri             | Elixir   | 1.15     | RFC    | Standard library          |
| 30  | erlang-hackney-url     | Erlang   | 4.0.0    | RFC    | 1.4k stars                |
| 31  | erlang-ibrowse-lib     | Erlang   | 4.4.2    | RFC    | 518 stars                 |
| 32  | erlang-uri-string      | Erlang   | 27       | RFC    | Standard library          |
| 33  | go-net                 | Go       | 1.21     | RFC    | Standard library          |
| 34  | haskell-modern-uri     | Haskell  | 0.3.6.1  | RFC    | 26k downloads             |
| 35  | haskell-network-uri    | Haskell  | 2.6.4.2  | RFC    | 184k downloads            |
| 36  | haskell-uri-bytestring | Haskell  | 0.4.0.1  | RFC    | 49k downloads             |
| 37  | java-apache-httpclient | Java     | 4.5.13   | RFC    | Apache HttpComponents     |
| 38  | java-netty             | Java     | 4.1.132  | RFC    | 35k stars                 |
| 39  | java-okhttp            | Java     | 4.9.3    | RFC    | 47k stars                 |
| 40  | java-spring-web        | Java     | 7.0.6    | RFC    | 60k stars                 |
| 41  | java-uri               | Java     | 17       | RFC    | Standard library          |
| 42  | java-url               | Java     | 17       | RFC    | Standard library          |
| 43  | javascript-fast-uri    | JS       | 3.1.0    | RFC    | 73M weekly downloads      |
| 44  | javascript-parse-uri   | JS       | 2.0.5    | RFC    | 658k monthly downloads    |
| 45  | javascript-uri-js      | JS       | 4.4.1    | RFC    | 104M weekly downloads     |
| 46  | javascript-uri-parser  | JS       | 1.0.1    | RFC    | Alternate implementation  |
| 47  | javascript-urijs       | JS       | 1.19.11  | RFC    | 4.9M weekly downloads     |
| 48  | javascript-url-toolkit | JS       | 2.2.5    | RFC    | 541k weekly downloads     |
| 49  | perl-mojo-url          | Perl     | 9.42     | RFC    | 2.7k stars                |
| 50  | perl-uri               | Perl     | 5.34     | RFC    | De facto Perl URI library |
| 51  | php-amphp-uri          | PHP      | 0.1.4    | RFC    | 1.9M downloads            |
| 52  | php-guzzle-psr7        | PHP      | 2.9.0    | RFC    | 1.05B downloads           |

*Continued on next page*

Table 5 continued from previous page

| No. | Parser                     | Language | Version | Spec   | Signal                   |
|-----|----------------------------|----------|---------|--------|--------------------------|
| 53  | php-laminas                | PHP      | 2.14.0  | RFC    | 37M downloads            |
| 54  | php-ml-iri                 | PHP      | 1.1.4   | RFC    | 6.9M downloads           |
| 55  | php-nette-http             | PHP      | 3.3.3   | RFC    | 20M downloads            |
| 56  | php-phrity-net-uri         | PHP      | 2.2.1   | RFC    | 11M downloads            |
| 57  | php-sabre                  | PHP      | 3.0.2   | RFC    | 37M downloads            |
| 58  | python-furl                | Python   | 2.1.4   | RFC    | 2.8k stars               |
| 59  | python-httpx               | Python   | 0.28.1  | RFC    | 717M monthly downloads   |
| 60  | python-hyperlink           | Python   | 21.0.0  | RFC    | 31M monthly downloads    |
| 61  | python-purl                | Python   | 1.6     | RFC    | 486k monthly downloads   |
| 62  | python-rfc3986             | Python   | 2.0.0   | RFC    | Reference implementation |
| 63  | python-uritools            | Python   | 6.0.1   | RFC    | 10M monthly downloads    |
| 64  | python-urllib3             | Python   | 3.14.3  | RFC    | Standard library         |
| 65  | python-yarl                | Python   | 1.23.0  | RFC    | 500M monthly downloads   |
| 66  | ruby-addressable           | Ruby     | 2.8.9   | RFC    | 1.6k stars               |
| 67  | ruby-uri                   | Ruby     | 1.1.1   | RFC    | Standard library         |
| 68  | rust-fluent-uri            | Rust     | 0.4.1   | RFC    | 45M downloads            |
| 69  | rust-iref                  | Rust     | 4.0.0   | RFC    | 6M downloads             |
| 70  | rust-uriparse              | Rust     | 0.6.4   | RFC    | 28M downloads            |
| 71  | swift-url                  | Swift    | 5.9     | RFC    | Standard library         |
| 72  | swift-urlcomponents        | Swift    | 5.9     | RFC    | Standard library         |
| 73  | zig-std-uri                | Zig      | 0.14.0  | RFC    | Standard library         |
| 74  | csharp-systemuri           | C#       | .NET 10 | Hybrid | Standard library         |
| 75  | haskell-http-client        | Haskell  | 0.7.19  | Hybrid | 372k downloads           |
| 76  | haskell-http-conduit       | Haskell  | 2.3.9.1 | Hybrid | 299k downloads           |
| 77  | java-galimatias            | Java     | 0.2.1   | Hybrid | ~1M annual downloads     |
| 78  | javascript-domurl          | JS       | 2.3.4   | Hybrid | 25k weekly downloads     |
| 79  | javascript-fast-url-parser | JS       | 1.1.3   | Hybrid | 2.6M weekly downloads    |
| 80  | javascript-jsuri           | JS       | 1.3.1   | Hybrid | 119k weekly downloads    |
| 81  | javascript-legacy          | JS       | 25.8.2  | Hybrid | Standard library         |
| 82  | javascript-parse-url       | JS       | 11.1.0  | Hybrid | 4.7M weekly downloads    |
| 83  | javascript-parseuri        | JS       | 3.0.2   | Hybrid | 1.5M weekly downloads    |
| 84  | javascript-smithy          | JS       | 4.2.12  | Hybrid | 36M weekly downloads     |
| 85  | javascript-url-parse       | JS       | 1.5.10  | Hybrid | 34M weekly downloads     |
| 86  | php-opis-uri               | PHP      | 1.1.0   | Hybrid | 25M downloads            |
| 87  | php-parseurl               | PHP      | 8.4.19  | Hybrid | Standard library         |
| 88  | php-pear-net-url2          | PHP      | 2.2.3   | Hybrid | 13M downloads            |
| 89  | php-spatie                 | PHP      | 2.4.0   | Hybrid | 15M downloads            |
| 90  | python-lazr-uri            | Python   | 1.0.7   | Hybrid | 264k monthly downloads   |
| 91  | python-mdurl               | Python   | 0.1.2   | Hybrid | 513M monthly downloads   |
| 92  | r-http2                    | R        | 1.1.2   | Hybrid | Maintained by r-lib      |
| 93  | r-urlparse                 | R        | 0.2.1   | Hybrid | Alternate implementation |
| 94  | r-urltools                 | R        | 1.7.3   | Hybrid | Alternate implementation |
| 95  | rust-hyper-uri             | Rust     | 1.8.1   | Hybrid | 691M downloads           |
| 96  | rust-urlparse              | Rust     | 0.7.3   | Hybrid | 3.2M downloads           |

## Appendix D. Generative AI

Generative AI was used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

## Appendix E. Ethics Considerations

**Vulnerability Disclosure:** We disclosed each of the vulnerabilities described in Section 6 through the channel appropriate to the affected system. MediaWiki was reported to the Wikimedia Foundation through their preferred security bug reporting platform, Phabricator. Safe Browsing and VirusTotal were reported through Google’s Vulnerability Reward Program. CTFd was reported under GitHub’s Security Advisory process (GHSA). Jenkins and Slack were also reported through their respective security advisory processes. For each disclosure we provided a minimal reproducing input, identified the parser pair responsible, and proposed a remediation.

**Working With Live Systems:** Our case studies divide into two categories. For open-source systems (MediaWiki, Jenkins, CTFd) we reproduced every vulnerability on a local instance of the affected version, with test accounts and third-party application configurations we constructed for the purpose. For closed-source platforms (Slack, Google Safe Browsing, VirusTotal, Discord, Teams) local reproduction was not possible, and we tested against the public service within the published scope of the relevant bug bounty program or vulnerability reporting program. Link rendering was confined to the link-preview pipeline, where we sent messages from a dedicated test workspace and did not exfiltrate, persist, or forward any data observed in response. Google API testing was confined to URL submissions we controlled, with the exception of one well-known publicly available malicious test domain (`malware.wicar.org`) used to verify the fail-open bypass. No account, system, or data outside our control was probed at any point in this study. All targets were selected from scope explicitly authorized under various bug bounty programs.

**Fingerprinting and Dataset:** An additional ethical question surrounding this work is whether publishing the dataset, the techniques, and the fingerprinting method helps defenders more than it helps attackers. The material attackers need to mount parser differential attacks is already in the open record. What is missing is the systematic, cross-ecosystem measurement defenders need to determine whether their own composition is vulnerable. Withholding the dataset would leave that asymmetry intact: attackers retain the resources required to find differentials by hand, and defenders retain no efficient way to check.

We do not publish a turnkey tool that automates fingerprinting and dataset lookup against arbitrary live targets, since such a tool is not required for the scientific contribution and would lower the floor for opportunistic attackers without

commensurate defender benefit. A sophisticated attacker who chooses to reproduce our infrastructure (a Kubernetes cluster, Celery task queue, custom harnesses for 96 parsers across 21 language ecosystems, the PCFG generator, and the analysis pipeline) is already operating at a level where the public record is sufficient input. A defender wanting to audit their own composition has no equivalent shortcut absent our dataset. The asymmetry favors defenders.

## **Appendix F.**

### **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.

#### **F.1. Summary**

This paper presents TURLMOIL, a grammar-based, feedback-driven differential fuzzer for large-scale testing of URL parser disagreements. The authors evaluate 96 parsers across 21 language ecosystems and discover widespread parsing differentials: 99.96% of parser pairs diverge on at least one URL component, and 97.92% diverge on every component, with notable discrepancies even among parsers that follow the same specification. The authors organize these discrepancies into 21 attack techniques. They also introduce a parser fingerprinting method to identify parser combinations in black-box settings and uncover six vulnerabilities in real-world systems.

#### **F.2. Scientific Contributions**

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

#### **F.3. Reasons for Acceptance**

- 1) The paper identifies impactful vulnerabilities. The authors uncover six vulnerabilities in deployed systems, including MediaWiki, Jenkins, CTfD, Google Safe Browsing, and VirusTotal. These vulnerabilities were responsibly disclosed to the respective maintainers, demonstrating the real-world security impact of the work.
- 2) The paper creates a new tool that can enable future research. TURLMOIL provides a grammar-based, feedback-driven differential fuzzer specifically designed for URL parsers. Reviewers noted that the individual components of TURLMOIL are not themselves novel, but that the tool as a whole constitutes a valuable contribution to the field. It enables large-scale testing of URL parser disagreements and can be used by other researchers to further investigate this area.
- 3) The paper provides a valuable step forward in an established field. While parser differentials and URL parsing issues have been studied previously, this work provides the first large-scale, systematic measurement across 96 parsers and 21 language ecosystems.