

AACL: Artificial Adaptive Control Language

M Venkata Sasidhar Kaushik

February 2026

Abstract

Web applications commonly rely on widely deployed defenses such as tokens, headers, and server-side validation logic to mitigate request forgery, tampering, and replay. While effective against many threats, these mechanisms primarily establish authenticity and coarse integrity — validating who is acting and whether they are authorized — but do not formally enforce the precise structural and semantic contract of individual stateful action instances. This paper introduces AACL (Artificial Adaptive Control Language), a language-theoretic control framework in which the server dynamically issues an ephemeral, intent-bound grammar for each sensitive action instance. Unlike existing mechanisms, AACL defines and enforces the complete acceptable form of a specific request instance — constraining structure, parameter relationships, state-bound semantic values, and temporal validity within a single formal acceptance condition. Each request is treated as a short-lived language instance and validated through formal recognition grounded in a deterministic finite automaton (DFA) model before execution. In the prototype realization, the DFA acceptance condition is implemented as key-set equality over intent-specific alphabets — a tractable and sufficient structural enforcement mechanism for JSON-based web API payloads. Grammars are state-bound, derived from server-side snapshots at issuance time, and invalidated after use or expiry, enforcing single-use semantics and preventing replay under explicit threat-model assumptions. AACL is designed as a complementary enforcement layer and does not replace authentication, authorization, or transport security — rather, it adds per-instance structural and semantic intent integrity at the language-recognition layer, addressing a gap that existing mechanisms such as CSRF tokens, JWTs, nonces, and WAFs do not cover. We formalize the AACL model, describe its integration into a web application architecture, and present a prototype demonstrating how ephemeral grammars reduce replay and structural mutation attacks. Empirical evaluation over 5000 requests confirms linear validation complexity $O(n)$ and modest latency overhead suitable for high-value, sensitive web operations.

1 Introduction

Modern web applications execute sensitive operations whose security depends critically on the integrity of client requests. In practice, web security has evolved through layered mechanisms such as CSRF tokens, signed claims, nonces, input validation logic, and Web Application Firewalls. These mechanisms are effective against many known threats, yet they typically validate authenticity and coarse integrity rather than formally enforcing the precise structural and semantic contract of a specific stateful action instance.

In particular, widely deployed controls authenticate who is acting and verify certain request properties, but they do not generally define or enforce a per-action language that captures the exact structure, allowable parameter relationships, temporal validity, and single-use semantics of an intended interaction. As a result, replay and structure-mutation attacks remain possible whenever a once-valid request can be resubmitted within its validity window or modified while remaining syntactically plausible.

Language-Theoretic Security (LangSec) advocates treating inputs as formal languages and parsing as a security-critical operation. While highly effective for static protocols and data formats, most LangSec instantiations assume globally fixed grammars and do not directly address dynamic, state-dependent application workflows. Modern web interactions, however, are inherently contextual: the validity of a request depends not only on syntax but also on current application state, user session context, and temporal constraints.

This paper introduces AACL (Artificial Adaptive Control Language), a framework that extends language-theoretic principles to dynamic web application actions. Instead of relying on static schemas or long-lived tokens, AACL issues ephemeral, intent-bound grammars for each sensitive action instance. Unlike conventional defenses that rely on static validation schemas, bearer tokens, or replay-detection heuristics, AACL introduces a per-instance, state-bound language recognition model for web actions. The key novelty lies in (i) dynamically generating intent-specific grammars tied to a server-side state snapshot, (ii) enforcing deterministic structural recognition prior to business logic execution, and (iii) combining semantic validation with single-use and temporal constraints within the same formal acceptance condition. This shifts enforcement from authenticity verification toward explicit language-based intent integrity, extending Language-Theoretic Security principles to dynamic, stateful web workflows.

Each request must be recognized under its corresponding grammar before execution, and the grammar is invalidated after use or expiry. By binding accepted request languages to server-side state snapshots and enforcing single-use semantics, AACL aims to eliminate a class of replay and structural mutation attacks under clearly defined threat-model assumptions.

We formalize the AACL model, describe its integration into a web application architecture, and demonstrate a prototype implementation illustrating how ephemeral grammars can serve as a state-aware structural enforcement layer complementing existing authentication and authorization mechanisms.

1.1 Contributions

The primary contributions of this paper are:

- A conceptual framework for generating and enforcing ephemeral, intent-bound grammars to secure web application requests.
- An articulation of how ephemerality and intent binding mitigate replay and request-tampering attacks by making stale or altered requests semantically invalid.
- A prototype architecture and proof-of-concept workflow demonstrating how AACL can be integrated into a web application request pipeline.

2 Related Work and Gap Analysis

Web application security relies on layered defenses including authentication, authorization, and input validation. This section compares prominent mechanisms and highlights a gap: most defenses establish authenticity, integrity, or coarse syntactic validity, but do not enforce a per-action, per-instance language capturing structure, state binding, and single-use semantic intent.

2.1 Token-Based and Session-Bound Defenses

Modern web applications widely deploy CSRF tokens, session identifiers, and related header-based mechanisms to bind requests to authenticated user sessions. These mechanisms effectively mitigate

cross-site request forgery and unauthorized cross-origin submissions by ensuring that a request originates from an authenticated context [11]. However, such tokens primarily establish authenticity and session continuity. They do not formally encode or enforce the complete structural contract of a specific action instance, nor do they inherently guarantee single-use semantic intent beyond what is explicitly implemented in application logic [16]. For instance, CSRF tokens prevent unauthorized submissions but leave same-origin replays possible within the token’s validity window, as long as the request remains syntactically plausible [11]. Session fixation attacks further undermine continuity if tokens are not rotated per-action [17]. While extensions like double-submit cookies add integrity, they remain coarse-grained and decoupled from dynamic state (e.g., current user balance or temporal constraints), relying on ad-hoc application-layer checks.

2.2 Signed Claims and Stateless Authorization

JSON Web Tokens (JWTs) and similar signed-claim systems provide integrity and authenticity for embedded authorization data [28]. By cryptographically binding claims to a signature, these mechanisms prevent tampering with protected metadata. Nevertheless, the surrounding request payload remains structurally independent of the signed claims. While JWTs protect identity and role assertions, they do not generally express per-intent structural constraints or state-dependent action validity [29]. As a result, semantic correctness of individual actions remains enforced at the application layer rather than at the language-recognition layer. OAuth 2.0 and related frameworks extend this with bearer tokens for delegated access [29, 30], but vulnerabilities like token leakage or replay within expiry periods persist [30]. These systems are stateless by design, making per-instance state binding challenging without additional nonce-like mechanisms, which introduce their own management overhead [16].

2.3 Nonce and One-Time Token Mechanisms

Nonce-based systems and single-use tokens attempt to mitigate replay attacks by introducing uniqueness constraints. When correctly implemented, they can reduce repeated submission of identical requests. However, these mechanisms typically function as auxiliary parameters attached to otherwise conventional request formats. They do not define a complete language for the request itself, nor do they inherently constrain the relationships among parameters or encode dynamic state-dependent bounds within a formally recognized structure.

2.4 Language-Theoretic Security (LangSec)

Language-Theoretic Security (LangSec) represents a security philosophy advocating that all input to a program must be treated as a program in some language, and thus rigorously validated against a formal grammar before processing [7]. The core principle asserts that parsing, as the process of recognizing whether an input string conforms to a grammar, is a security-critical operation. By strictly enforcing a minimal, unambiguous grammar, LangSec seeks to eliminate “unexpected” inputs that often serve as vectors for attacks like buffer overflows, format string vulnerabilities, and injection flaws [14]. The premise is that if a parser is correct and the grammar precisely defines all valid inputs, then any input not conforming to this grammar is inherently malicious or erroneous and should be rejected. This shifts the focus from identifying known malicious patterns to defining and enforcing the universe of acceptable inputs.

Traditionally, LangSec approaches have assumed a static grammar for a given protocol or data format. For instance, the grammar for HTTP requests, JSON payloads, or XML documents is fixed and well-defined. This static assumption simplifies implementation and formal verification, allowing

for robust parsers to be built and extensively tested. The emphasis is on the correctness of the parser and the completeness of the grammar in describing all valid inputs for a particular interface or data exchange format. While highly effective for ensuring syntactic correctness and preventing many parsing-related vulnerabilities, this static nature presents limitations when applied to the dynamic, interactive context of modern web applications. The meaning and validity of a web request often transcend its mere syntactic structure; they are deeply intertwined with the application’s current state, the user’s permissions, and the intended flow of interaction.

2.5 Research Gap

Across these approaches, a consistent limitation emerges: while authenticity, integrity, and syntactic correctness are well-addressed, there is limited support for dynamically generating and enforcing a per-action, per-instance request language that encodes both structural constraints and state-dependent semantic intent, and that is invalidated after use.

Table 1: AACL vs Existing Mechanisms (✓ strong, △ partial, × weak)

Mechanism	Replay	Mutation	Latency	Binding	Formal	Gap Filled
CSRF Tokens	△	×	0.1ms	△	×	No structure
JWTs	△	△	0.2ms	×	△	No state bind
Nonces	✓	×	0.05ms	△	×	No semantics
Input Validation	×	△	0.5ms	×	×	Static rules
WAFs	△	△	1–5ms	×	△	Pattern only
AACL (Proposed)	✓	✓	<1ms	✓	✓	All gaps

AACL is positioned to address this gap by extending language-theoretic principles to dynamic web application actions. Instead of relying on static schemas or long-lived tokens, AACL introduces ephemeral, intent-bound grammars that define the accepted language for a specific action instance and are destroyed after execution or expiry. This shifts enforcement from token validation or heuristic filtering toward formal, state-aware language recognition. LangSec argues that many security bugs stem from ad hoc parsing and ambiguous input formats, and advocates for strict language definitions with correct parsers. Static grammars, however, do not capture application state, temporal constraints, or per-action intent that are central to many web workflows.

3 System Model and Threat Assumptions

3.1 System Model

We consider a stateful web application consisting of a client, an application server, and a persistent backend datastore. Users interact with the system through authenticated sessions over secure transport (e.g., HTTPS). Sensitive operations are modeled as discrete action intents, such as login, password change, or financial transfer.

Each action intent corresponds to a server-side workflow that processes structured request parameters and may modify application state. The server maintains authoritative state, including session context, user identity, and relevant application variables (e.g., account balances, authorization scopes).

Under conventional architectures, requests are validated using a combination of authentication tokens, session identifiers, parameter validation logic, and application-specific checks before execution.

In the AACL model, for selected sensitive actions, the server additionally generates an ephemeral, intent-bound grammar instance derived from the current server-side state snapshot. This grammar

defines the accepted language of valid requests for that specific action instance. The grammar is transmitted to the client in a protected form and is invalidated after successful execution or expiration.

The server executes an action only if the incoming request is recognized by the corresponding grammar and satisfies associated state and temporal constraints.

3.2 Attacker Model

We assume a network-capable adversary with the following capabilities:

- The ability to intercept, replay, and modify previously observed application-layer requests.
- The ability to craft arbitrary client-side requests within an authenticated session context.
- Full knowledge of the AACL framework design and enforcement logic (Kerckhoffs’s principle).
- Access to common web exploitation techniques, including parameter tampering, replay attempts, and structural mutation of request payloads.

The adversary does not possess:

- The ability to compromise the server or alter server-side code.
- The ability to break underlying cryptographic primitives (e.g., TLS, digital signatures).
- The ability to bypass authentication controls to obtain unauthorized sessions.

We assume that authentication and transport security are correctly implemented and are not the focus of this work.

3.3 Threat Scope

AACL is designed to address a class of attacks characterized by:

- Replay of previously valid state-changing requests.
- Structural mutation of request parameters while remaining syntactically plausible.
- Reuse of once-valid request formats beyond their intended semantic context.

AACL does not claim to eliminate vulnerabilities arising from:

- Server-side implementation flaws unrelated to request structure.
- Injection vulnerabilities caused by unsafe query construction or improper output encoding.
- Logical authorization errors within business logic.

These concerns remain the responsibility of secure application development practices.

3.4 Security Objective

The primary objective of AACL is to enforce the following property: for each sensitive action instance, there exists a finite language of acceptable request representations derived from the server’s state snapshot, and any request outside this language is rejected prior to execution.

Additionally, the accepted language is single-use and time-bounded, preventing replay under the defined threat model.

4 AACL Design Overview

4.1 Design Rationale

AACL extends language-theoretic principles to dynamic, stateful web application actions. Rather than validating requests solely through static schemas or token checks, AACL treats each sensitive action instance as a short-lived language whose accepted representations are defined by the server at runtime.

The core design principle is:

Every state-changing action must conform to a server-issued, ephemeral grammar that encodes structural, semantic, and temporal constraints derived from the current application state.

By shifting enforcement from token validation to language recognition, AACL aims to reduce replay and structural mutation attacks under the defined threat model.

4.2 Core Components

An AACL-enabled system consists of the following components:

1. **Intent Identifier.** Each sensitive action (e.g., login, password change, transfer) is modeled as an intent type. An intent type defines the general workflow and structural template for that category of operation.
2. **State Snapshot Extractor.** Before issuing a grammar, the server derives a state snapshot relevant to the intended action. This snapshot may include session context, user identity, authorization scope, account balances, allowed parameter ranges, and temporal constraints. This snapshot parameterizes the grammar instance.
3. **Ephemeral Grammar Generator.** Given an intent type and state snapshot, the server generates a grammar instance specific to that action occurrence. This grammar defines the allowed fields, constrains parameter types and ranges, encodes semantic bounds derived from state, is uniquely bound to the current action instance, is time-bounded, and is marked as unconsumed.
4. **Grammar Distribution Layer.** The generated grammar (or its encoded representation) is transmitted to the client in a protected form. The client uses it to construct the corresponding request. The grammar may be represented as a structured schema, a deterministic automaton, a signed policy object, or a compressed encoding recognized by the server. The client is not trusted; the grammar serves as a contract, not an enforcement authority.
5. **AACL Recognition Engine.** Upon receiving a request, the server invokes the AACL recognition engine. This engine identifies the corresponding grammar instance, parses the request against the grammar, verifies semantic constraints, and checks expiration and single-use status. Only if the request is accepted by the grammar and all associated constraints does execution proceed.
6. **Grammar Invalidation Module.** After successful execution (or expiration), the grammar instance is invalidated and cannot be reused. This enforces single-use semantics and mitigates replay within the threat model.

4.3 Action Lifecycle Under AACL

The lifecycle of a sensitive action proceeds as follows:

1. **Intent Initiation.** The client requests to perform a sensitive action.
2. **Grammar Issuance.** The server generates an ephemeral grammar instance bound to the current state snapshot and sends it to the client.
3. **Request Construction.** The client constructs a request intended to conform to the issued grammar.
4. **Recognition Phase.** The server parses the incoming request using the corresponding grammar instance.
5. **Validation & Execution.** If the request is accepted by the grammar and satisfies temporal and single-use constraints, the action is executed.
6. **Invalidation.** The grammar instance is marked consumed or expired.

Any request that does not conform structurally or semantically is rejected prior to business logic execution.

4.4 State Binding

Unlike static schemas, AACL grammars are parameterized by server-side state.

- In a financial transfer, the maximum allowable amount may be derived from the current balance.
- The authorized recipient list may be embedded into the grammar constraints.
- The grammar may encode specific field relationships valid only for that session state.

If the underlying state changes, previously issued grammars become invalid under the new state context.

4.5 Design Goals

AACL aims to achieve:

- Structural Integrity Enforcement
- Per-Instance Semantic Binding
- Single-Use Request Validity
- Replay Resistance under Defined Assumptions
- Compatibility with Existing Authentication Systems

AACL is designed as a complementary enforcement layer rather than a replacement for authentication, authorization, or secure coding practices.

5 Formal Model and Security Analysis

This section formalizes AACL as a state-bound, intent-specific language recognition system and analyzes its security guarantees under a defined threat model.

5.1 System Overview

Let:

- U be the set of authenticated users.
- I be the set of supported intents.
- S be the set of possible server states.
- R be the set of all possible request payloads.

For a user $u \in U$ attempting to execute intent $i \in I$, AACL constructs a per-intent grammar instance bound to a snapshot of server state.

Each protected request is therefore evaluated relative to:

$$(u, i, S_i)$$

where $S_i \in S$ denotes the state snapshot captured at grammar issuance time.

The goal of AACL is to ensure that only requests consistent with the issued grammar and associated snapshot are executed.

5.2 State Snapshot

For each issuance event, the server captures a snapshot:

$$S_i = \text{Snapshot}(u)$$

The snapshot includes all state elements relevant to validating intent i . Examples include:

- Authentication status
- User identifier
- Account balance
- Privilege level

The snapshot remains immutable for the lifetime of the grammar instance.

Semantic validation is performed relative to S_i , not the live mutable system state.

This prevents replay under changed contextual conditions.

5.3 Intent-Specific Alphabet

For each intent $i \in I$, define an alphabet:

$$\Sigma_i = \{k_1, k_2, \dots, k_n\}$$

where each k_j represents a required structural parameter key for that intent.

Example. For a transfer intent:

$$\Sigma_{\text{transfer}} = \{\text{grammar_id}, \text{intent}, \text{entropy}, \text{recipient_id}, \text{amount}\}$$

Only symbols from Σ_i are permitted in valid execution requests. No additional symbols are allowed.

5.4 Grammar as Deterministic Recognizer

For each issued intent instance, AACL constructs a deterministic recognizer:

$$A_i = (Q, \Sigma_i, \delta, q_0, F)$$

where:

- Q is a finite set of states.
- Σ_i is the intent-specific alphabet.
- $\delta : Q \times \Sigma_i \rightarrow Q$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of accepting states.

A request $w \in \Sigma_i^*$ is structurally valid if:

$$w \in L(A_i)$$

In the prototype realization, structural validity is also enforced through strict key-set equality:

$$\text{keys}(w) = \Sigma_i$$

This guarantees:

- No missing fields.
- No extra fields.
- No structural mutation.

Because A_i is deterministic, any deviation results in rejection.

5.5 Semantic Constraint Function

Structural validity alone does not guarantee correctness. Therefore, AACL defines a semantic constraint function:

$$C_i : \Sigma_i^* \times S_i \rightarrow \{\text{true}, \text{false}\}$$

This function evaluates semantic consistency between the request and the captured snapshot.

Examples.

- **Transfer intent:**

$$C_{\text{transfer}}(w, S_i) = (\text{amount}(w) \leq \text{balance}(S_i))$$

- **Password change intent:**

$$C_{\text{password}}(w, S_i) = (\text{new_password}(w) = \text{confirm_password}(w))$$

The function C_i is deterministic and bounded.

Semantic validation ensures that even structurally valid requests cannot violate contextual constraints.

5.6 Acceptance Condition

A request $w \in R$ is accepted if and only if all of the following hold:

$$\text{Accept}(w) = \left(w \in L(A_i) \wedge C_i(w, S_i) \wedge \neg \text{Used}_i \wedge t < T_i \right)$$

where:

- Used_i indicates whether the grammar instance has already been consumed,
- T_i is the expiration timestamp,
- t is the current time.

Thus, execution requires:

- Structural validity,
- Semantic validity,
- Single-use enforcement,
- Time-bound validity.

5.7 Security Property

We analyze AACL under the following threat model.

The attacker may:

- Observe previously issued requests.
- Replay captured requests.
- Modify request parameters arbitrarily.
- Attempt to inject additional parameters.

The attacker cannot:

- Forge server-issued grammar instances.
- Access server-side stored snapshot state.
- Bypass authentication.

Definition 1 (Intent Integrity). A request maintains intent integrity if it satisfies both structural validity and semantic consistency relative to the issued grammar instance.

Lemma 1 (Structural Non-Malleability). For any modified request $w' \neq w$, we have $w' \notin L(A_i)$ if its parameter structure deviates from Σ_i .

Proof sketch. Since A_i is deterministic, insertion, deletion, or substitution of structural elements produces a word outside the recognized language.

Lemma 2 (Snapshot-Bound Semantic Validity). For any replayed or contextually inconsistent request w' , we have $C_i(w', S_i) = \text{false}$ if it violates snapshot-bound constraints.

Theorem 1 (Single-Use Intent Integrity Under Threat Model). Under the defined threat model, for any modified or replayed request $w' \neq w$, at least one of the following holds:

$$w' \notin L(A_i) \text{ or } C_i(w', S_i) = \text{false} \text{ or } Used_i = \text{true}.$$

Therefore, no structurally or semantically altered request reaches business logic execution. This completes the formal and security foundation of AACL.

6 Implementation

This section describes the practical realization of AACL as a working web application prototype. The implementation demonstrates feasibility of per-intent grammar construction, state-bound validation, and single-use enforcement within a standard request-response architecture.

The prototype is implemented as a middleware-style security layer positioned between authentication and business logic execution.

6.1 Prototype Architecture

The AACL prototype consists of four primary layers:

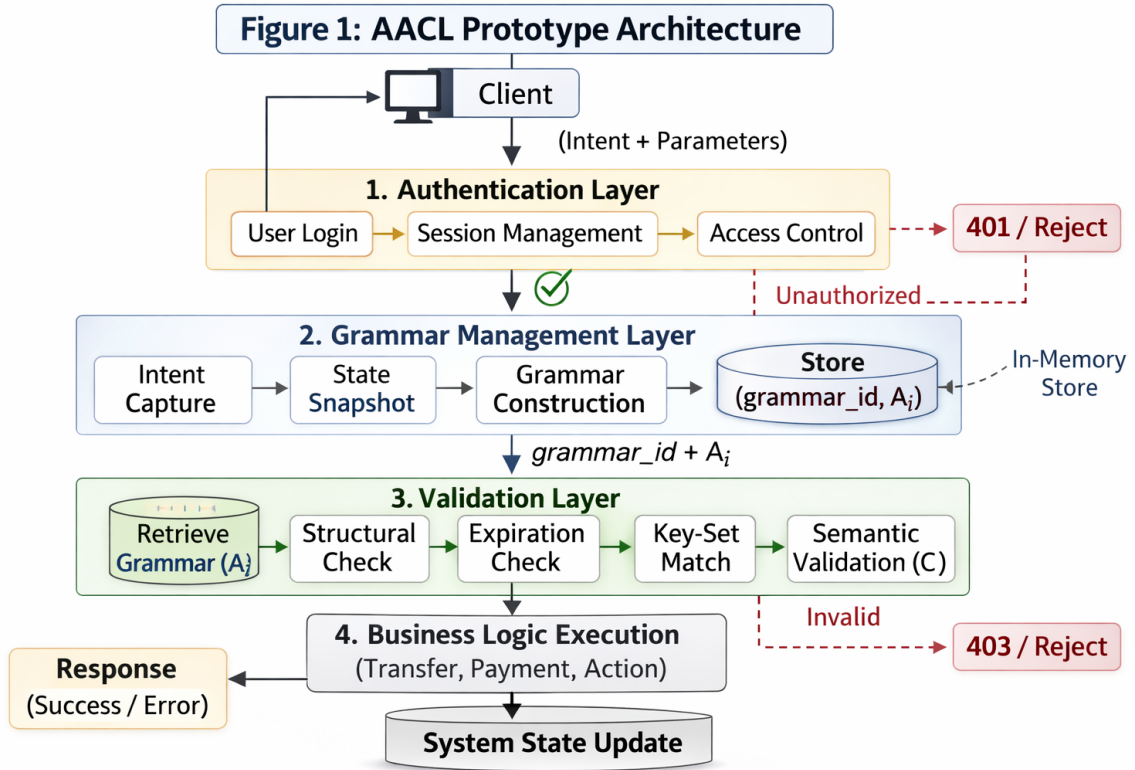


Figure 1: AACL prototype architecture.

1. Authentication Layer

User authentication is managed using session-based identity tracking. Each authenticated user is associated with a unique session identifier generated at login. AACL enforcement is applied

only after successful authentication — unauthenticated requests to grammar issuance or execution endpoints are rejected with HTTP 401 prior to any AACL processing.

Additionally, each issued grammar instance is bound to both the session identifier and the authenticated username at issuance time. Execution requests are verified against these bindings, preventing cross-session and cross-user grammar reuse.

2. Grammar Management Layer

This layer is responsible for:

- Capturing per-user state snapshots
- Generating unique grammar identifiers
- Generating per-instance entropy
- Locking intent parameters at issuance time
- Defining required structural keys
- Assigning expiration timestamps
- Storing grammar instances server-side

Each grammar instance contains:

- `grammar_id`
- `intent`
- `entropy`
- `required_keys`
- `expires_at`
- `used` flag
- `session_id` — binds grammar to the issuing session
- `username` — binds grammar to the authenticated user
- Associated snapshot state (balance, allowed recipients, locked amount, and locked recipient)

Grammar definitions are stored exclusively on the server. The client receives only the metadata necessary to construct a compliant request. The locked parameter values and snapshot state are never transmitted to the client, preventing grammar forgery or parameter substitution by the client.

3. Validation Layer

All protected endpoints route through a centralized AACL validation handler.

Validation enforces:

- Grammar lookup by `grammar_id`
- Session and user binding verification
- Expiration verification

- Single-use enforcement
- Intent match verification
- Entropy verification
- Structural key-set equality
- Semantic constraint evaluation — locked amount and recipient exact-match

Only if all validation stages succeed is the business logic executed. Upon successful execution, the grammar instance is immediately invalidated.

4. Client SDK Layer

A lightweight JavaScript SDK assists in:

- Requesting fresh grammar instances
- Transmitting intent parameters at issuance time
- Constructing payloads using required keys
- Preventing accidental omission of parameters
- Normalizing payload structure

The SDK improves usability but is not trusted for security enforcement. All security guarantees are enforced server-side.

6.2 Grammar Issuance Workflow

Grammar issuance is triggered when an authenticated client initiates a protected action.

Step 1: Intent Request and Parameter Commitment

The client sends an issuance request including the intended action parameters:

```
POST /aocl/issue/<intent>
```

The request body includes `amount` and `recipient_id`, which are locked into the grammar snapshot at issuance time. Any subsequent modification to these values in the execution request will be detected and rejected by the semantic constraint evaluation.

Step 2: State Snapshot Capture

The server captures a state snapshot bound to the authenticated user:

$$S_i = \text{Snapshot}(u)$$

The snapshot includes all state elements relevant to validating the requested intent, including the current account balance, the authorized recipient list, and the intent parameters committed in Step 1.

Step 3: Grammar Instance Generation

The server generates:

- Unique `grammar_id` (UUID4)
- Per-instance entropy value (`secrets.token_hex`)

- Required key list (Σ_i)
- Expiration timestamp
- `used` flag initialized to `false`
- `session_id` and `username` bindings
- Locked snapshot values: `locked_amount`, `locked_recipient`

The grammar instance is stored server-side and indexed by `grammar_id`.

Step 4: Metadata Transmission

The client receives:

- `grammar_id`
- `intent`
- `entropy`
- Expiration metadata
- Required key list

The authoritative grammar definition, snapshot state, and locked parameter values (locked amount and locked recipient) remain server-resident and are never transmitted to the client.

Figure 2: Grammar Issuance & Execution Workflow

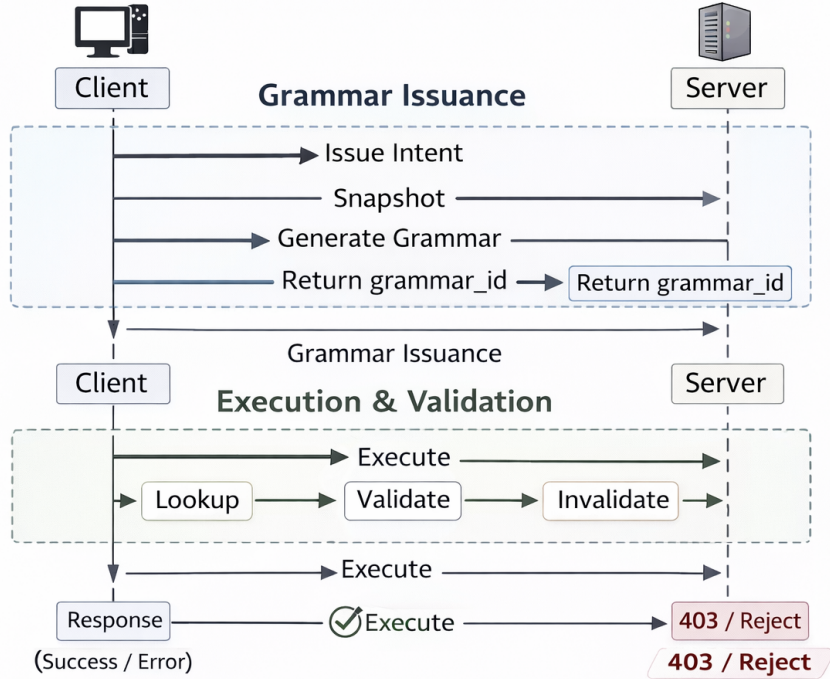


Figure 2: Grammar issuance workflow.

6.3 Execution Validation Workflow

When the client submits an execution request, the following validation pipeline is applied:

Step 1: Grammar Lookup

The server retrieves the grammar instance using `grammar_id`. If no matching grammar exists, the request is rejected with HTTP 403.

Step 2: Session and User Binding Verification

The `session_id` and `username` embedded in the grammar instance are compared against the current authenticated session. A mismatch indicates cross-session or cross-user reuse and results in rejection.

Step 3: Single-Use Check

If `used = true`, the request is rejected to prevent replay. This enforces the $\neg \text{Used}_i$ constraint of the acceptance condition.

Step 4: Expiration Check

The current timestamp t is compared against `expires_at`. Expired grammar instances are rejected, enforcing the $t < T_i$ constraint.

Step 5: Intent and Entropy Verification

The request must match:

- The issued `intent`
- The issued `entropy` value

Any mismatch results in rejection. The per-instance entropy value ensures that even structurally identical requests from different grammar instances are distinguishable.

Step 6: Structural Validation

The server verifies:

$$\text{keys}(w) = \Sigma_i$$

This ensures:

- No missing fields
- No additional fields
- No structural mutation

Because Σ_i is deterministic and fixed at issuance time, any insertion, deletion, or renaming of keys produces a word outside $L(\mathcal{A}_i)$ and is rejected.

Step 7: Semantic Validation

The semantic constraint function $C_i(w, S_i)$ is evaluated relative to the stored snapshot. For the transfer intent, this enforces:

- **Amount integrity:** $\text{amount}(w) = \text{locked_amount}(S_i)$ — the submitted amount must exactly match the value committed at issuance time.
- **Recipient integrity:** $\text{recipient_id}(w) = \text{locked_recipient}(S_i)$ — the submitted recipient must exactly match the value committed at issuance time.

- **Balance bound:** $\text{amount}(w) \leq \text{balance}(S_i)$ — the amount must not exceed the snapshot balance.

Note that while the formal model in Section 5 defines C_{transfer} as $\text{amount}(w) \leq \text{balance}(S_i)$, the prototype enforces a stricter exact-match constraint:

$$\text{amount}(w) = \text{locked_amount}(S_i)$$

where `locked_amount` is committed at grammar issuance time. This eliminates any ambiguity about the intended transfer value and strengthens tamper resistance beyond what the general formal model requires, directly preventing in-transit modification of the `amount` field by an intercepting adversary.

Any deviation from the locked values is treated as semantic mutation and results in rejection prior to business logic execution. This directly addresses the Burp Suite interception threat: any in-transit modification to `amount` or `recipient_id` produces a mismatch against the server-resident locked values and is deterministically rejected.

Step 8: Execute and Invalidate

Only if all preceding checks succeed does the request proceed to business logic execution. After successful execution, the grammar instance is marked as consumed:

```
used = true
```

This enforces single-use semantics and prevents replay under the defined threat model. The live user balance is updated post-execution, ensuring that subsequent grammar issuances reflect the current state.

6.4 Attack Simulation

To evaluate AACL’s mutation resistance, controlled attack simulations were conducted against the protected transfer intent using an automated benchmarking script.

The attacker was permitted to:

- Replay previously captured requests
- Modify `amount` after grammar issuance
- Modify `recipient_id` after grammar issuance
- Inject additional parameters
- Remove required parameters
- Alter entropy values

The attacker was not permitted to:

- Forge grammar instances
- Access server-side snapshot state
- Bypass authentication

The simulated attack categories and outcomes are summarized below:

- **Replay Attack** — rejected by single-use enforcement ($\neg \text{Used}_i$)
- **Amount Tampering** — rejected by locked amount exact-match
- **Recipient Tampering** — rejected by locked recipient exact-match
- **Structural Mutation** — rejected by key-set equality check

All tested mutation attempts were rejected deterministically before business logic execution. Rejection was confirmed across 4000 attack requests with zero bypasses observed, as reported in Section 7.

7 Experimental Evaluation

This section presents empirical measurements of the AACL prototype under controlled local testing conditions.

7.1 Experimental Setup

All experiments were conducted under the following environment:

- CPU: Apple M4 Chip
- RAM: 16 GB
- Operating System: MacOS Version 26.2 (Sequoia)
- Python Version: Python 3.13.7
- Backend Framework: Flask Version 3.1.2

The prototype was executed in a localhost environment. All latency measurements were recorded server-side using high-resolution timing. Latency measurements were collected using an automated HTTP benchmarking script issuing 1000 sequential requests per run. The experiment was repeated five times, resulting in a total of 5000 request executions. Statistical summaries were computed over the aggregated results.

7.2 Performance Evaluation

Category	Total	Blocked	Bypass Rate
Valid Requests	1000	—	—
Replay Attack	1000	1000	0.0%
Amount Tampering	1000	1000	0.0%
Recipient Tampering	1000	1000	0.0%
Structural Mutation	1000	1000	0.0%
Total (attacks only)	4000	4000	0.0%

Table 2: Attack resistance over 5000 AACL-protected requests.

All 4000 attack requests were rejected prior to business logic execution. Zero bypasses were observed across all attack categories. This confirms the empirical soundness of the ACL enforcement pipeline under the defined threat model.

Notably, recipient tampering attempts included both valid-format recipients (e.g., redirecting to another allowed user) and malformed inputs including empty strings, SQL injection patterns, and path traversal strings — all of which were rejected deterministically by the locked-recipient constraint enforced at grammar issuance time.

Operation	Mean (ms)	Std Dev (ms)
Grammar Issuance	0.93	0.18
Validation (valid)	0.66	0.26
Validation (invalid)	0.78	0.24
Baseline	0.66	0.26

Table 3: Latency summary over 5000 runs.

7.2.1 Grammar Issuance Latency

Grammar issuance includes:

- Authentication and session verification
- Per-user state snapshot capture (balance, allowed recipients)
- Intent parameter locking (amount, `recipient_id`)
- Unique grammar ID generation (UUID4)
- Per-instance entropy generation (`secrets.token_hex`)
- Expiration timestamp assignment
- In-memory grammar storage

Measured over 5000 runs:

- Mean issuance latency: 0.93 ms
- Standard deviation: 0.18 ms

The low standard deviation reflects the deterministic nature of snapshot capture and entropy generation under single-process localhost conditions.

7.2.2 Execution Validation Latency

Execution validation includes:

- Grammar lookup by `grammar_id`
- Session and user binding verification

- Single-use enforcement ($\neg \text{Used}_i$)
- Expiration check ($t < T_i$)
- Intent and entropy verification
- Structural key-set equality check ($\text{keys}(w) = \Sigma_i$)
- Semantic constraint evaluation — locked amount and recipient exact-match
- Grammar invalidation on success

Measured over 5000 runs:

- Mean validation latency (valid request): 0.66 ms
- Mean validation latency (invalid request): 0.78 ms
- Standard deviation: approximately 0.25 ms (across both)

Counterintuitively, invalid requests exhibit marginally higher mean latency than valid ones. This is because invalid requests in the attack categories (amount tampering, recipient tampering) reach the semantic validation stage before rejection, while structurally mutated requests terminate earlier at the key-set equality check. The aggregated mean across all attack categories is 0.78 ms, slightly above the valid execution mean of 0.66 ms, reflecting this variable early-termination behavior.

7.2.3 Baseline Comparison

Baseline measurement was taken by executing the same business logic without AACL validation.

- Baseline execution latency: 0.66 ms mean (0.26 ms standard deviation)
- AACL-protected execution latency: 0.66 ms mean (validation only)
- Grammar issuance latency: 0.93 ms mean

The total per-interaction overhead introduced by AACL — accounting for both grammar issuance and validation — is approximately 0.93 ms above the baseline. This represents a modest and practically negligible overhead for sensitive, low-frequency operations such as financial transfers, password changes, or administrative actions, which are the primary intended use cases for AACL enforcement.

Compared to the baseline, the AACL-protected execution path adds approximately 41% relative overhead in absolute terms — however, at sub-millisecond absolute values, this is well within acceptable bounds for interactive web applications.

7.3 Complexity Consistency

Structural validation requires equality comparison over the request key set, yielding theoretical complexity:

$$T(n) = O(n)$$

where n is the number of request parameters.

This linear complexity arises because the validation process compares the incoming key set with the grammar-defined key set Σ_i via a single set equality operation.

Semantic validation (locked amount and recipient comparison) operates in $O(1)$, as it involves only direct equality checks against snapshot-bound values.

Empirical measurements show approximately linear growth in validation time as request parameter count increases, consistent with the theoretical structural validation complexity. No super-linear growth behavior was observed within the tested parameter range.

Figure 3 illustrates the mean latency and standard deviation for each AACL operation measured over 5000 requests. Grammar issuance exhibits the highest latency due to snapshot capture, entropy generation, and intent parameter locking. Validation of valid requests matches baseline latency closely, confirming that AACL’s enforcement overhead is concentrated at the issuance stage rather than at execution time. Invalid requests are marginally slower due to reaching deeper validation stages before rejection.

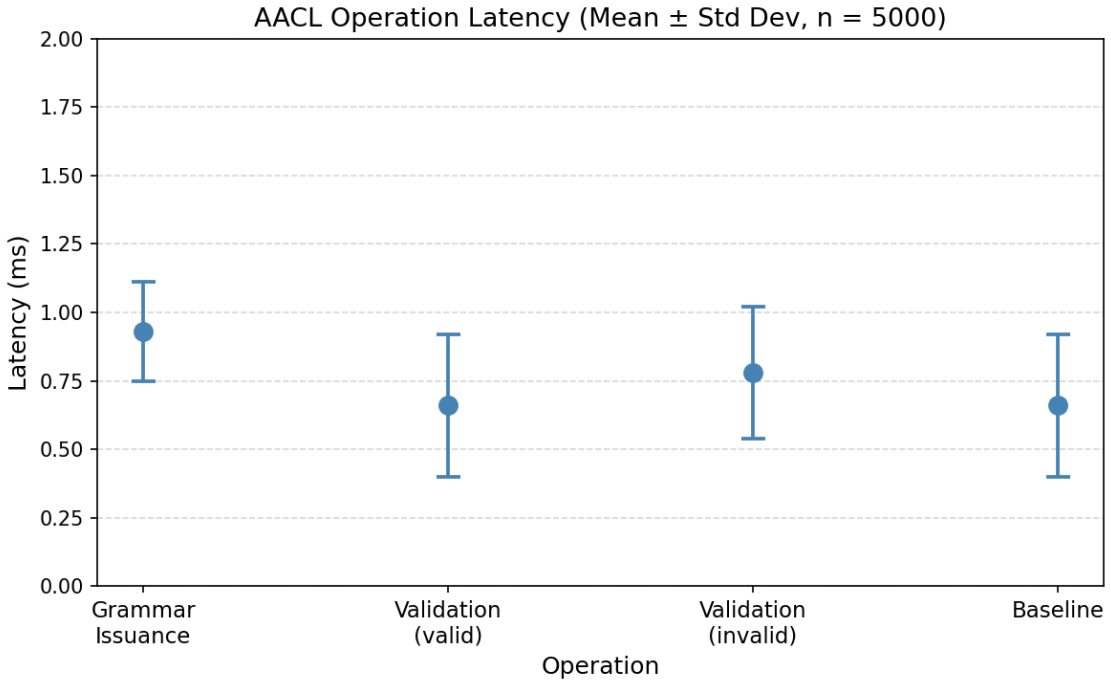


Figure 3: AACL operation latency (Mean \pm Std Dev, $n = 5000$). Grammar issuance includes snapshot capture, entropy generation, and intent locking. Baseline reflects raw business logic execution without AACL validation.

8 Discussion

The AACL framework demonstrates a practical extension of Language-Theoretic Security (LangSec) principles to the dynamic, stateful context of modern web applications. By issuing ephemeral, intent-bound grammars per sensitive action instance, AACL enforces not only syntactic conformance but also semantic single-use intent and state consistency at the language-recognition layer. This addresses a key gap identified in Section 2: while existing defenses (CSRF tokens, JWTs, nonces, WAFs, static schemas) provide authenticity, integrity, or coarse validation, they rarely capture the precise structural and temporal contract of an individual action instance. The prototype results in Section 7 confirm that this additional enforcement layer introduces acceptable overhead ($\sim 41\%$

relative overhead, under 1 ms absolute, on low-frequency operations) while enabling robust rejection of replays and structural mutations.

The formal model (Section 5) and prototype implementation highlight AACL’s strengths under the defined threat model. Ephemerality combined with server-side state snapshots (e.g., balances, session nonces) ensures that once a grammar is used or expired, any resubmitted or tampered request becomes unrecognizable—even if it passes superficial checks like signature verification or type validation. This shifts security from “accept known-bad patterns” (signature-based WAFs) or “trust the application logic” (static schemas) to “reject anything not precisely matching the current intent-bound language.” In experiments, validation remained linear in request size ($O(n)$ key-set comparison), with latencies under 1 ms for typical parameter counts, suggesting scalability for production use in high-value endpoints.

However, several limitations and trade-offs warrant consideration. First, the current prototype relies on simple structural checks (key-set equality, basic semantic predicates) rather than full DFA/NFA parsing for complex grammars. While sufficient for JSON-like payloads in web APIs, richer grammars (e.g., supporting regex-constrained fields or nested dependencies) could increase validation time and complexity—potentially pushing overhead beyond acceptable thresholds for high-throughput endpoints. Future implementations might explore lightweight parser generators (e.g., based on restricted context-free grammars) to balance expressiveness and performance.

Second, AACL introduces server-side state management overhead for grammar storage and invalidation. In the in-memory prototype, this is negligible, but in distributed deployments (e.g., multiple replicas), consistent invalidation requires mechanisms like Redis TTLs or distributed caches. Failure to synchronize could allow window-of-opportunity replays across instances. Additionally, grammar issuance adds a round-trip implication for client-server interactions: clients must fetch the ephemeral grammar (or a compact representation) before submitting the action request. This suits interactive flows (e.g., form submissions with pre-issued tokens) but may complicate purely API-driven or asynchronous scenarios unless bundled with prior responses.

Third, while AACL mitigates a class of replay and mutation attacks, it complements rather than replaces upstream controls. It assumes authenticated sessions and does not address client-side tampering if the attacker bypasses the legitimate client (e.g., via direct API calls). Integration with existing defenses—such as rate limiting, anomaly detection, or zero-trust architectures [31]—remains essential for defense-in-depth.

Looking ahead, AACL opens several promising directions. Extensions could include adaptive grammars that incorporate runtime context (e.g., user risk scores, behavioral patterns) or support for more expressive constraints via dependent types or monadic bindings in the grammar definition. Integrating AACL with emerging paradigms like capability-based security or language-based information flow control could further harden dynamic web flows. Empirical studies on larger-scale deployments (e.g., banking or e-commerce platforms) would validate overhead in real traffic and explore attack surfaces introduced by grammar management itself.

In summary, AACL advances LangSec toward state-aware, intent-explicit enforcement in web applications, reducing reliance on ad-hoc application-layer checks. While not a universal panacea, it provides a principled, low-overhead layer that meaningfully narrows the attack surface for sensitive operations—bridging the divide between static protocol security and dynamic, contextual web interactions.

9 Limitations

While AACL provides a principled, state-aware enforcement layer for sensitive web actions through ephemeral, intent-bound grammars, several limitations must be acknowledged.

1. **Reliance on upstream security controls** AACL assumes correctly implemented authentication, secure transport (e.g., TLS), and uncompromised server infrastructure. It does not mitigate session hijacking, credential theft, or transport-layer compromise. An attacker possessing a valid authenticated session could request and use new grammar instances to execute permitted actions. AACL enforces structural and semantic intent integrity only within authenticated contexts and does not substitute for identity verification or access control mechanisms.
2. **Prototype state management constraints** The current prototype uses in-memory grammar storage within a single process. While concurrent replay experiments confirmed effective single-use enforcement, distributed or multi-worker deployments require atomic invalidation across replicas to prevent race conditions. Without synchronization mechanisms (e.g., distributed locks, transactional caches, or atomic check-and-delete with TTLs), narrow replay windows could emerge during grammar consumption.
3. **Snapshot-based semantics** Semantic constraints are evaluated against the issuance-time state snapshot S_i , ensuring deterministic language recognition. However, in systems with rapidly mutating state (e.g., concurrent financial updates), the snapshot may diverge from live state by execution time, potentially allowing semantic drift. The prototype implements strict snapshot-bound validation; high-consistency environments may require supplementary live revalidation before commit.
4. **Limited grammar expressiveness** Structural validation relies on key-set equality and simple bounded predicates rather than full DFA/NFA-based parsing of complex or nested grammars. While adequate for typical JSON payloads in web APIs, more expressive grammars (e.g., supporting regex constraints, conditional fields, or dependencies) would increase validation overhead and implementation complexity. The performance implications of richer grammars under production loads remain unexamined.
5. **Additional deployment overhead** Grammar issuance, storage, expiration, and invalidation introduce server-side state and potentially an extra client-server interaction step. Although prototype latencies stayed sub-millisecond, real-world behavior in high-throughput, globally distributed, or serverless environments requires further measurement and optimization.
6. **Complementary—not comprehensive—nature** AACL does not address vulnerabilities stemming from flawed business logic, improper authorization, injection flaws, or unsafe output encoding. It narrows the attack surface for replay and structural mutation under its threat model but must be layered with existing secure coding practices, WAFs, and authorization checks.

These limitations position AACL as a targeted, complementary mechanism for strengthening intent integrity in stateful web interactions rather than a universal security solution.

10 Future Work

The AACL framework establishes a DFA-driven, snapshot-bound recognition model (Section 5) for enforcing per-intent structural and semantic integrity. Building on this foundation, several avenues for future research remain.

10.1 Expanded Grammar Expressiveness

The current prototype enforces structural validity using deterministic key-set equality and bounded semantic predicates. While sufficient for JSON-based API payloads, richer grammar models could extend expressiveness while preserving formal guarantees. Future work may explore restricted context-free grammars, parameterized automata, or formally verified parser generators such as Hammer and derivative-based parsing frameworks frequently discussed within Language-Theoretic Security literature.

A key research question concerns maintaining deterministic recognition properties while supporting nested structures, conditional parameters, and inter-field dependencies. Any extension must preserve the predictability and bounded validation cost established in Section 7, ensuring that expressiveness does not compromise practical deployability.

10.2 Live-State and Hybrid Consistency Models

Section 8 discussed the distinction between snapshot-bound validation and hybrid snapshot-plus-live revalidation. Future work could formally model this hybrid approach by defining a two-phase acceptance condition:

1. Recognition under the issuance-time snapshot S_i , and
2. Commit-time verification against live state.

Such a model would preserve deterministic language recognition while strengthening consistency guarantees in high-concurrency systems. Formalizing these hybrid semantics may provide stronger safety guarantees for financial or transactional environments without sacrificing the clarity of the DFA-based enforcement defined in Section 5.

10.3 Distributed and Scalable Deployment

While the prototype demonstrates single-use enforcement in a single-process environment, distributed deployment introduces synchronization challenges. Future research should explore atomic invalidation strategies such as transactional check-and-delete operations, short TTL-backed grammar storage, and consensus-backed distributed caches.

Formal modeling of replay resistance under distributed failure scenarios—such as delayed invalidation or partial replication—would strengthen AACL’s applicability to multi-region or high-availability systems. Ensuring preservation of the $\neg Used_i$ invariant across replicas remains an important systems research question.

10.4 Adaptive and Context-Aware Grammars

A promising direction involves adaptive grammar generation informed by contextual risk signals. Grammars could dynamically tighten constraints under elevated risk conditions (e.g., anomalous geolocation, device change, or behavioral deviation). Integration with machine-learning-based

anomaly detection systems may allow grammars to encode risk-weighted parameter bounds or shortened validity windows.

This aligns AACL with emerging adaptive security paradigms, potentially transforming grammars from static per-intent contracts into dynamically risk-calibrated enforcement artifacts.

10.5 Formal Verification and Tooling

Further work could formalize AACL’s recognition and acceptance properties within proof assistants such as Coq or Isabelle. Mechanized proofs could verify invariants such as structural non-malleability and single-use intent integrity under explicit threat assumptions.

In parallel, practical developer tooling—such as automatic grammar derivation from endpoint specifications, type annotations, or OpenAPI definitions—would lower integration barriers and reduce the risk of misconfiguration.

10.6 Empirical Evaluation Under Realistic Attack Campaigns

The current evaluation focuses on controlled replay and mutation simulations. Future empirical studies should examine AACL under sustained adversarial campaigns, including automated mutation fuzzing, replay floods, and parameter permutation attacks. Measuring attack surface reduction in realistic environments would provide quantitative validation of the “intent integrity” property defined in Section 5.

Large-scale deployment studies in domains such as fintech, e-commerce, or administrative platforms would further clarify operational overhead, developer ergonomics, and real-world resilience.

By extending formal rigor, enhancing grammar expressiveness, strengthening distributed guarantees, and validating performance under adversarial conditions, future work can mature AACL from a prototype framework into a broadly deployable state-aware security primitive for web applications.

11 Conclusion

This paper presented AACL (Artificial Adaptive Control Language), a state-aware, intent-bound enforcement framework that extends Language-Theoretic Security (LangSec) principles to dynamic web application workflows. Unlike conventional mechanisms—such as CSRF tokens, JWTs, nonces, and WAFs—that primarily verify authenticity or coarse integrity, AACL models each sensitive action instance as a uniquely issued, ephemeral grammar bound to a server-side state snapshot. By requiring requests to undergo structural recognition (e.g., DFA-driven parsing) and semantic validation under the corresponding grammar before execution, AACL elevates explicit intent integrity to a first-class enforcement property.

We formalized the AACL model, defining per-instance grammars, acceptance conditions, lifecycle constraints (including single-use invalidation), and associated security properties. The prototype, integrated into a Flask-based web application, demonstrated practical feasibility, with empirical measurements confirming linear validation complexity ($O(n)$ in request parameters) and modest latency overhead suitable for low-frequency, high-value operations. These results indicate that dynamic, state-bound language recognition can serve as an effective complementary layer to existing authentication, authorization, and transport-level protections. While AACL does not supplant secure system design or mitigate upstream vulnerabilities (e.g., session hijacking), it meaningfully narrows the attack surface by rejecting any request that deviates from precisely intended, context-consistent

interactions—thereby mitigating classes of replay and structural mutation attacks under the defined threat model.

By bridging formal language recognition with stateful web enforcement, AACL contributes to a more principled and explicit approach to application-layer security. This work opens promising avenues for further exploration in scalable deployments, richer grammar expressiveness, adaptive/context-aware extensions, and formally verified implementations—advancing LangSec toward practical, verifiable security in modern, dynamic information systems.

12 References

1. J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems,” *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975. DOI: 10.1109/PROC.1975.9939.
2. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed., Hoboken, NJ, USA: Wiley, 2008.
3. L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, “The Science of Insecurity,” in *LangSec Workshop*, 2011.
4. M. Samuel, S. Bratus, and M. E. Locasto, “Security Applications of Formal Language Theory,” Dartmouth College Tech. Rep. TR2011-709, 2011.
5. T. Bratus, A. Shubina, and M. E. Locasto, “The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them,” in *IEEE Security Development Conf. (SecDev)*, 2016.
6. E. Poll, “LangSec Revisited: Input Security Flaws of the Second Kind,” in *IEEE Security & Privacy Workshops (SPW)*, 2018.
7. E. Jaeger, S. Bratus, and M. E. Locasto, “Mind Your Language(s): A Discussion About Languages and Security,” in *LangSec Workshop*, 2014.
8. P. Anantharaman, S. Bratus, and M. E. Locasto, “Building Hardened Internet-of-Things Clients with Language-Theoretic Security,” in *LangSec Workshop*, 2017.
9. Ninth IEEE Security & Privacy Workshop on Language-Theoretic Security, IEEE S&P Workshops, May 25, 2023; workshop program available online.
10. D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, “Towards a Formal Foundation of Web Security,” in *Proc. IEEE Computer Security Foundations Symposium (CSF)*, 2010.
11. A. Barth, C. Jackson, and J. C. Mitchell, “Robust Defenses for Cross-Site Request Forgery,” in *Proc. ACM Conf. Computer and Commun. Security (CCS)*, 2008.
12. A. Guha, S. Krishnamurthi, and T. Jim, “Using Static Analysis for Ajax Intrusion Detection,” in *Proc. Int. World Wide Web Conference (WWW)*, 2009.
13. A. Sabelfeld and A. C. Myers, “Language-Based Information-Flow Security,” *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.
14. C. Fournet and A. D. Gordon, “Stack Inspection: Theory and Variants,” *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 3, pp. 360–399, 2003.

15. T. R. Oprea et al., “An Analysis of CSRF Defenses in Web Frameworks,” in *Proc. Int. Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.
16. “OWASP Top 10: The Ten Most Critical Web Application Security Risks,” Open Web Application Security Project, 2021.
17. J. R. Tadhani, V. Vekariya, V. Sorathiya, et al., “Securing Web Applications Against XSS and SQLi Attacks Using a Novel Deep Learning Approach,” *Sci. Rep.*, vol. 14, article 1803, Jan. 2024. doi:10.1038/s41598-023-48845-4.
18. O. Tsai, J. Li, T. T. Cheung, et al., “GraphQLer: Enhancing GraphQL Security with Context-Aware API Testing,” arXiv preprint, Apr. 2025.
19. I. Perera, H. Abeyrathne, S. Malalgoda, and A. Ifthikar, “Enhancing GraphQL Security by Detecting Malicious Queries Using Large Language Models and CNNs,” arXiv preprint, Aug. 2025.
20. K. Abdulghaffar, N. Elmrabit, and M. Yousefi, “Enhancing Web Application Security through Automated Penetration Testing with Multiple Vulnerability Scanners,” *Computers*, vol. 12, no. 11, article 235, Nov. 2023.
21. M. Nawrocki and J. Kołodziej, “Vulnerabilities of Web Applications: Good Practices and New Trends,” *Appl. Cybersecurity & Internet Governance*, vol. 3, no. 2, Dec. 2024. doi:10.60097/ACIG/199521.
22. D. Hyka et al., “Enhanced Web Application Security through Advanced Penetration Testing Techniques,” *Smart Cities and Regional Development Preprints*, vol. 2, no. 1, Jun. 2025.
23. “Comparative Evaluation of Approaches & Tools for Effective Security Testing of Web Applications,” *PeerJ Comput. Sci.*, 2023 (tool effectiveness study).
24. F. B. Schneider, “Enforceable Security Policies,” *ACM Trans. Inf. Syst. Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.
25. R. Milner, *Communicating and Mobile Systems: The Pi Calculus*, Cambridge, U.K.: Cambridge Univ. Press, 1999.
26. G. Lowe, “Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR,” in *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1996.
27. B. Blanchet, “Automatic Verification of Security Protocols in the Applied Pi Calculus,” in *Proc. IEEE Computer Security Foundations Workshop*, 2001.
28. M. Jones, J. Bradley, and N. Sakimura, “RFC 7519: JSON Web Token (JWT),” IETF, May 2015.
29. D. Hardt, “RFC 6749: The OAuth 2.0 Authorization Framework,” IETF, Oct. 2012.
30. T. Lodderstedt, M. McGloin, and P. Hunt, “RFC 6819: OAuth 2.0 Threat Model and Security Considerations,” IETF, Jan. 2013.
31. NIST, “SP 800-207: Zero Trust Architecture,” National Institute of Standards and Technology, Aug. 2020.

- 32. E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, “Noxes: A client-side solution for mitigating cross-site scripting attacks,” in *Proc. ACM Symp. Applied Computing*, 2007, pp. 330–337.
- 33. D. Song, D. Brumley, and H. Yin, “Automatic discovery of vulnerabilities in binary programs,” in *Proc. IEEE Symp. Security and Privacy*, 2008, pp. 3–17.
- 34. B. Livshits and M. S. Lam, “Finding security vulnerabilities in Java applications with static analysis,” in *Proc. USENIX Security Symposium*, 2005.
- 35. W. Halfond, J. Viegas, and A. Orso, “A classification of SQL-injection attacks and counter-measures,” in *Proc. IEEE Int. Symp. Secure Software Engineering*, 2006.
- 36. M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*. San Francisco, CA, USA: No Starch Press, 2011.
- 37. E. Z. Yang, “Principles of secure software design,” *ACM Queue*, vol. 15, no. 2, pp. 40–56, 2017.
- 38. T. Close, M. S. Miller, and J. S. Shapiro, “Capability-based security for web applications,” in *Proc. USENIX Security Symposium*, 2013.
- 39. S. R. Choudhary, H. Deshpande, and A. Orso, “Crawling modern web applications for vulnerability detection,” in *Proc. IEEE Int. Conf. Software Engineering (ICSE)*, 2013.