AutoSpear: Bypassing and Inspecting Web Application Firewalls

We present an automated framework, **AutoSpear**, to intelligently bypass and inspect Web Application Firewalls (WAFs) through semantically meaningful SQL injection payloads. AutoSpear represents each SQLi payload as a hierarchical tree, enabling fine-grained, structure-aware transformations via context-free grammar. This encoding allows the system to preserve the malicious intent while applying robust mutations, such as comment injection and whitespace substitution. To search the most evasive mutations efficiently, we incorporate a Monte Carlo Tree Search strategy that operates without requiring internal WAF feedback, making it ideal for black-box environments. AutoSpear also reconstructs payloads from modified trees to generate real-world attack traffic and evaluates bypass success across multiple commercial WAF services. Our evaluations show that AutoSpear achieves high attack success rates and outperforms both handcrafted and machine learning–based bypass strategies, confirming the vulnerabilities in mainstream WAF-as-a-service platforms.

1 Web Attacks and the Role of Web Application Firewalls (WAFs)

Web applications are among the most frequently targeted assets in the digital landscape, with attack vectors evolving rapidly over time. Classic vulnerabilities such as SQL injection (SQLi), Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and Server-Side Request Forgery (SSRF) continue to pose significant risks despite advancements in secure development practices. Attackers leverage these vulnerabilities to exfiltrate sensitive data, escalate privileges, or disrupt services, making them a critical concern for enterprises of all sizes.

In response, Web Application Firewalls (WAFs) have emerged as a frontline defense mechanism. A WAF operates by monitoring, filtering, and blocking HTTP traffic to and from a web application, applying a set of security rules based on known attack signatures and behavioral heuristics. Early WAFs primarily relied on static rule-based detection, where patterns like typical SQL keywords or JavaScript snippets triggered alerts or blocks. However, attackers quickly adapted by developing evasion techniques that circumvented these static defenses.

Modern WAFs incorporate machine learning (ML) models and behavioral analysis to detect anomalies and previously unseen attack patterns. These systems analyze various features, such as input length, entropy, token distribution, and request structure, to flag suspicious activities. Nonetheless, the addition of ML brings its own challenges, including susceptibility to adversarial examples and model drift over time. Moreover, the trade-off between minimizing false positives and maximizing detection efficacy remains a persistent balancing act, leading to inconsistencies in WAF performance across different deployments.

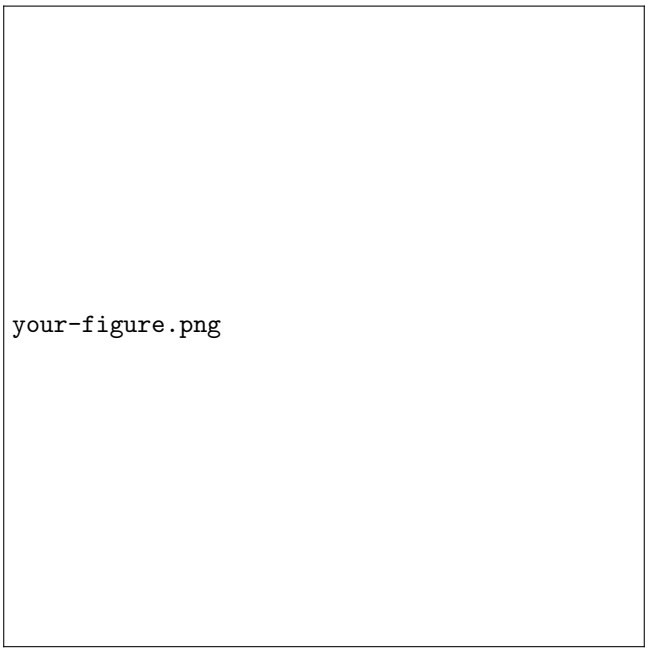

Figure 1: Web Application Firewall Detection Process

2 Techniques and Challenges in Bypassing WAFs

Bypassing a WAF typically involves crafting requests that either obfuscate malicious intent or exploit inconsistencies in parsing logic. These techniques are often classified into three major categories: architecture-level, protocol-level, and payload-level evasion. Architecture-level attacks target deployment flaws, such as inconsistencies between load balancers and WAFs. Protocol-level attacks exploit discrepancies in how HTTP requests are interpreted by different layers. However, payload-level evasion—transforming the actual malicious payload—remains the most accessible and portable strategy across different environments.

Payload-level techniques exploit the flexibility and redundancy in program-

ming languages and protocols. For instance, attackers may use URL encoding, alternate casing, inline comments, unusual whitespace, or even non-standard delimiters to alter the appearance of an attack without changing its underlying logic. More advanced payload transformations involve exploiting SQL dialect-specific features or abusing lesser-known protocol extensions, complicating detection further.

Technique	**Description**
URL Encoding	Obfuscating payloads using URL encoding
Inline Comments	Inserting comments within SQL keywords
Whitespace Substitution	Replacing spaces with other characters like %20

Table 1: Common Payload-Level Evasion Techniques

Despite their effectiveness, automating these transformations in a way that consistently bypasses WAFs is extremely challenging. Naive random mutations often produce syntactically invalid or semantically broken payloads that are either rejected by the target application or flagged by basic input validation layers. Moreover, effective payload crafting demands an understanding of both the grammar of the injection language (e.g., SQL) and the specific quirks of the target application's input handling. Therefore, successful automation requires sophisticated mutation strategies that maintain payload validity while maximizing evasion potential.

Another complexity arises from the diversity of real-world deployment scenarios. Some WAFs are placed behind content delivery networks (CDNs) or operate in distributed environments, introducing inconsistencies based on caching, latency, or parsing anomalies. Additionally, different request formats—such as JSON bodies versus traditional URL-encoded parameters—may trigger different inspection paths within a WAF, necessitating format-aware attack generation strategies.

3 Limitations of Existing WAF Bypass Methods

Most existing WAF bypass tools fall into two broad categories: static transformation engines and machine learning-based adversarial generators. Static tools, like those integrated into platforms such as SQLMap, apply a fixed set of predefined mutations to payloads. These include adding inline comments between keywords, using alternative encodings, or inserting redundant operations. While such techniques can be effective against basic or poorly maintained WAFs, they tend to perform poorly against adaptive or learning-based defenses.

A significant limitation of static approaches is their lack of semantic awareness. These tools often treat the payload as a raw string, indiscriminately applying transformations without understanding the syntactic structure or semantic dependencies within the input. This leads to payloads that either fail at execution or are easily detected by WAFs trained on recognizing common ob-

fuscation patterns. Furthermore, because these transformations are static, they fail to generalize across different WAF configurations or adapt to new detection strategies.

On the other hand, adversarial machine learning approaches show promise in generating sophisticated evasion payloads. These methods often involve training generative models or perturbation engines against a surrogate WAF model to find inputs that are misclassified. However, such techniques require extensive training data, computational resources, and access to feedback signals from the WAF under attack—conditions that are rarely available in practical black-box testing scenarios. Moreover, these models can be brittle, overfitting to specific WAF behaviors and failing to generalize to unseen systems.

Consequently, there is a pressing need for an automated, generalizable, and semantic-aware WAF bypass strategy that operates efficiently in black-box settings without extensive prior knowledge or manual intervention.

4 Introducing AutoSpear: Automated WAF Bypass and Inspection Tool

AutoSpear is designed to bridge the gaps left by existing tools through a novel combination of grammar-aware payload encoding and intelligent search strategies. Unlike traditional static mutators, AutoSpear models each SQL injection payload as a structured syntax tree, enabling transformations that respect both syntactic correctness and semantic intent. This ensures that mutated payloads remain functional and capable of executing intended attacks, while appearing sufficiently novel to evade detection.

Central to AutoSpear's search capability is the use of Monte Carlo Tree Search (MCTS), a powerful strategy originally popularized in game AI domains such as AlphaGo. MCTS allows AutoSpear to efficiently explore large mutation spaces without exhaustive enumeration, balancing exploration of novel mutation paths with exploitation of promising transformations. Importantly, AutoSpear's MCTS algorithm operates independently of internal WAF feedback, making it highly effective in real-world black-box testing environments where no detailed error signals are available.

Another key feature of AutoSpear is its modular grammar design. By abstracting SQL syntax rules into context-free grammars (CFGs), AutoSpear can easily adapt to different SQL dialects or customize transformations based on target-specific characteristics. This modularity not only enhances the framework's generalizability but also facilitates extensibility to other injection-based attack types, such as XPath or LDAP injection.

Extensive empirical evaluations against a variety of commercial WAF-as-a-service platforms demonstrate AutoSpear's superior performance. In comparative studies, AutoSpear consistently achieved higher bypass success rates and uncovered detection weaknesses that were missed by both manual testing and existing automated tools. These results underscore the critical importance of

structured, semantic-aware mutation strategies in advancing the state-of-the-art in web application security testing.

5 Hierarchical Tree Representation of SQLi Payloads

At the heart of AutoSpear's mutation engine lies its hierarchical tree representation of SQLi payloads. Rather than treating the payload as a flat string, AutoSpear decomposes it into a structured tree, capturing the syntactic and semantic relationships between different components. Each payload is divided into three primary segments: the left boundary (e.g., opening quotes or parentheses), the SQL core (i.e., the injectable query fragment), and the right boundary (e.g., trailing comments or quotes).

The SQL core is further parsed into a syntax tree based on a custom-defined context-free grammar. Leaf nodes in the tree represent atomic SQL tokens, such as keywords (e.g., SELECT, FROM), operators (e.g., =, AND), identifiers (e.g., table names), and literals (e.g., strings or numbers). Internal nodes correspond to higher-level constructs, such as expressions, predicates, or full query statements. This rich structural information enables AutoSpear to apply mutations at precise levels of granularity, ensuring that modifications preserve both syntactic validity and semantic coherence.

Mutations in AutoSpear are defined as grammar-preserving tree transformations. Examples include inserting benign tautologies (e.g., 1=1) into WHERE clauses, replacing comparison operators with equivalent constructs, or interleaving comments within identifiers and keywords. By operating on the tree rather than the string, AutoSpear avoids common pitfalls such as breaking quotation contexts, mismatching parentheses, or corrupting query logic.

Finally, the modified syntax tree is serialized back into a concrete SQL payload, incorporating format-specific adjustments as needed (e.g., URL encoding or JSON formatting). This end-to-end process—from tree parsing to mutation to reserialization—enables AutoSpear to generate highly effective evasion payloads while maintaining robust correctness guarantees, marking a significant advancement over prior approaches.