Work-in-progress: Assertive Trace

Shun Kashiwa
University of California San Diego
skashiwa@ucsd.edu

Michael Coblenz
University of California San Diego
mcoblenz@ucsd.edu

Deian Stefan
University of California San Diego
dstefan@ucsd.edu

***Abstract*—We present *Assertive Trace*, a policy enforcement framework that leverages server-side application traces as the substrate for specifying and enforcing security policies. Web applications today instrument their behavior using observability frameworks, such as OpenTelemetry, to help diagnose failures and monitor performance by emitting detailed traces that capture the app's sequences of operations and their attributes. In practice, these traces should—and typically *do*—contain enough information for developers to catch many security and correctness violations. We built the Assertive Trace framework following this insight; Assertive Trace allows developers to specify expected patterns of application behavior as *policies* over traces. The framework then enforces these policies at runtime non-intrusively, i.e., with minimal changes to application code, by analyzing traces on the fly. We describe the design and implementation of our framework and demonstrate its applicability through case studies.**

I. Introduction

Access control failures in Web applications are *the* most common and severe class of security vulnerabilities. The OWASP Top Ten has ranked *Broken Access Control* as the most critical risk in its 2021 awareness document [1], a position Broken Access Control continues to hold today [2]. We see a similar pattern in vulnerability-centric datasets: in the 2025 CWE Top 25 Most Dangerous Software Weaknesses [3]—derived from 39,080 CVE records and ranked using a composite of frequency and severity—*Missing Authorization* ranks fourth overall, with closely related weaknesses such as *Improper Access Control* and *Missing Authentication for Critical Function* also appearing in the Top 25.

This is not surprising: developers today largely sprinkle access control checks alongside their application logic—and inevitably this ad-hoc approach to "policy specification and enforcement" fails as applications evolve and grow in complexity. Unfortunately, principled Web frameworks designed to address these vulnerabilities—e.g., by separating policy specification from business logic, and enforcing these policies throughout the system [4]–[7]—have seen little to no adoption in practice. This is largely because these frameworks are *intrusive*—they require developers to rewrite their applications in a specific programming language, using a particular framework and its design decisions. For example, Hails [4] requires developers to write their application in Haskell, atop MongoDB, and using a custom standard library. In practice this is a significant barrier to adoption.

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-06-0
https://dx.doi.org/10.14722/madweb.2026.23036
www.ndss-symposium.org

And, in practice, it's not necessary. Modern applications are heavily instrumented for observability and produce rich traces of runtime events—they capture which user performed which operation, what resources were accessed, and in what order. In other words, even though today these traces are largely passive artifacts (e.g., used for performance tuning, debugging, and incident response), they contain sufficient information that can be used to enforce access control.

We distill this insight in *Assertive Trace*, a new policy enforcement framework that uses observability traces—specifically OpenTelemetry traces—as the substrate for both writing and enforcing policies. With Assertive Trace, developers write policies as expected sequences of OpenTelemetry events—e.g., "authentication must precede database writes" and "password changes must trigger email notifications." Assertive Trace then enforces these policies at runtime by ensuring the application execution, i.e., its trace, abides by the policy.

We implement Assertive Trace as a custom OpenTelemetry TypeScript SDK. Our library is a drop-in replacement for the official OpenTelemetry SDK, i.e., it doesn't require developers to modify their application code, but additionally allows developers to associate policies with *spans*, the building blocks (e.g., "database write") that constitute a trace. This approach allows developers of already-instrumented applications to specify and enforce policies without intrusive changes, i.e., without modifying their application code, database system, middleware, etc. Our preliminary evaluation on several case studies, spanning role-based and break-the-glass access controls and auto-instrumented cloud services, shows that Assertive Trace is non-intrusive, easy to integrate, and expressive enough to not only capture fine-grained access control policies but also correctness properties.

Our contributions are as follows:

- ▶ We present *Assertive Trace*, a novel framework for specifying and enforcing security policies using traces.
- ▶ We design a policy language for expressing temporal and attribute-based constraints over OpenTelemetry traces.
- ▶ We implement a prototype in TypeScript that enforces policies, which is built as a custom OpenTelemetry SDK, requiring minimal changes to the codebase.
- ▶ We evaluate our approach through case studies, demonstrating its effectiveness in capturing and enforcing important security properties.

```
app.post("/reset-password", async ({ input }) => {
  const { email, currentPassword, newPassword } = input;
  // Step 1: Verify current password
  const result = await verifyUserPassword(email,
    currentPassword);
  if (!result.success)
    throw new Error("Current password is incorrect");
  // Step 2: Update password in database
  const hashedPassword = await hashPassword(newPassword);
  await updateUser(email, { hashedPassword });
  // Step 3: Send notification email
  await sendPasswordChangedEmail(email);
  return { success: true };
});
```

Fig. 1. Server-side password reset endpoint implementation.

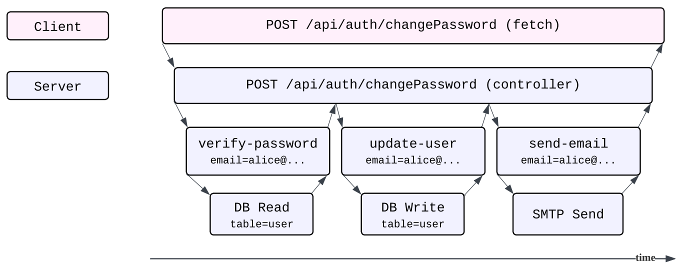

Fig. 2. Distributed trace of a password reset. The root span originates at the client, with child spans on the server representing request handling and downstream operations. Each span carries attributes that record structured information about the operation.

II. A Tour of Policy Enforcement with Assertive Trace

In this section, we give a tour of Assertive Trace by showing how modern web applications are typically instrumented and how Assertive Trace can be used to define and enforce policies on top of existing instrumentation.

A. *Example: Password Change Endpoint*

Consider a password-change endpoint in a typical web application. This endpoint must:

1) Accept the user's email, current, and new passwords.
2) Verify the current email-password pair is correct.
3) Update the password in the `User` table.
4) Send a confirmation email notifying the user that the password was changed.

A straightforward server-side implementation might look like Figure 1. But even such a simple endpoint surfaces non-trivial invariants. For example, the user whose password is updated must be exactly the user whose password was verified, and every successful password update must be followed by a notification email. Alas, these invariants are encoded implicitly in control flow and function calls, which makes them both difficult to audit and easy to violate as the code evolves.

B. *How Developers Instrument and Use Traces Today*

Traces are OpenTelemetry observability signals that give developers insight into the flow of program execution. A *trace* represents a single end-to-end execution of an operation and is composed of one or more *spans*. Each span represents a specific operation—handling an HTTP request, executing a database query, or calling a downstream service—and is related to other spans through parent-child relationships. Together, these spans form a tree that reflects the causal flow of execution.

Figure 2 shows an example distributed trace for an invocation of the password reset endpoint shown in Figure 1. The trace begins on the client when it initiates an HTTP request to change the user's password. This creates the *root span* of the trace. When the request is sent, tracing context is propagated to the server using standard mechanisms such as W3C Trace Context [8], linking the client and server spans. Upon receiving the request, the server creates a span for handling `POST /reset-password`, which becomes a child of the client's root span. As execution proceeds, additional child spans are created for operations such as `verifyUserPassword`, `updateUser`, and `sendPasswordChangedEmail`. Each of these operations may themselves spawn further child spans for internal activities. This hierarchical structure makes it possible to see not only *what* operations occurred, but also *where* they occurred and *how* they relate to one another.

Spans can carry *attributes*, which are key-value pairs tied to operations. In our example, attributes may include the email address of the user requesting the password reset, the database table updated by `updateUser`, or the destination endpoint of the outbound email service call. To support consistent analysis across systems, OpenTelemetry defines *Semantic Conventions*, which standardize attribute schemas for common protocols and services such as HTTP, SQL, Object Store, and generative AI.

In practice, developers and operators primarily use distributed traces *post hoc*. They inspect traces to debug failures, diagnose regressions (e.g., by identifying the operation that dominates end-to-end latency), or answer retrospective audit questions such as "who initiated this password change, and which services were involved?" They don't use traces to enforce invariants—OpenTelemetry doesn't have a way to express or enforce invariants as executable policies.

C. *Our Observation: Traces as a Policy Substrate*

The key observation behind this work is that traces for individual requests already capture the sequence of semantically meaningful actions we care about: which user was authenticated, which records were read or written, which external systems were called, and in what order. We can thus write policies over the *trace* of an execution.

Concretely, the trace in Figure 2 contains enough information to express high-level policies such as:

- ▶ A successful password verification span must precede any database span that updates that user's password.
- ▶ A successful password update must be followed by a span that sends a password-change email to the same user.

If we can specify such expected patterns of spans, and automatically check them against the trace as it is produced, then we can turn existing observability infrastructure into a vehicle for *enforcement*, not just post-hoc analysis.

```
1  const resetPasswordPolicy = seq([
2      span({
3          name: "verify-user-password",
4          email: str().bind({ verifiedEmail: self() }),
5      }, {
6          success: true,
7      }),
8      span({
9          name: "update-user",
10         email: get("verifiedEmail"),
11         fields: ["hashedPassword"],
12     }),
13     span({
14         name: "send-password-changed",
15         email: get("verifiedEmail"),
16     }),
17 ]);
```

Fig. 3. Simplified Assertive Trace policy for password reset endpoint. It specifies that a successful password verification, followed by a password update and a notification email, must occur in order. The email used in all three operations must be the same.

D. Our Approach: Assertive Trace

With Assertive Trace, developers can turn existing traces into an enforcement mechanism by:

1) **Replace the OpenTelemetry SDK** with Assertive Trace's policy-aware SDK. This one-line change preserves existing OpenTelemetry functionality while enabling policy enforcement.
2) **Write a policy over spans** using Assertive Trace's policy language, which treats spans and their attributes as the primitive vocabulary. Policies describe admissible sequences of spans and constraints over their attributes, such as data types, values, and dependencies across operations. Figure 3 shows a simplified policy for the password reset endpoint.
3) **Attach the policy to the request**. Spans produced while handling the request are checked against it.

As spans are produced during execution, Assertive Trace checks them against the attached policy. If spans appear in an unexpected order or with attributes that violate the policy, the evaluation engine throws an error. In the context of an HTTP request, this error aborts the request and immediately returns an HTTP error response to the client, preventing the unintended behavior. Developers can also customize the error handling logic—e.g., create an alert if the request handler span ended without sending the notification email.

III. Design and Implementation

At a high level, Assertive Trace consists of three main components: the policy language, the runtime checker, and the OpenTelemetry integration. In this section we describe these components—and our TypeScript implementation.[1]

A. Policy DSL

We designed an embedded domain-specific language (eDSL) in TypeScript for specifying security policies over OpenTelemetry traces. Each policy is defined by combining five primitive constructs: `span`, `event`, `seq`, `alt`, and `rep`.

`span` specifies constraints on individual spans, including attributes. Since each span and its attributes are represented as key-value pairs in OpenTelemetry, we need a way to express constraints on such key-value pairs. For this purpose, we use a popular schema library Zod [9] to define and validate span attributes. [2] Zod schemas allow us to express complex constraints on span attributes, such as types, ranges, and other properties. For example, lines 3 and 4 in Figure 3 mandate a `verify-user-password` span with string attribute `email`. Because spans may acquire attributes over their lifetime, `span` takes a second argument for specifying end-of-span attributes. Line 6 requires that the `success` attribute is set to `true` when the span ends.

`event` similarly describes *span events*, which are point-in-time occurrences within a span. Events carry key-value attributes like spans, but because they are recorded at a single instant, `event` uses a single attribute schema.

`seq`, `alt`, and `rep` are used to compose more complex policies from simpler ones. `seq` specifies that a sequence of policies must be matched in order. `alt` allows for alternative policies, where at least one of the specified policies must be matched. `rep` specifies that a policy can be repeated zero or more times. `rep` can be further constrained by specifying minimum and maximum number of repetitions.

Our policy DSL also supports writing policies that depend on earlier spans or events. For example, in the reset password endpoint discussed in Section II, we required that the `email` attribute in the `update-user` and `send-password-changed` spans match the same attribute in the `verify-user-password` span. To achieve this, we allow policies to *bind* attribute values from spans or events to variables, which can then be referenced in later policies. In Figure 3, line 4, we bind the `email` attribute from the `verify-user-password` span to the variable `verifiedEmail`, which is stored in the policy evaluation context. Later, when specifying the `update-user` and `send-password-changed` spans, we reference the `verifiedEmail` variable using the `get` function to ensure that the `email` attributes match. [3]

B. Runtime Checker

Our runtime checker is designed to monitor trace events during program execution and verify their compliance with the given policies.

The checker operates over a unified abstraction of trace events, which we call *marks*. Each mark represents an observable event in the trace, such as the start or end of a span, or an event within a span. Marks also carry the associated attributes as key-value pairs. When the checker is initialized with a policy, it converts any `spans` to their corresponding

[1] Adapting Assertive Trace to other languages is straightforward; we leave this to future work.

[2] We use Zod for our prototype due to its popularity, though any schema validation library in any language could serve the same purpose.

[3] In addition to equality checks, we also support more complex constraints, such as substring and prefix checks for string attributes and comparisons for numeric attributes, by providing constructs like `contains`, `startsWith` and `greaterThan`.

sequences of `span start` and `span end` marks, allowing the checker to reason about spans and span events uniformly.

Our prototype checker uses Brzozowski derivatives [10] to evaluate policies against the stream of marks, as they provide a clean, compositional semantics that matches our policy structure. Given a policy P and a mark m, the derivative of P with respect to m, denoted as $\delta(P, m)$, represents the remaining policy that must be satisfied after consuming the mark m. A policy is *nullable* if it can accept the empty sequence (i.e., has been fully satisfied). The definitions of derivatives and nullability for our policy constructs are shown in Table I. The checker processes each incoming mark by computing the derivative of the current policy with respect to the mark. A stream of marks is accepted by the policy if all marks can be consumed without reaching a non-derivable state and the resulting policy is nullable.

The checker is also responsible for maintaining the said policy evaluation context—a mapping from variable names to their bound values.

C. OpenTelemetry SDK Integration

We implemented our runtime checker as a custom OpenTelemetry SDK. This allows developers to integrate Assertive Trace into their existing applications with minimal changes. Specifically, developers only need to replace the standard OpenTelemetry SDK with our drop-in compatible SDK—and attach policies to relevant spans.

Our custom SDK extends the standard OpenTelemetry tracing APIs to support policy attachment while preserving compatibility with existing instrumentation. For example, the `startActiveSpan` method, which is commonly used to create and start a new span, normally takes the span name and a callback function that contains the program logic to be executed within the span. Our SDK simply extends this method to accept an optional policy parameter.

```
tracer.startActiveSpan('span-name',
    { policy: spanPolicy }, // <- new policy parameter
    (span) => {
        // --- Execute program here ---
        span.end(); // throws if policy is not satisfied
    }
);
```

When a policy is provided, the tracer creates a monitor that tracks execution state and attaches it to the span. If any policy violations are detected during execution or when the span ends, a `PolicyViolationError` is thrown.

IV. Case Studies

To understand if Assertive Trace is expressive enough to capture real-world policies, and if our approach can be used to retrofit policy enforcement to existing code, we evaluate our framework on several case studies.

A. Role-based Access Control: Group Wiki

We implemented a multi-tenant group wiki application with role-based access control (RBAC) using Assertive Trace. In this application, users can create groups and add members with one of three roles: `admin`, `editor`, or `viewer`. Each role has different permissions regarding wiki page and group management actions.

```
export const addMemberPolicy = seq([
  // Must be logged in
  event({
    name: "auth",
    "user.id": str().bind("requesterId"),
  }),
  // Check membership first
  event({
    name: "check-membership",
    "membership.id": str(),
    "user.id": get("requesterId"),
    "group.id": str(),
    "user.role": "admin",
  }),
  // Then add member
  span({
    name: "add-member",
    "request.user.id": str(),
    "request.group.id": str(),
  }),
]);
```

Fig. 4. Policy for adding a member to a group wiki, enforcing that only admins can add members to their own group.

The policies we wrote are representative of real-world policies. Consider, for example, the policy for adding members to a group: We allow adding members to a group only if the requesting user belongs to the same group as an `admin`. We express this policy in Assertive Trace by specifying the required sequence of spans and attribute constraints. When a user attempts to add a member, the app first checks the requester's group membership. The policy requires that the app emits a `check-membership` event with the IDs of the membership entry, group, user, and the user's role (which must be `admin`) before proceeding to the `add-member` span. Figure 4 shows the policy implementation.

The important aspect of this policy is that it uses the event as evidence that the app has performed the necessary authorization checks. In this case, the membership ID can only be obtained by performing the check. Having this evidence in the trace ensures that the app cannot bypass the authorization step without violating the policy. This pattern of including evidence as events is applicable to many access control scenarios.

B. Break-the-Glass Access: Medical Records

To ensure that Assertive Trace can express access control patterns beyond simple RBAC, we implemented a "break-the-glass" policy for medical records. In this pattern, patients can normally view only their own records, but doctors can gain temporary (e.g., 24-hour) emergency access by first recording an audit entry with a justification.

Despite the added complexity—disjunctive access paths and temporal constraints—the policy still follows the same pattern: access is granted only when the trace contains the required evidence.

Figure 5 shows the policy, which allows two access paths. The first path (self-access) permits patients to access their own records when the requester ID matches the patient ID. The second path (emergency access) requires that the system

TABLE I

DERIVATIVE AND NULLABLE DEFINITIONS. VARS(M) EXTRACTS VARIABLE BINDINGS FROM MARK m AND $\oplus$ DENOTES UPDATING THE ENVIRONMENT. THE RELATION $m \models_E s$ HOLDS WHEN MARK m MATCHES SCHEMA s UNDER E.

Policy P	**Derivative** $\partial(P, m)$	**Nullable** $\nu(P)$
$\varepsilon[E]$	$\emptyset$	`true`
$\emptyset$	$\emptyset$	`false`
$\mathtt{mark}(s)[E]$	$\varepsilon[E \oplus vars(m)]$ if $m \models_E s$; else $\emptyset$	`false`
$\mathtt{seq}(P_1[E_1], P_2)$ if $\nu(P_1)$	$\mathtt{alt}(\mathtt{seq}(\partial(P_1, m), P_2),\ \partial(P_2[E_1], m))$	$\nu(P_1) \wedge \nu(P_2)$
$\mathtt{seq}(P_1, P_2)$ if $\neg\nu(P_1)$	$\mathtt{seq}(\partial(P_1, m), P_2)$	
$\mathtt{alt}(P_1, P_2)$	$\mathtt{alt}(\partial(P_1, m), \partial(P_2, m))$	$\nu(P_1) \vee \nu(P_2)$
$\mathtt{rep}(P, n, x)$	$\mathtt{seq}(\partial(P, m), \mathtt{rep}(P, n{-}1, x{-}1))$	$n \leq 0$

```
export const dataAccessPolicy = seq([
  event({
    name: "auth",
    "user.id": str().bind("requesterId"),
    "user.role": str().bind("role"),
  }),
  alt([
    // Path A: Patient self-access
    event({
      name: "user-is-patient",
      "requester.id": get("requesterId"),
      "patient.id": str().bind("patientId"),
    }),
    // Path B: Doctor emergency access
    event({
      name: "exists-break-glass-entry",
      "break-glass.id": str(),
      "doctor.id": get("requesterId"),
      "patient.id": str().bind("patientId"),
      "expiration": num()
        .refine(e => e > Date.now()),
    }),
  ]),
  // access medical record via http
  span({
    method: "GET",
    "server.address": "database.internal",
    "url.path": "/medical-records/{patientId}",
  }),
]);
```

Fig. 5. Policy for medical record access with break-the-glass emergency override, enforcing evidence-based access control with temporal constraints.

has a break-glass entry in the database indicating that the doctor has emergency access to the patient's records, and that the entry has not expired. If either path is satisfied, the policy allows making an HTTP GET request to the internal medical records server. We use the standard attributes defined in OpenTelemetry Semantic Conventions [11].

The policy uses attribute bindings to ensure that the requester ID and patient ID are consistent across spans and events. In this scenario, `break-glass.id` serves as evidence that the application has performed the database lookup and found a valid emergency access entry.

C. Auto-instrumentation: DynamoDB Example

The previous case studies show how user-defined events can be used to enforce security policies. One of the main reasons to build on OpenTelemetry, however, is to minimize the amount of code developers need to modify: OpenTelemetry provides auto-instrumentation libraries for many frameworks and services, generating spans without requiring code changes. To demonstrate that Assertive Trace integrates seamlessly with these existing instrumentations, we applied it to a DynamoDB example from the AWS SDK documentation [12].

The example runs a "movies scenario" that performs a sequence of DynamoDB operations: creating a table, waiting for it to become available, inserting and querying movie records, and finally deleting the table. Using OpenTelemetry's AWS SDK instrumentation, each DynamoDB API call automatically generates a span with attributes such as `aws.dynamodb.table_names`.

Figure 6 shows an excerpt of the policy, which validates both the operation sequence and a cross-operation invariant: all operations must target the same table that was initially created. The policy captures the table name from the `CreateTable` span and verifies that subsequent operations reference the same table. Operations that may occur multiple times (e.g., `DescribeTable` during polling, paginated `Query` and `Scan`) use the `repeat` combinator.

This example highlights a key benefit of building on OpenTelemetry: developers can write policies over auto-generated spans without modifying application code, enabling policy enforcement for existing applications with minimal integration effort.

V. RELATED WORK

A. Observability and OpenTelemetry

Structured logging and distributed tracing are established techniques for observing production systems. While early logging used unstructured text with basic metadata [13], modern *structured logging* emits machine-readable event records with named fields for reliable indexing and correlation. Distributed tracing systems—from early work like Magpie [14] and X-Trace [15] to Dapper's trace-and-span model [16]—introduced end-to-end request reconstruction via context propagation.

OpenTelemetry is a community-driven standard for generating, exporting, and collecting telemetry signals (traces, metrics, and logs). As a Cloud Native Computing Foundation project launched in 2019, it has achieved broad industry adoption, with activity levels surpassing Kubernetes [17]. Major cloud providers—AWS [18], Google Cloud [19], and Azure [20]—offer OpenTelemetry-compatible observability services.

```
export const dynamoDBPolicy = seq([
  // CreateTable - capture table name
  span({
    name: "DynamoDB.CreateTable",
    "aws.dynamodb.table_names":
      tuple([str().bind("tableName")]),
  }),
  // DescribeTable (polling, 1+ times)
  rep(span({
    name: "DynamoDB.DescribeTable",
    "aws.dynamodb.table_names":
      tuple([get("tableName")]),
  }), 1, Infinity),
  // PutItem, GetItem, UpdateItem, DeleteItem
  span({
    name: "DynamoDB.PutItem",
    "aws.dynamodb.table_names":
      tuple([get("tableName")]),
  }),
  // ... (GetItem, UpdateItem, DeleteItem)
  // DeleteTable - verify same table
  span({
    name: "DynamoDB.DeleteTable",
    "aws.dynamodb.table_names":
      tuple([get("tableName")]),
  }),
]);
```

Fig. 6. Policy for DynamoDB movies scenario, validating operation sequence and table name consistency using auto-instrumented spans.

B. Security Policy for Web Applications

A long line of work advocates separating security policy from application code so policies can be reviewed independently and enforced consistently.

Such enforcement is often provided by platform- or datastore-level policy languages. PostgreSQL row-level security (RLS) [21] enforces per-row visibility and update predicates inside the query engine. Backend-as-a-service platforms similarly offer declarative policies such as Firebase Security Rules, which restrict reads and writes based on request context and resource attributes [22].

The academic community explored information flow control to enforce policies more systematically. Frameworks like Hails [4], Jacqueline [23], LWeb [24] and STORM [5] integrate information flow tracking into web applications, allowing developers to specify and enforce security policies and guarantee noninterference properties. However, these systems often require specific programming languages or frameworks, limiting their applicability in diverse development environments.

Assertive Trace differs from these approaches by writing policies over server-side traces, which allows it to operate independently of the choice of application framework or datastore. It can enforce a wide range of policies, including cross-component and temporal properties that are difficult to express in traditional policy languages whose substrate is bound to a single component or operation.

VI. Discussion and Future Work

In this section, we discuss several directions for future work and extensions to our current design of Assertive Trace.

a) Language Support: While we implemented our prototype in TypeScript, the concepts and techniques we present are language-agnostic and can be applied to other programming languages. OpenTelemetry has SDKs for many popular languages, which can serve as the basis for implementing Assertive Trace in those languages. Compiled languages like Go and Rust may have different trade-offs in terms of performance and integration complexity compared to interpreted languages.

b) Performance: We have not yet evaluated the performance overhead introduced by our runtime checker. While we designed the checker to be efficient by using Brzozowski derivatives with some simplifications already in place, a thorough performance evaluation is necessary to understand its impact on application latency and throughput. Future work should include benchmarking the checker under different workloads and policies.

c) Static Policy Enforcement: Our current prototype enforces policies at runtime. In some settings, however, static enforcement may be preferable—for example, safety-critical systems where runtime violations are unacceptable, or latency-sensitive services where any runtime overhead is prohibitive.

We plan to explore static verification of policy adherence. One approach is to compile policies into typestates [25] to encode control flow in types; to capture constraints over event attributes, this can be combined with refinement types [26]. Existing systems such as Flux [27] and Liquid Haskell [28] provide practical implementations, and verification tools such as Dafny [29] and Verus [30] may also apply.

d) Meta policies and hyperproperties: Our current design attaches policies to individual spans and enforces them within each span's lifetime, which keeps integration with OpenTelemetry simple and policies modular. Some requirements, however, are inherently global or relational: a *meta policy* may constrain all spans in an application (e.g., any database write must be preceded by authentication, independent of endpoint), and *hyperproperties* relate multiple spans (e.g., rate limiting requires bounding how often a user performs an action within a time window). Enforcing such properties requires trace-level specification and monitoring over histories of spans and their attributes, beyond our current language and enforcement mechanism.

VII. Conclusion

We presented Assertive Trace, a framework for specifying and enforcing security and correctness policies over server-side application traces. By treating traces as the substrate for policy specification, Assertive Trace enables developers to express expected execution behavior as executable policies and enforce them at runtime.

Our prototype integrates as a drop-in replacement for the OpenTelemetry SDK and reuses existing instrumentation, requiring minimal changes to application code. Through case studies, we showed that this approach can express and enforce non-trivial access control and correctness properties in realistic web applications.

Assertive Trace demonstrates that observability infrastructure can serve not only as a diagnostic tool, but also as a practical enforcement layer for application-level guarantees.

References

[1] "Owasp top 10:2021," Online, 2021, accessed: 2025-12-10. [Online]. Available: https://owasp.org/Top10/2021/

[2] "Owasp top 10:2025," Online, 2025, accessed: 2025-12-10. [Online]. Available: https://owasp.org/Top10/2025/

[3] MITRE Corporation. (2025) CWE Top 25 Most Dangerous Software Weaknesses. MITRE. [Online]. Available: https://cwe.mitre.org/top25/index.html

[4] D. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," *Journal of Computer Security*, vol. 25, no. 4-5, pp. 427–461, Jul. 2017. [Online]. Available: https://journals.sagepub.com/doi/full/10.3233/JCS-15801

[5] N. Lehmann, R. Kunkel, J. Yang, A. Software, N. Vazou, N. Polikarpova, D. Stefan, and R. Jhala, "Storm: Refinement Types for Secure Web Applications."

[6] J. Renner, A. Sanchez-Stern, F. Brown, S. Lerner, and D. Stefan, "Scooter & Sidecar: a domain-specific approach to writing secure database migrations," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Virtual Canada: ACM, Jun. 2021, pp. 710–724. [Online]. Available: https://dl.acm.org/doi/10.1145/3453483.3454072

[7] J. Parker, N. Vazou, and M. Hicks, "LWeb: information flow security for multi-tier web applications," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3290388

[8] W3C Distributed Tracing Working Group, "Trace context," W3C Recommendation, Nov. 23 2021, uRL: https://www.w3.org/TR/trace-context/ (accessed 2025-12-10).

[9] C. Williams, "Zod," https://zod.dev, 2020, accessed: 2025-12-11.

[10] J. A. Brzozowski, "Derivatives of Regular Expressions," *Journal of the ACM*, vol. 11, no. 4, pp. 481–494, Oct. 1964. [Online]. Available: https://dl.acm.org/doi/10.1145/321239.321249

[11] OpenTelemetry, "Semantic conventions," https://opentelemetry.io/docs/concepts/semantic-conventions/, accessed: 2025-12-16.

[12] Amazon Web Services, "Learn the basics of dynamodb with an aws sdk," https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/example_dynamodb_Scenario_GettingStartedMovies_section.html, Amazon Web Services, accessed: 2025-12-17. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/example_dynamodb_Scenario_GettingStartedMovies_section.html

[13] R. Gerhards, "The syslog protocol," RFC 5424, Mar. 2009. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5424.html

[14] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: online modelling and performance-aware systems."

[15] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A Pervasive Network Tracing Framework."

[16] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure."

[17] C. Aniszczyk, "A mid-year 2025 look at cncf, linux foundation, and the top 30 open source projects," https://www.cncf.io/blog/2025/07/18/a-mid-year-2025-look-at-cncf-linux-foundation-and-the-top-30-open-source-projects/, Jul. 2025, blog post.

[18] "Otlp endpoints – amazon cloudwatch," https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-OTLPEndpoint.html, 2025, accessed: 2025-12-10.

[19] "Use the ops agent and opentelemetry protocol (otlp) – google cloud trace," https://docs.cloud.google.com/trace/docs/otlp, 2025, accessed: 2025-12-10.

[20] "Opentelemetry — azure monitor application telemetry," https://learn.microsoft.com/en-us/azure/azure-monitor/app/opentelemetry, 2025, accessed: 2025-12-10.

[21] PostgreSQL Global Development Group, "Postgresql: Documentation: 18: 5.9. row security policies," 2025. [Online]. Available: https://www.postgresql.org/docs/current/ddl-rowsecurity.html

[22] Google, "Basic security rules," Firebase Documentation. [Online]. Available: https://firebase.google.com/docs/rules/basics

[23] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Santa Barbara CA USA: ACM, Jun. 2016, pp. 631–647. [Online]. Available: https://dl.acm.org/doi/10.1145/2908080.2908098

[24] J. Parker, N. Vazou, and M. Hicks, "Lweb: information flow security for multi-tier web applications," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290388

[25] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, Jan. 1986. [Online]. Available: http://ieeexplore.ieee.org/document/6312929/

[26] T. Freeman and F. Pfenning, "Refinement types for ml," *SIGPLAN Not.*, vol. 26, no. 6, p. 268–277, May 1991. [Online]. Available: https://doi.org/10.1145/113446.113468

[27] N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala, "Flux: Liquid Types for Rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1533–1557, Jun. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3591283

[28] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. Gothenburg Sweden: ACM, Aug. 2014, pp. 269–282. [Online]. Available: https://dl.acm.org/doi/10.1145/2628136.2628161

[29] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6355, pp. 348–370, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-17511-4_20

[30] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586037