Testing Certificate Authorities: An Empirical Study and Stateful Fuzzing Approach

Akshith Gunasekaran, Max Chase, Zane Ma, Rakesh B. Bobba, Manish Motwani

Abstract*—Certificate Authorities (CAs) form a critical component of the web PKI, implementing standardized, stateful workflows for account management, domain validation, certificate issuance, and revocation. Many modern CAs rely on the ACME (Automatic Certificate Management Environment) protocol to automate domain validation and certificate management at scale, supported by related standards such as OCSP (Online Certificate Status Protocol) and CRLs (Certificate Revocation Lists). Although ACME is formally verified for protocol-level correctness [1], real-world CA implementations continue to exhibit behavioral inconsistencies, and correctness issues in practice [2], [3]. These observations raise a fundamental question: *how well are CA implementations tested, and do developer-written tests adequately exercise protocol-defined behavior?

This paper presents the first empirical study of testing quality of ACME-compliant CAs. We analyze four widely used open-source CA implementations (Boulder, Vault, Smallstep, and Xipki), examining developer-written tests across unit, integration, and end-to-end levels, and measure how effectively they exercise the externally observable certificate management operations defined in the ACME, OCSP, and CRL specifications. Our study shows that developer-written test suites provide limited protocol coverage, with substantial gaps in stateful workflows and specification-defined error-handling logic, and that regression tests are rarely added when protocol bugs are fixed.

To address these gaps, we introduce CAFuzz, a protocol-aware *stateful* black-box test generator that derives behavioral models from RFC specifications and systematically tests CA behavior. CAFuzz achieves 100% protocol operation coverage and 100% protocol error coverage across four CA implementations, compared to 39.5% and 25.6% for developer-written tests, while also improving code coverage from 15.5% to 37.3%. Beyond coverage improvements, CAFuzz detected 3 previously unknown bugs and 43 known protocol-related bugs when replaying historical buggy versions. These results demonstrate that specification-guided, state-aware test generation can effectively improve protocol coverage and expose real-world compliance issues that developer-written tests fail to detect.

I. Introduction

The web Public Key Infrastructure (PKI) enables secure communication over the internet by validating the authenticity of server public keys through digital certificates. Transport Layer Security (TLS) clients (*e.g.*, web browsers) rely on these certificates to authenticate servers they interact with. Certificate Authorities (CAs) are a central component of the web PKI, acting as trusted entities that validate the identities (typically DNS names) of certificate applicants and issue certificates only to verified entities.

Akshith Gunasekaran, Max Chase, Zane Ma, Rakesh B. Bobba, and Manish Motwani are with the School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA.

Manuscript received XXX; revised YYY.

Historically, this ecosystem was difficult to navigate—certificate issuance was complex, costly, manual, and error-prone. For website operators, certificate issuance was confusing and HTTPS certificates were tedious to deploy [4]. Server operators generated keys manually, configured server settings, completed domain control challenges, and paid certificate fees, making the process error-prone and inaccessible to smaller sites and hosting platforms. Historic certificate issuance was problematic for CAs as well. In 2017, Symantec was found to have repeatedly misissued certificates for unauthorized domains due to inadequate validation practices and weak internal controls [5]. This led to widespread certificate distrust, browser-level blacklisting, and an erosion of trust in web PKI.

To reduce complexity and promote HTTPS adoption, the Automated Certificate Management Environment (ACME) protocol [6] was introduced in 2019 to automate domain validation and certificate issuance. ACME has become the de facto standard for automated certificate issuance in modern CA implementations [7].

While the ACME protocol itself has been formally verified for security and correctness [8], real-world CA software remains prone to subtle implementation bugs whose consequences extend far beyond a single service. In 2020, Let's Encrypt's Boulder CA mis-issued roughly 1.7 million certificates after a validation-caching defect caused stale domain-control checks to be reused across orders [3], temporarily undermining the trustworthiness of a major portion of the web PKI. A later security audit of Smallstep CA revealed JSON-injection vulnerabilities, misuse of JSON Web Tokens, and acceptance of untrusted client inputs [2], demonstrating that even memory-safe, modern implementations can mishandle protocol data and authorization logic. These flaws arose not from ACME's design but from errors in the stateful validation and request-processing logic surrounding it. Supporting this broader view, Kumar et al. [9] found more than 1,600 mis-issued certificates across public CAs over two years, confirming that misissuance and non-compliance persist despite protocol standardization. Together, these incidents illustrate how small inconsistencies in CA implementations can cascade into large-scale trust failures—underscoring the need for systematic, implementation-level testing of CA software.

While substantial research has focused on SSL/TLS protocol correctness [10], [11], [12] and client-side certificate validation [13], [14], [15], [16], **little is known about how CA software are tested internally, and it is unclear whether developer-written tests sufficiently cover all operations of a CA.** Coverage-guided fuzzers such as AFL [17] and AFLNet [18], as well as grammar-based network fuzzers such as*Boofuzz* [19], operate at the message or byte level and

rely on low-level instrumentation, mutation-based exploration, and crash-oriented oracles. In contrast, modern CAs are implemented in memory-safe languages (*e.g.*, Go, Java) and communicate over HTTPS using structured, cryptographically authenticated JSON payloads with strict semantic constraints (e.g., valid JWS signatures, fresh nonces, and correct account bindings). As a result, most malformed inputs are rejected early, and implementation bugs manifest as semantic or specification-level violations rather than crashes—rendering traditional fuzzing oracles ineffective even when code is exercised. Moreover, CA protocols such as ACME are *stateful and bidirectional*, rendering request-response-based fuzzers unable to maintain protocol consistency or explore the complex, multi-step workflows that characterize CA behavior.

To address these gaps, we conduct the first empirical study of how well-tested CA systems are. Specifically, we analyze the developer-written test suites across four widely used, ACME-compliant CA systems. We begin by identifying critical CA operations based on the ACME protocol (RFC 8555 [6]), and supplement it with externally testable operations related to certificate revocation as defined in the CRL protocol (RFC 5280 [20]) and OCSP (RFC 6960 [21]). We then develop CAFuzz, a protocol-aware, *stateful* black box testing technique that produces test cases guided by the protocol's state model. Our analysis revealed that developer-written tests often prioritize happy paths and common operations, leaving portions of the protocol state space—especially error conditions—untested. CAFuzz complements these tests by automatically exercising neglected paths, improving coverage, and surfacing specification deviations.

By systematically traversing this specification-driven models, CAFuzz generates semantically valid request sequences that explore protocol state space — including transitions and error cases that developer tests often overlook.

We address the following research questions:

RQ1: What types of tests do CA developers write, and do they adequately test stateful behaviors and error handling in CA protocols?
Our analysis of four widely used CA systems shows that developers predominantly rely on unit and integration tests, with only two systems including tests for multi-step, stateful workflows and protocol-defined error conditions. Most tests focus on isolated components and success cases, providing limited assurance about cross-component interactions, state transitions, and error-handling behaviors.

RQ2: To what extent do CA systems comply with protocol specifications, and how well do developer-written tests cover these behaviors?
Analyzing protocol compliance in four CA systems revealed that CAs vary in their adherence to protocol-defined operations. While all systems implement core ACME flows, many ommit or incompletely test less common operations (e.g., account deactivation, OSCP/CRL). Developer-written tests cover at most 14 of the 17 expected operations, 0–7 of the 16 protocol defined errors and achieve modest code coverage (0–41.6%), indicating incomplete specification compliance and functional validation.

RQ3: How effective is CAFuzz in enhancing the developer-written tests?
CAFuzz substantially outperforms developer-written tests across all four CA implementations. It achieves 100% protocol operation coverage and 100% error coverage, compared to 39.5% and 25.6% for developer tests, while improving code coverage from 15.5% to 37.3%. Beyond coverage gains, CAFuzz discovered 3 previously unknown bugs and successfully rediscovered 43 known protocol-related bugs when testing historical versions. These results demonstrate that specification-guided, stateful test generation effectively addresses the testing gaps in developer-written suites by systematically exercising protocol-defined behaviors and error conditions that manual testing overlooks.

RQ4: How does CAFuzz compare with state-of-the-art protocol testing techniques?
CAFuzz substantially outperforms AFLNet [18], a state-of-the-art stateful protocol fuzzer, across all evaluation metrics. While AFLNet achieves only 11.8–12.5% operation coverage and 25.0% error coverage with minimal code coverage (1.6–3.8%), CAFuzz achieves 100% operation and error coverage with significantly higher code coverage (24.9–47.3%). This significant improvement stems from CAFuzz's specification-guided stateful approach, which systematically explores protocol-defined state transitions and error conditions rather than relying on blind mutation-based exploration.

Overall, this paper makes the following contributions:

- An empirical analysis of developer-written tests across four open-source, ACME-compliant CA implementations.
- CAFuzz, a state-machine-guided, protocol-aware test generation approach for certificate authorities.
- A prototype of CAFuzz, supporting the testing of four widely used CAs using ACME, OSCP, and CRL protocols for certificate management.
- A qualitative evaluation of the quality of developer-written tests in the four real-world CA systems.
- A quantitative evaluation of CAFuzz, demonstrating its effectiveness to: (a) cover all protocol operations including those missed by developers, (b) exercise error-handling transitions, and (c) detect RFC compliance violations in real-world CA implementations.
- A replication package with code and data to support reproducibility published at: https://github.com/ANSWER-OSU/cafuzz

The rest of the paper is organized as follows. Section II describes background on CAs. Section III motivates this study by illustrating examples of under-tested and non-compliant CA operations automatically detected by CAFuzz. Section IV outlines our methodology for assessing developer-written tests in CAs and developing CAFuzz. Section V empirically evaluates developer-written tests, CAFuzz, and existing state-of-the-art protocol testing technique on four CAs. Section VI discusses the implications of our results, suggests future directions for research, and describes the limitations and threats to validity.

Section VII places our work in the context of existing research and, Section VIII summarizes our contributions.

II. Background

The web PKI provides the foundation for secure communication over the internet. At the heart of this ecosystem are CAs, which issue digital public-key certificates that enable server authentication and encrypted connections. Public-key certificates bind a domain name to a public key, allowing clients (e.g., browsers) to verify that they are communicating with a legitimate server. Any implementation flaws in CA software can lead to certificate misissuance, unauthorized traffic interception, and broader erosion of trust in the PKI.

A. The Role of Certificate Authorities in the Web PKI

Certificate Authorities (CAs) are large software systems that implement standardized workflows for issuing, validating, and revoking X.509 digital certificates. Many modern CAs automate this process using formally verified protocols such as the Automated Certificate Management Environment (ACME)[6] protocol, the Online Certificate Status Protocol (OCSP)[21], and Certificate Revocation Lists (CRLs) [20] and rely on supporting components such as certificate databases, revocation responders, and auditing infrastructure. While automation increases scalability and reduces operational complexity, it also introduces new attack surfaces — particularly in stateful logic, challenge-response validation, and revocation pathways. As CAs become automated and complex, the correctness of their implementations becomes increasingly critical.

B. Protocol-Driven Behavior in Modern CAs.

Figure 1 provides an overview of the certificate lifecycle, illustrating how ACME-driven issuance, revocation mechanisms (OCSP/CRL), and certificate transparency logs interact to maintain trust in PKI. The left side shows the ACME client initiating requests, the center depicts the Certificate Authority processing these requests and managing state transitions, and the right side shows relying parties (browsers/servers) validating certificates through revocation checks.

ACME defines a multi-step, stateful workflow for certificate issuance. The process begins with *account creation* (step 1), where the client registers with a CA using a public key. Next, the client submits an *order* (step 2) specifying the domain names for which it requests a certificate. The CA responds with *authorization challenges* (step 3) that the client completes to prove control over the requested domains. After the client fulfills these challenges (step 4)—typically by provisioning specific DNS records or HTTP resources—it submits a Certificate Signing Request during *finalization* (step 5). Finally, the client *retrieves* the issued certificate (step 6), which the CA submits to Certificate Transparency logs for public audit.

Each operation corresponds to a well-defined HTTP endpoint and triggers state transitions in one of five resource types (`Account`, `Order`, `Authorization`, `Challenge`, and `Certificate`), shown as red boxes in the figure. This explicit state machine structure enables fully automated issuance at

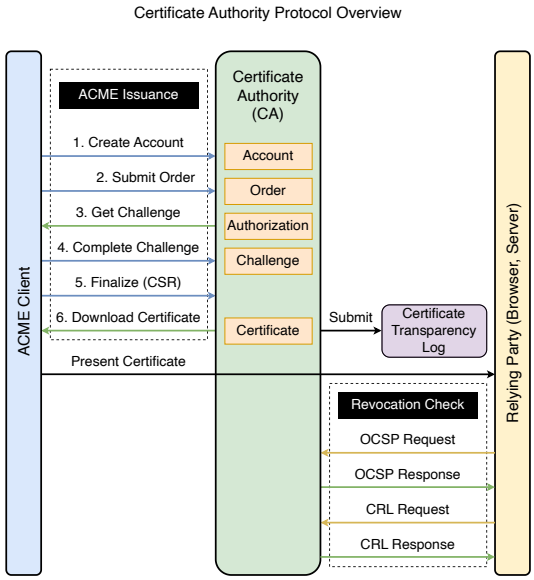

Fig. 1: *Protocol interactions exercised in this study. ACME governs certificate issuance (account creation, order submission, authorization and challenge verification, certificate finalization, and certificate retrieval) between the ACME client and the CA under test. During challenge verification, the CA temporarily acts as a client to a challenge server (e.g., HTTP, DNS, or TLS-ALPN). After issuance, the CA exposes revocation status through OCSP responders and CRL publishers, which are queried by relying parties. Together, these interactions define the externally observable state space that our test generation approach systematically explores.*

Internet scale, but it also creates a large behavioral surface that must be correctly implemented and thoroughly tested. Unlike typical REST APIs, ACME requires cryptographically authenticated requests, strict nonce handling, and multi-step interactions that must follow the precise sequence shown.

In parallel, OCSP and CRL provide post-issuance revocation checks (bottom of figure). These are stateless, query-response protocols that expose additional observable behaviors distinct from ACME's stateful flows. Together, ACME, OCSP, and CRL define the full set of externally visible operations that a CA implementation must support.

C. Testing Challenges in CA Implementations.

From a software testing perspective, CA systems exhibit the following properties that make them difficult to test using conventional unit testing or random-mutation-based fuzzing:

1) *Statefulness:* Many behaviors occur only after specific sequences of prior operations (e.g., an authorization must transition to *valid* before an order is finalized).
2) *Bidirectional interactions:* During challenge verification, the CA temporarily acts as client, flipping the direction of interaction and complicating black-box testing.
3) *Structured and authenticated messages:* Requests are cryptographically protected and highly structured. Random mutations rarely remain syntactically valid, limiting the effectiveness.

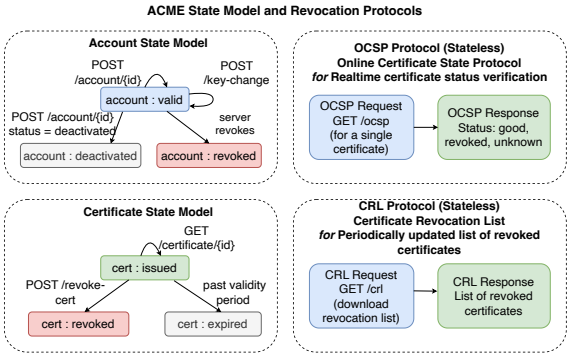

Fig. 2: *ACME state models and revocation protocols exercised in this study. The left panels show simplified state machines for ACME* `account` *and* `certificate` *resources, illustrating valid state transitions triggered by protocol operations. The right panels depict the stateless revocation interfaces exposed via OCSP and CRL, which allow relying parties to query certificate status after issuance. Together, these components define the externally observable protocol behavior that* CAFuzz *models and systematically explores.*

4) *Specification-driven behavior:* Correctness depends on conformance to protocol-defined operations and error semantics.

These factors collectively create a gap between what is testable in principle via the public protocol interface and what is actually exercised by developer-written tests in practice.

D. Protocol State Model.

The ACME, OCSP, and CRL specifications collectively define a finite set of protocol operations and the conditions under which each operation is valid. Each ACME resource type (`Account`, `Order`, `Authorization`, `Challenge`, and `Certificate`) progresses through a well-defined sequence of states. For example, the `Account` resource transitions between `valid`, `deactivated`, and `revoked` states; the `Order` resource progresses through `pending`, `ready`, `processing`, `valid`, and `invalid` states; and the `Certificate` resource moves between `issued`, `revoked`, and `expired` states. Each client request triggers a transition in exactly one of these resources, and the specifications enumerate the expected responses and error conditions for every transition.

Figure 2 illustrates representative ACME resource state models alongside the OCSP and CRL revocation protocols. The left side shows two stateful ACME resources: the `Account` resource (top) demonstrates state transitions including self-loops for account updates (`POST /newAccount`, contact information changes, key changes) while remaining in the `valid` state, with transitions to terminal states `deactivated` (client-initiated) or `revoked` (server-initiated for policy violations). The `Certificate` resource (bottom) shows the lifecycle from `issued` to either `revoked` (via `POST revokeCert`) or `expired` (time-based). Similar state machines exist for `Order`, `Authorization`, and `Challenge` resources, each with their own state transitions and operations.

In contrast, the right side shows OCSP and CRL as stateless request-response protocols: OCSP queries return the status of a specific certificate (good, revoked, or unknown) in real-time, while CRL requests fetch a periodically updated list of all revoked certificates. Unlike ACME's stateful workflows, these protocols maintain no state between requests—each query is independent.

We abstract this protocol behavior as a directed graph where nodes represent states and edges represent operations that cause transitions. This model captures all externally observable behavior prescribed by the RFCs and provides the structure needed to systematically enumerate request sequences, explore protocol paths, and evaluate conformance. By treating the union of ACME, OCSP, and CRL as a structured model of expected behavior, we can systematically generate tests that exercise both common and infrequent code paths, measure operation and error coverage, and detect violations by comparing actual CA behavior against specification-defined transitions.

Concretely, our model comprises a small but expressive state space. Across ACME, we model five stateful resources with a total of 18 distinct states and 19 specification-defined state transitions and 16 testable error transitions. In contrast, OCSP and CRL contribute no persistent protocol states, but expose a finite set of response outcomes (e.g., good, revoked, unknown) governed by certificate metadata and revocation timing. Together, these models form a compact yet complete representation of the externally observable behavior defined by the RFCs.

III. Motivation: Uncovering Untested Operations and Protocol Deviations

Despite formal specification and verification of protocols like ACME, our analysis revealed that real-world CA implementations often exhibit gaps in test coverage and deviations from protocol requirements. These issues undermine the goals ACME set out to achieve: secure, reliable, and fully automated certificate management.

For example, Figure 3 shows two examples of issues detected by CAFuzz in CA implementations:

Untested operations. Figure 3 (top) shows a request-response trace automatically generated by CAFuzz to test the *account deactivation* operation as specified in ACME RFC 8555 (section 7.3.6). While analyzing Let's Encrypt CA implementation - Boulder's developer-written tests, we found that this operation was completely untested. Failure to correctly implement or test this operation may prevent clients from deactivating compromised or obsolete accounts, potentially leaving cryptographic keys and ACME accounts active and exposed. Such gaps weaken a CA's ability to support safe automation and timely incident response. We emphasize that this example is illustrative. Similar gaps in coverage for low-frequency, stateful operations appear across multiple CAs in our study and are quantified in Section V.

Protocol deviations. Figure 3 (bottom) shows a request–response trace automatically generated by CAFuzz that exposes a protocol violation in Boulder [22]. The request attempts to finalize an order containing a syntactically valid CSR for a reserved domain (e.g., `*.example.com`), an input that is explicitly disallowed by ACME policy. Such adversarial-

```
# Trace 1: Untested account deactivation
1: GET /directory > 200 OK
2: HEAD /new-nonce > 200 OK
3: POST /new-acct > 201 Created
4: POST /acct/214023744 {"status":"deactivated"}
                            > 200 OK
```

```
# Trace 2: Incorrect error in order finalization
POST /acme/finalize/24/221 {"csr": "..."}
   -> 500 Internal Server Error
{
 "type":
     ↪ "urn:ietf:params:acme:error:serverInternal",
  "detail": "Error finalizing order",
  "status": 500
}
```

Fig. 3: CAFUZZ*-generated request/response traces highlighting untested and non-compliant behaviors in Boulder. Trace 1 shows a fully valid account-deactivation workflow that is implemented but never exercised by developer-written tests. Trace 2 illustrates a protocol deviation in error handling: when finalizing an order for a reserved domain, the CA returns a transient* `serverInternal` *error instead of a permanent policy error as prescribed by the ACME specification. Together, these examples demonstrate how protocol-aware test generation exposes untested operations and subtle specification violations that are missed by existing test suites.*

but-valid inputs are generated by CAFUZZ using protocol-defined input generation rules designed to exercise specification-mandated error behavior.

When processing this request, Boulder responds with a `serverInternal` error (HTTP 500), indicating a server-side failure. According to RFC 8555 §6.7, the CA should instead (1) gracefully reject the request without crashing and (2) return a permanent policy error such as `rejectedIdentifier` (HTTP 403) to signal that the request is invalid and must not be retried. Returning a generic 500 error conflates a policy violation with an internal failure, causing ACME clients to interpret the rejection as a transient server error and potentially leading to unnecessary retries.

This bug is triggered only under specific policy-violating inputs and is not exercised by developer-written tests, which primarily focus on successful issuance workflows and do not validate robustness or correctness of error handling under specification-relevant adversarial conditions.

These examples illustrate a deeper problem: formal protocol correctness does not guarantee robust CA implementations. Developer-written tests often focus on the *happy* paths and may overlook edge cases or strict compliance requirements. As a result, even widely deployed CAs may silently violate specifications, risking client incompatibility, degraded automation, or security blind spots.

The goal of this study is to systematically uncover such gaps and address them by building CAFUZZ, which models the protocol state machine and validates CA functionality against specifications derived from RFCs. CAFUZZ reveals both untested transitions and deviations from specification—complementing manual tests and strengthening implementation correctness.

IV. METHODOLOGY

Our methodology has four main components: (1) selecting a diverse set of real-world, open-source CAs as study subjects (§IV-A); (2) defining metrics to evaluate the effectiveness of developer-written or automatically-generated tests to verify CA implementations (§IV-B); (3) qualitatively and quantitatively analyzing their developer-written test suites with respect to protocol-defined behaviors (§IV-C); and (4) designing CAFUZZ, a protocol-aware black-box test generator that systematically exercises CA behavior using models derived from RFC specifications (§IV-D).

A. Subject CAs

To evaluate the testing and protocol conformance in realistic settings, we focus on actively maintained, open-source CAs that implement the ACME protocol. Our objective is to cover both Internet-scale public deployments and widely used enterprise PKI systems. While our subject selection requires ACME support for comparability, we analyze OCSP and CRL behavior for all selected systems as part of the CA's externally observable protocol surface.

Public web CAs. Public CAs issue certificates that are trusted by web browsers and other relying parties. We use market share data from W3Techs [23], which builds on the Chrome UX Report [24] to analyze the CAs used by the top 1000 websites. From this dataset, Let's Encrypt emerges as the dominant provider. Its open-source backend, **Boulder**, is therefore included as the representative public CA in our study. Other major public CAs (e.g., DigiCert, GlobalSign, Sectigo) rely on proprietary backends and are not available for source-level analysis.

Enterprise PKI systems. For enterprise and internal PKI deployments, we conduct a systematic search on GitHub and the web using queries such as *"certificate authority"*, *"ACME server"*, and *"PKI CA software"*. From these results, we identify three prominent open-source systems that support ACME and are widely adopted as internal CAs: **HashiCorp Vault**, **Smallstep Certificates**, and **Xipki**.

Inclusion criteria. We include a CA system in our study if it meets all of the following criteria:

- (C1) **ACME protocol support.** The system must implement ACME (RFC 8555), enabling a consistent protocol-level analysis and direct comparison across implementations.
- (C2) **Open-source availability.** The CA implementation and its developer-written tests must be publicly accessible, allowing us to inspect and execute the test suites.
- (C3) **Active maintenance.** The repository must show evidence of ongoing development: at least one commit within the past 90 days, at least one recently updated issue, and a tagged release within the past six months. This filters out stale or abandoned projects.

Applying these criteria yields four ACME-compatible CAs: **Boulder** [22], **Vault** [25], **Smallstep** [26], and **Xipki** [27]. Together, these systems cover both public Internet-scale issuance and configurable enterprise PKI frameworks. Table I summarizes the selected CAs, including implementation language, repository metadata, and typical deployment contexts.

TABLE I: *Metadata of ACME compliant open-source CA systems included in our study.*

CA	Lang	Stars	Commit SHA	Type
`Boulder`	Go	5,000	`b26b11686`	Public
`Vault`	Go	31,000	`322786e236`	Enterprise
`Smallstep`	Go	7,000	`0cf1c568`	Enterprise
`Xipki`	Java	500	`786c50722`	Enterprise

B. Test suite effectiveness metrics

We evaluate the effectiveness of test suites validating the implementations of CAs using the following metrics:

- **Statement Coverage (SC):** Measures the proportion of source code statements executed by tests. A higher statement coverage suggests that large portions of the codebase are exercised, however, it does not imply that all behaviors are validated.
- **Operational Coverage (OC):** Measures the fraction of testable protocol operations (Table III) exercised by the test suite. This captures protocol-awareness and functional completeness in test suite.
- **Error Coverage (EC):** Measures the fraction of protocol-defined error conditions (Table III) triggered by malformed or invalid inputs. This reveals to what extent tests validate robustness against incorrect inputs or protocol misuse.
- **Defect Detection Ratio (DDR):** Measures the fraction of resolved issues whose commits involve adding or modifying tests as part of the fix. A low DDR indicates that developers often do not update tests for fixed issues, creating a potential test gap, while a high fraction indicates good test hygiene and regression protection.
- **Bugs Detected (BD):** Measures the number of unknown (in latest stable version) and known (in historic buggy versions) protocol deviations detected by tests. For a highly tested real-world application, we expect unknown bugs to be zero.

Together, these metrics provide a comprehensive view of how well test suites exercise protocol-level behaviors and identify failures.

C. Analyzing Developer-Written Tests

To understand how CA developers test protocol-related behavior, we analyze the developer-written tests of each subject CA using both qualitative (§IV-C1) and quantitative (§IV-C2) methods.

1) Qualitative Analysis: The qualitative analysis focuses on identifying the structure and intent of developer-written tests. We classify tests into the following three categories:

1) **Unit Tests.** Tests that validate individual functions, handlers, or internal modules in isolation, typically without persistent state or network communication.
2) **Integration Tests.** Tests that exercise interactions between multiple components (e.g., storage layers, signing services, HTTP handlers) using embedded or simulated infrastructure.
3) **End-to-End Protocol Tests.** Tests that drive the system through ACME/OCSP/CRL operations via the public API. We evaluate these tests along two dimensions:
   - *Protocol-defined operations*: Whether the test covers a valid operation defined by the RFCs (e.g., account creation, challenge validation, certificate issuance, OCSP/CRL retrieval).
   - *Protocol-defined errors*: Whether the test checks for expected error responses (e.g., malformed requests, replayed nonces, unauthorized revocation).

Analysis procedure: We examine each repository's directory structure, test organization, and build/CI configuration to identify test entry points and execution patterns. To locate error-handling tests, we apply lightweight static analysis over test files, searching for HTTP error assertions (4XX codes), exception patterns, and ACME-specific error identifiers (`urn:ietf:params:acme:error:*`). Results are manually validated.

To detect stateful tests, we extract sequences of API interactions within test cases and identify whether they reuse resource identifiers (orders, authorizations, challenges) across multiple requests. Tests that perform multi-step workflows are labeled as end-to-end protocol tests.

This classification provides a structural overview of testing practices prior to any coverage measurement.

2) Quantitative Analysis: We quantitatively assess how effectively developer-written tests exercise the externally observable behavior defined in ACME (RFC 8555), OCSP (RFC 6960), and CRL (RFC 5280).

Externally observable protocol operations and errors: We manually extract all protocol-defined operations and structured error conditions from the RFCs, including their preconditions, success responses, and error semantics. An operation is considered *testable* if it can be invoked through the public API by an external client. Internal behaviors (e.g., background CRL generation, challenge expiration timers) are marked untestable (from an external perspective) and excluded.

In total, we identify 17 testable operations and 16 testable error conditions across ACME, OCSP, and CRL. Table III summarizes these behaviors.

D. CAFuzz*: Protocol-Aware Black-Box Test Generation*

To assist CA developers in testing protocol behavior, we design CAFuzz, a protocol-aware, stateful black-box test generation approach that systematically explores CA behavior using protocol models derived from the ACME, OCSP, and CRL specifications. Figure 4 provides an overview.

1) Inputs and Outputs: CAFuzz takes as input (1) the RFC specifications for ACME (RFC 8555), OCSP (RFC 6960), and CRL (RFC 5280), and (2) a running CA deployment that exposes the corresponding protocol endpoints. From the RFCs, CAFuzz derives a machine-readable protocol specification comprising: (i) a Protocol State Model (PSM), (ii) Request Schemas (RS), and (iii) Input Generation Rules (IGR).

For each CA, CAFuzz produces:

- a full request-response trace for every explored path,
- protocol-level coverage (operations and errors exercised),
- and a list of deviations from the RFC-defined model.

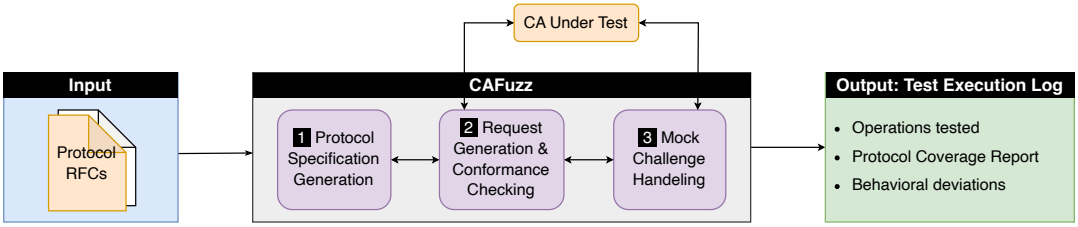

Fig. 4: *Protocol-aware, stateful black-box test generation workflow implemented by* CAFuzz. *Starting from protocol RFCs,* CAFuzz *first performs* ***(1) Protocol Specification Generation****, where an LLM-assisted extraction followed by manual validation produces a machine-readable protocol specification comprising the Protocol State Model (PSM), request schemas (RS), and input generation rules (IGR). Next, in* ***(2) Request Generation and Conformance Checking****,* CAFuzz *systematically enumerates protocol-permitted paths from the PSM and generates state-aware request sequences, validating observed responses against specification-defined success and error semantics. During execution,* ***(3) Mock Challenge Handling*** *coordinates challenge responders to enable deterministic traversal of ACME validation workflows. All request-response interactions are logged in a test execution log, from which protocol coverage metrics and behavioral deviations from the RFC-defined model are reported.*

2) Stage 1: Protocol Specification Generation: CAFuzz begins by constructing an explicit, machine-readable protocol specification from the ACME, OCSP, and CRL RFCs. This specification serves as the semantic foundation for all subsequent test generation and conformance checking.

LLM-assisted extraction and validation. Protocol behavior in CA-related RFCs is distributed across prose, tables, registries, and cross-referenced sections, making manual construction of complete protocol models tedious and error-prone. We therefore employ a hybrid extraction workflow in which a large language model (GPT-4.1) is used strictly as a *structured extraction aid*, followed by comprehensive manual validation and correction. The prompts used are shown in Figure 5.

Extraction proceeds in two passes. In the first pass, the LLM is prompted with the RFC text and a fixed JSON schema to extract candidate protocol elements, including: (i) protocol resources and their lifecycle states, (ii) state machines with events and transitions and (iii) protocol-defined error identifiers. In the second pass, we re-prompt the LLM for request/response schemas for each operation. The full extraction prompt enforcing complete enumeration of RFC-defined resources, state machines, and error registries under a fixed JSON schema is available in our project repository.

Normalization and verification. The raw LLM output is then normalized and validated by multiple authors. Normalization includes: (i) canonicalizing endpoint templates (e.g., mapping concrete URLs to templated paths such as `/acct/{id}`), (ii) standardizing resource, state, and event names across RFC sections, and (iii) resolving aliases and cross-references introduced by inconsistent terminology in the specifications. Verification consists of cross-checking every extracted state, transition, and error condition against the cited RFC section(s), correcting missing preconditions, invalid transitions, or misidentified error semantics. All reported protocol models are manually validated, and no experiments depend on unverified LLM output.

Protocol State Model (PSM). The Protocol State Model is a directed graph that captures the externally observable behavior of the protocol. Nodes correspond to resource states, and edges correspond to protocol operations that either advance the resource to a new state on success or produce a specification-defined error. For ACME, the PSM comprises multiple resource-specific state machines (e.g., `Account`, `Order`, `Authorization`, `Challenge`, `Certificate`), together with cross-resource preconditions that constrain legal transitions (e.g., an order may be finalized only after all referenced authorizations are in the `valid` state).

TABLE II: ***Accuracy of LLM-assisted protocol model extraction.*** *We compute precision (P), recall (R), and F1 separately for extracted protocol states and state transitions by comparison against manually constructed ground-truth models validated independently by two authors. Manual effort reports the time required to construct and validate the ground-truth model for each protocol.*

		States			Transitions				
RFC	**Protocol**	**P**	**R**	**F1**	**P**	**R**	**F1**	**overall F1**	**manual effort**
8555	ACME	0.89	1.00	0.94	0.86	1.00	0.92	0.93	2.5h
6960	OCSP	1.00	0.67	0.80	1.00	0.86	0.92	0.86	0.8h
5280	X.509	1.00	0.60	0.75	1.00	0.40	0.57	0.66	0.7h
Average		**0.96**	**0.76**	**0.83**	**0.95**	**0.75**	**0.80**	**0.82**	**1.3h**

Request Schemas (RS). Request Schemas define the syntactic structure required to construct a valid request and interpret its response for each protocol operation. For example, the ACME `newOrder` schema specifies the HTTP method and endpoint template, the required JWS envelope fields (`protected`, `payload`, `signature`), and the operation-specific payload fields such as `identifiers` and optional validity bounds.

Input Generation Rules (IGR). Input Generation Rules define how individual fields in a request schema are instantiated with concrete values. IGR is partitioned into two domains: (i) *valid domains*, which satisfy all protocol constraints and enable state progression (e.g., a fresh nonce, a correctly bound account key, or a Certificate Signing Request (CSR) matching validated identifiers), and (ii) *error-triggering domains*, which remain syntactically valid but violate a specific protocol constraint to elicit a specification-defined error (e.g., replaying a nonce to trigger `badNonce`, submitting a malformed CSR to trigger `badCSR`, or using a reserved identifier to trigger `rejectedIdentifier`).

Extraction accuracy. We quantify the quality of the extracted protocol models by comparing the LLM-assisted output against independently constructed ground-truth models. Table II reports

Stage 1: Protocol state model extraction
System prompt. You are an expert in networking protocols and RFC interpretation. Given an RFC, identify and extract all actors, protocol resources, lifecycle states for each resource, transition between those states, and all protocol-defined error identifiers, and output them as a single JSON object conforming to the schema provided. Do not infer behavior beyond normative RFC text. Output JSON only.

Protocol specification schema

```
{
  "protocol_name": string,
  "rfc_reference": string,
  "actors": [{ "id": string, ... }],
  "resources": [{
    "id": string,
    "status_field": string | null,
    "status_values": [string],
    "rfc_references": [string]
  }],
  "state_machines": [{
     "id": string,
     "scope": "global" | "resource:<id>",
     "states": [{ "id": string, ... }],
     "events": [{ "id": string, ... }],
     "transitions": [{
       "from": string,
       "to": string,
       "event": string, ...
     }]
  }],
  "error_conditions": [{
     "id": string, "description": string,
         ↪ ...
  }]
}
```

User prompt. Extract the complete protocol state machine from the RFC at: <RFC URL>. Return a single JSON object conforming to the schema. Output JSON only.

Stage 2: Request/response schema extraction
System prompt. Given an RFC and an extracted protocol state model, extract request and response schemas for each protocol operation as defined in the RFC. Output JSON only.

User prompt. Using RFC <RFC URL> and the protocol state model, generate request and response schemas for all protocol transitions. Output JSON only. Protocol State Model: <PSM from Stage 1>

Fig. 5: *Two-stage prompt and enforced JSON schema used for LLM-assisted protocol specification generation.*

precision, recall, and F1 scores for extracted states and transitions, along with the manual effort required to construct and validate the ground truth across ACME, OCSP, and CRL. The LLM achieved average F1 scores of 0.83 for states and 0.80 for transitions, with consistently high precision but variable recall that correlated with how explicitly the RFC defined its state machines. ACME's explicit status enumerations enabled near-perfect extraction (F1 0.94 states, 0.92 transitions), while X.509's implicit states yielded lower scores (F1 0.75 states, 0.57 transitions).

3) Stage 2: Request Generation and Conformance Checking:
Path enumeration over protocol execution states. ACME, OCSP, and CRL define multiple interacting protocol resources, such as accounts, orders, authorizations, and certificates. These resources are not independent: protocol operations create, update, and link resources over time. For example, creating an order requires an existing valid account and produces a new order resource that is subsequently referenced by authorization and finalization operations.

Algorithm 1 CAFUZZ: Protocol-Aware Test Generation

Input: Protocol State Machine PSM, request schemas RS, input generation rules IGR, CA under test CA
Output: Test execution log
$visited \leftarrow \emptyset$
$paths \leftarrow$ enumerateAllPaths(PSM)
for each protocol path $p \in paths$ **do**
  initialize empty test log
  $state \leftarrow$ initial state of PSM
  for each operation $op \in p$ **do**
    $req \leftarrow$ constructRequest(op, RS, IGR)
    $resp \leftarrow$ send(req, CA)
    log(req, $resp$)
    if $resp$ conforms to expected success semantics **then**
      $state \leftarrow$ nextState($state, op$)
      $visited \leftarrow visited \cup \{state\}$
    else
      record protocol deviation or expected error
      break ▷ terminate current path
  if all protocol states in PSM are visited **then**
    break
return testExecutionLog

To capture this behavior, CAFUZZ models protocol execution as a single evolving *protocol execution state*. This state records (i) the currently instantiated resources and their protocol states, and (ii) the bindings between resources (e.g., which account owns an order, which authorizations belong to an order). The initial execution state corresponds to a fresh protocol session with no resources instantiated and only protocol-wide prerequisites available (e.g.,directory discovery and nonce acquisition).

CAFUZZ enumerates protocol-permitted execution paths by repeatedly applying operations that are valid in the current execution state, as defined by the Protocol State Model (PSM). Each operation either (i) creates a new resource, (ii) advances the state of an existing resource, or (iii) triggers a protocol-defined error. Path enumeration terminates when workflows reach completion (e.g., certificate issuance), enter terminal invalid states (e.g., order invalidation), or when no further protocol-permitted operations are available. Because the number of protocol resources and their lifecycles are bounded by the specification, exhaustive exploration completes within tens of minutes per CA in practice.

Algorithm 1 summarizes this enumeration and execution process.

Request generation. For each operation along an enumerated path, CAFUZZ instantiates a concrete request using the corresponding Request Schema (RS) and Input Generation Rules (IGR). Generated requests are *semantically valid* with respect to protocol invariants: JSON Web Signatures verify under the correct account key, nonces are fresh, and resource identifiers are consistently bound across requests. This enables CAFUZZ to reach deep protocol logic beyond initial input validation. To exercise robustness, CAFUZZ selectively replaces individual

TABLE III: ***Externally observable protocol operations and error conditions.*** *Extracted from ACME (RFC 8555), OCSP (RFC 6960), and CRL (RFC 5280). Testable items can be triggered and observed by a black-box client. Internal behaviors (e.g., expiration timers, background CRL generation) are marked untestable and excluded.*

Protocol-Defined Operations				
Operation	**Protocol**	**Endpoint**	**Description**	**Testable**
Directory Discovery	ACME	`GET /directory`	Retrieve server metadata and endpoints	✓
Get Nonce	ACME	`HEAD /newNonce`	Obtain anti-replay nonce	✓
Account Creation	ACME	`POST /newAccount`	Register new account	✓
Account Update	ACME	`POST /acct/{id}`	Update contact information	✓
Account Deactivation	ACME	`POST /acct/{id}`	Permanently deactivate account	✓
Key Rollover	ACME	`POST /key-change`	Rotate account key	✓
Order Submission	ACME	`POST /newOrder`	Request certificate for identifiers	✓
Order Status	ACME	`POST-as-GET /order/{id}`	Retrieve order state	✓
Order Finalization	ACME	`POST /order/{id}/finalize`	Submit CSR to complete order	✓
Authorization Status	ACME	`POST-as-GET /authz/{id}`	Retrieve authorization state	✓
Authorization Deactivation	ACME	`POST /authz/{id}`	Cancel domain authorization	✓
Challenge Response	ACME	`POST /chall/{id}`	Signal readiness for validation	✓
Certificate Download	ACME	`POST-as-GET /cert/{id}`	Download issued certificate	✓
Certificate Revocation	ACME	`POST /revoke-cert`	Revoke a certificate	✓
OCSP Query	OCSP	`POST /ocsp`	Query certificate status	✓
CRL Download	CRL	`GET <crlDistributionPoint>`	Retrieve current revocation list	✓
			Total Testable Operations:	**17**

Protocol-Defined Error Conditions				
Error Type	**Protocol**	**Trigger Context**	**Semantic Meaning**	**Testable**
`accountDoesNotExist`	ACME	Various	Referenced account not found	✓
`alreadyRevoked`	ACME	`/revoke-cert`	Certificate previously revoked	✓
`badCSR`	ACME	`/finalize`	CSR rejected (e.g., weak key)	✓
`badNonce`	ACME	Various	Invalid or expired nonce	✓
`badPublicKey`	ACME	Various	JWS key unacceptable	✓
`badRevocationReason`	ACME	`/revoke-cert`	Invalid revocation reason code	✓
`badSignatureAlgorithm`	ACME	Various	Unsupported JWS algorithm	✓
`connection`	ACME	Challenge validation	Cannot reach challenge endpoint	✓
`dns`	ACME	DNS-01 validation	DNS query failure	✓
`incorrectResponse`	ACME	Challenge validation	Challenge token mismatch	✓
`invalidContact`	ACME	`/newAccount`	Malformed contact URI	✓
`malformed`	ACME	Various	Syntactically invalid request	✓
`orderNotReady`	ACME	`/finalize`	Authorizations incomplete	✓
`rateLimited`	ACME	Any	Client exceeded rate limit	✓
`unauthorized`	ACME	Various	Insufficient permissions	✓
`unsupportedContact`	ACME	`/newAccount`	Unsupported URI scheme	✓
			Total Testable Errors:	**16**

fields with values drawn from error-triggering domains in IGR to elicit specification-defined failures (e.g., replayed nonces or policy-violating identifiers).

Conformance checking. After each request is executed, CAFUZZ checks observed behavior against the protocol specification using three complementary checks: (i) *state conformance*, verifying that the response corresponds to a permitted state transition from the current execution state; (ii) *schema conformance*, verifying that required response fields are present and well-formed according to RS; and (iii) *error conformance*, verifying that error responses match the RFC-defined error type and HTTP status for the triggering operation. Any mismatch — such as an unexpected HTTP status, incorrect error type, missing mandatory fields, or an invalid state transition — is recorded as a protocol deviation.

4) Working Example: We illustrate CAFUZZ's end-to-end workflow using the protocol deviation in order finalization shown in Figure 3 (bottom), where a CA returns an internal server error instead of a specification-defined policy error.

Stage 1: Protocol specification generation. From RFC 8555, CAFUZZ extracts a Protocol State Model (PSM) that includes the `Order` resource lifecycle and the `finalizeOrder` operation. The PSM specifies that an order in the `ready` state may transition to `valid` upon successful finalization, or return a protocol-defined error (e.g., `rejectedIdentifier`) if the requested identifiers violate CA policy (RFC 8555 §6.7). The corresponding Request Schema (RS) defines the structure of `POST /acme/finalize/{orderId}`, including a JWS-protected payload containing a CSR. Input Generation Rules (IGR) include both valid CSRs and adversarial-but-valid CSRs containing reserved identifiers (e.g., `*.example.com`),

which are syntactically correct but prohibited by policy.

Stage 2: Request generation and conformance checking. Guided by the PSM, CAFuzz enumerates a protocol-permitted execution path that reaches a finalizable order. The generated path is `directory` → `newNonce` → `newAccount` → `newOrder` → `satisfyAuthorizations` → `finalizeOrder`. Using RS and IGR, CAFuzz constructs a semantically valid request sequence, including a correctly authenticated `finalizeOrder` request whose CSR contains a reserved identifier. Upon execution, the CA responds with an HTTP 500 `serverInternal` error. During conformance checking, CAFuzz compares the observed response against the PSM and detects a violation. The RFC requires a permanent policy error (e.g., `rejectedIdentifier`, HTTP 403) rather than an internal server failure, and this mismatch is recorded as a protocol deviation.

Stage 3: Mock challenge handling. To reach the `ready` order state deterministically, CAFuzz uses mock HTTP-01, DNS-01, and TLS-ALPN-01 responders to satisfy all required authorizations. This eliminates reliance on external infrastructure and ensures that the detected deviation is attributable solely to order finalization logic rather than challenge nondeterminism.

Together, this example demonstrates how CAFuzz systematically generates a valid protocol execution, injects a specification-relevant adversarial input, and detects a concrete deviation from RFC-defined error semantics that is missed by developer-written tests.

E. CAFuzz *Implementation and Experiment Procedure*

Implementation. CAFuzz is implemented in Python 3. HTTP interactions with CA servers are handled using the `requests` library, while cryptographic functionality required for ACME—such as JSON Web Signature (JWS) generation, key management, and CSR construction—is implemented using the `cryptography` and `jose` libraries. Protocol specifications, including the Protocol State Model (PSM), Request Schemas (RS), and Input Generation Rules (IGR), are represented in normalized JSON and consumed directly by the test generation engine.

For protocol specification generation, CAFuzz uses GPT-4.1 strictly as a structured extraction aid. The model is accessed via the `litellm` interface with deterministic decoding and is used only during an offline extraction phase. The LLM is not involved in test generation, execution, or evaluation.

Deployment. Each tested CA is deployed in an isolated Docker container using its standard configuration. To enable deterministic traversal of ACME validation workflows, CAFuzz deploys lightweight mock responders for HTTP-01, DNS-01, and TLS-ALPN-01 challenges. These responders supply challenge tokens on demand and eliminate reliance on external infrastructure. Where supported, CAs are instrumented for statement coverage collection using Go's `-cover` or Java's JaCoCo, restricted to CA-specific code paths.

Test execution. For each CA, CAFuzz executes a complete protocol exploration campaign by deterministically traversing all enumerated protocol paths. Test generation does not rely on coverage feedback, random mutation, or probabilistic heuristics.

TABLE IV: *While CA developers write unit and integration tests, they often omit end-to-end (E2E) protocol tests that validate protocol-defined operations and error handling.* ***Legend:*** *✓: tests present; ✗: test absent.*

CA	Unit Tests	Integration Tests	E2E Protocol Tests	
			Protocol Operations	Protocol Errors
Boulder	✓	✓	✓	✓
Vault	✓	✓	✓	✓
Smallstep	✓	✓	✗	✗
Xipki	✓	✓	✗	✗

TABLE V: *A significantly lower DDR for all CA systems indicates that developers do not update tests while fixing bugs, creating a potential test gap.*

CA	#Closed Issues	Defect Detection Ratio (DDR)
Boulder	44	$12/44 = 27.3\%$
Vault	38	$11/38 = 28.9\%$
Smallstep	9	$3/9 = 33.3\%$
Xipki	3	$0/3 = 0.0\%$

All request–response traces, protocol coverage metrics, and detected specification deviations are logged for offline analysis. A full campaign—including container startup, protocol exploration, and teardown—completes in approximately 30–40 minutes per CA.

Infrastructure. All experiments were conducted on a dedicated server equipped with a 12-core Intel Xeon E5-2618L v3 (2.30 GHz), 64 GB RAM, running Ubuntu 20.04.

V. Evaluation

This section describes our evaluation results in terms of the three research questions we ask.

A. RQ1: Developer Test Types and Their Evolution with Bug Fixes

As summarized in Table IV, all CAs include unit and integration tests, but only Boulder and Vault implement end-to-end (E2E) protocol tests that cover both success and error paths. Smallstep and Xipki lack such tests entirely. Without E2E tests, subtle bugs arising from component interactions may go undetected. This can lead to failures in compliant ACME clients, problems that unit or integration tests are not designed to catch.

In addition to manually written tests, several CAs incorporate auxiliary tools to improve code quality and security. These include linters, semantic analyzers, and security checkers. For example, Boulder uses CodeQL and an extensive set of linters (e.g., `gosec`, `staticcheck`, `govet`, `revive`) and many more. Vault uses CodeQL, and applies multiple linters through `golangci-lint`. Smallstep includes CodeQL integration. While these tools are not designed to validate protocol behavior directly, they demonstrate attention to code quality, correctness, and security practices across different CA ecosystems.

To understand how developer test suites evolve with bug fixes, we analyzed the Defect Detection Ratio (DDR) (percentage

of bug-fix commits that add or update tests) of the test suite. As shown in Table VI, all CAs exhibit a low DDR, ranging from 0% for Xipki to 33.3% for Smallstep CA. This low ratio indicates that when externally reported bugs are fixed, developers rarely add a corresponding test case to prevent regression. This practice suggests that test development is primarily focused on new features rather than on hardening the software against past failures, leaving the CAs vulnerable to recurring bugs.

> Developer-written tests for CAs primarily focus on component-level functionality and new features. This creates two critical blind spots: a lack of holistic, end-to-end protocol validation and a failure to systematically build a regression suite from past bugs, leaving them vulnerable to recurring and systemic failures. (RQ1)

B. RQ2: Protocol Compliance and Developer Test Effectiveness

Dev column in Table VI shows the effectiveness of the developer-written tests in terms of the operation coverage, error coverage, and statement coverage metrics (recall §IV-C). Analyzing operation and error coverage reveals that developer-written tests largely focus on the "happy path" of certificate issuance. For instance, developer-written tests in Boulder do not exercise *Key Rollover* or *Account Deactivation* operations, and Boulder does not expose a CRL download endpoint, making CRL-related functionality untestable in that system. Vault tests on the other hand do not test *Account Update*, *Key Rollover*, *Authorization Deactivation*, *OCSP Query* and *CRL Download* operations. Smallstep and Xipki developers do not implement any end-to-end protocol tests, resulting in zero operation coverage.

Considering error coverage, developer-written tests in all CAs fail to cover most protocol-defined error conditions. For instance, Boulder tests neglect critical failure types such as *badCSR*, *badNonce*, and *badRevocationReason*. Vault tests do not test *badNonce*, *badPublicKey*, *badRevocationReason*, *badSignatureAlgorithm*, *invalidContact*, and *unsupportedContact*. The gap is even more stark for Smallstep and Xipki, which lack end-to-end protocol tests, leaving all 17 operations and 16 error types completely unverified at the protocol level.

A key example of this testing gap is the *account deactivation* feature. This operation, critical for account management and security, is implemented in multiple CAs but is not covered by any developer-written test in any of the CAs we analyzed. This demonstrates that even fully implemented features can be left untested, creating a false sense of security and a high risk of runtime failures or regressions.

Statement coverage further reflects the absence of protocol-level testing in some systems. For Smallstep and Xipki, developers do not implement any tests to verify end-to-end protocol interactions leading to the statement coverage of 0.0%.

Boulder's statement coverage is lower because its test environment does not trigger several production-level features. Although core protocol operations are exercised, components such as *CDN caching*, *database management*, and *Certificate Transparency logging* remain untested, reducing the overall statement coverage.

Vault exhibits a similar but more pronounced effect. ACME and PKI functionality constitute only a small portion of Vault's broader secrets-management codebase, which includes numerous subsystems unrelated to certificate issuance (e.g., authentication backends, secret engines, storage layers, and policy enforcement). Because our evaluation—and developer-written CA tests—exercise only the PKI/ACME components, large portions of Vault's codebase are intentionally not executed, resulting in low overall statement coverage. This lower coverage therefore reflects the scope of the tested functionality rather than insufficient testing of the CA logic itself.

> Developer tests tend to focus on common *happy path* workflows, often leaving error-handling logic and less frequently used protocol features under-tested. As a result, significant portions of the RFCs may remain unverified. (RQ2)

C. RQ3: Effectiveness of CAFuzz in Enhancing Developer-Written CA Tests

The **CAFuzz** column in Table VI summarizes the effectiveness of CAFuzz-generated tests in terms of *operation coverage*, *error coverage*, *statement coverage*, and *bugs detected* (recall §IV-B). Across all four evaluated CA implementations, CAFuzz achieves **complete operation and error coverage** and substantially improves statement coverage compared to developer-written end-to-end (E2E) protocol tests.

Comparing CAFuzz-generated tests with developer-written tests (Dev) shows that CAFuzz consistently improves all coverage metrics and, in some cases, provides the *only* systematic protocol-level testing. Below we describe the analysis of results for each metric.

Comparing the effectiveness of CAFuzz-generated tests with developer-written ones in terms of Operation, Error, and Statement Coverage shows that CAFuzz significantly improves all coverages, often serving as the only source of such testing for systems like Smallstep and Xipki. Below we describe the analysis of results for each metric.

a) Operation Coverage: CAFuzz significantly improves operation coverage across all evaluated CAs. In **Boulder**, operation coverage increases from **87.5%** with developer-written tests to **100%** with CAFuzz. In **Vault**, CAFuzz improves operation coverage from **70.6%** to **100%**. For both **Smallstep** and **Xipki**, which lack developer-written end-to-end ACME protocol tests, operation coverage improves from **0%** to **100%**.

These improvements stem from CAFuzz's ability to systematically exercise stateful and low-frequency protocol operations that are typically omitted from developer-written tests. Many ACME operations—such as *account deactivation*, *key rollover*, *authorization deactivation*, and *revocation*-related workflows—are not part of the common certificate issuance "happy path" and require valid multi-step state progressions to

TABLE VI: ***Coverage and Bug Detection Results.*** *We compare* developer-written end-to-end (E2E) protocol tests only *(Dev), AFLNet, and* CAFUZZ *across operation coverage, error coverage, and statement coverage.* CAFUZZ *achieves full operation and error coverage across all four CA implementations and significantly improves statement coverage compared to developer-written tests and AFLNet. Additionally,* CAFUZZ *detected new bugs (Unknown) and successfully rediscovers known protocol-related bugs when replaying buggy commits.*

	Operation Coverage			Error Coverage			Statement Coverage			Bugs Detected		
CA	Dev	*AFLNet*	CAFUZZ	Dev	*AFLNet*	CAFUZZ	Dev	*AFLNet*	CAFUZZ	Known	Unknown	Total
Boulder	87.5%	12.5%	100.0%	43.8%	25.0%	100.0%	41.6%	3.8%	47.3%	24	1	25
Vault	70.6%	11.8%	100.0%	31.3%	25.0%	100.0%	20.5%	1.6%	24.9%	12	1	13
Smallstep	0.0%	11.8%	100.0%	0.0%	25.0%	100.0%	0.0%	2.5%	35.5%	4	0	4
Xipki	0.0%	11.8%	100.0%	0.0%	25.0%	100.0%	0.0%	2.1%	41.4%	3	1	4
Average	39.53%	11.98%	100.0%	18.78%	25.0%	100.0%	15.53%	2.5%	37.28%	10.75	0.75	11.5

reach. As a result, developers often deprioritize testing these operations. By explicitly enumerating RFC-permitted protocol paths, CAFUZZ reliably reaches and validates these security-relevant operations, closing gaps left by developer-written tests.

b) Error Coverage: CAFUZZ achieves **100% error coverage** across all four CAs, representing substantial improvements over developer-written tests. In **Boulder**, error coverage increases from **43.8%** to **100%**, and in **Vault**, from **31.3%** to **100%**. For **Smallstep** and **Xipki**, there are no developer-written end-to-end tests, hence have **0%** error coverage, while CAFUZZ attains full coverage.

This improvement is enabled by CAFUZZ's deliberate construction of *valid-but-adversarial* protocol interactions that trigger specification-defined failure conditions. Many ACME error responses arise only after successful completion of prior protocol steps and require well-formed requests with subtle semantic violations (e.g., nonce reuse, policy rejection, or malformed-but-syntactically-valid CSRs). Developer-written tests typically terminate at successful execution paths. By combining state-aware path enumeration with targeted input generation, CAFUZZ systematically exercises deep error-handling logic that would otherwise remain untested.

c) Statement Coverage: CAFUZZ also substantially improves statement coverage across all systems. In **Boulder**, statement coverage increases from **41.6%** to **47.3%**, and in **Vault**, from **20.5%** to **24.9%**. More notably, for **Smallstep** and **Xipki**, statement coverage increases from **0%** to **35.5%** and **41.4%**, respectively. On average, CAFUZZ improves statement coverage by **1.4 times** relative to developer-written tests.

These gains reflect CAFUZZ's ability to drive CA implementations through a broader range of protocol-level behaviors, including state transitions, error handling, and edge-case validation logic. At the same time, developer-written tests remain complementary by covering implementation-specific logic and deployment policies that are not inferable from protocol specifications, such as rate-limiting exemptions or multi-perspective validation. Together, these results demonstrate that protocol-aware test generation and developer-written tests address distinct but complementary aspects of CA behavior.

d) Bugs Detected.: Beyond coverage improvements, CAFUZZ detects both previously known and previously undocumented protocol-related bugs across the evaluated CAs (Table VI, *Bugs Detected*). In total, CAFUZZ rediscovered **43 known protocol-related bugs** and uncovered **3 previously unknown bugs**.

e) Known Protocol Bug Dataset Construction: To evaluate whether CAFUZZ can rediscover historically observed ACME protocol failures, we constructed a dataset of known protocol-related bugs from the issue trackers of the evaluated CA implementations. All repositories use explicit bug labels (e.g., `bug`, `kind/bug`), often combined with subsystem-specific tags (e.g., ACME or PKI-related areas), which we use as an initial filter. From this set, we apply a systematic three-stage filtering pipeline to identify bugs that are *protocol-testable* via externally observable ACME interactions. The pipeline progressively refines candidates using (i) label-based filtering to retain ACME/PKI-relevant bugs while excluding documentation, testing, and infrastructure issues; (ii) file-path filtering based on the set of files modified in the issue's associated fixing changes, retaining only bugs affecting protocol-relevant modules such as ACME frontends, validation, nonce handling, and policy enforcement; and (iii) content-based scoring using protocol-specific keywords (e.g., ACME operations, challenge types, state transitions, and RFC compliance terms) while penalizing infrastructure-related and configuration terminology. Issues whose combined score exceeds a fixed threshold are classified as protocol-testable and included in the dataset.

For each selected issue, we attempt to reproduce the bug by checking out the corresponding buggy revision (or the revision immediately preceding the fix) and replaying the relevant ACME interactions using CAFUZZ-generated tests. Some historical bugs could not be reproduced because the corresponding versions could not be reliably built or executed due to dependency drift, or environmental incompatibilities. This limitation is well documented in prior work on software defect reproducibility [28] and is orthogonal to the effectiveness of the test generation approach itself. We therefore report rediscovery results only for bugs whose buggy versions could be executed in our experimental environment.

f) Known Protocol Bugs Detected: CAFUZZ successfully rediscovered **43 known protocol-related bugs** across the four CAs (Table VI, *Known* column). Some known bugs could not be rediscovered because the corresponding CA versions could not be reliably built or executed due to deprecated dependencies, missing infrastructure components, or environmental drift. As prior work has shown [28], reproducibility is a fundamental challenge in executable software defect datasets and is often orthogonal to the effectiveness of the testing approach itself.

In **Boulder**, CAFUZZ rediscovers two representative ACME specification deviations. First, Boulder omits the

mandatory `orders` field in the `account` object, violating RFC8555§7.1.2 (GitHub Issue #3335)[1]. Second, Boulder rejects the request parameters `notBefore` and `notAfter` fields in `newOrder` requests, despite these fields being explicitly defined by the specification. This behavior is acknowledged in Boulder's ACME divergences documentation[2]. Although these bugs do not always cause immediate failures, they violate client expectations and undermine strict protocol compliance.

Across all systems, the ability to rediscover known bugs confirms that CAFUZZ-generated tests capture realistic protocol interactions and regressions that developer-written tests fail to encode.

g) Unknown Protocol Bugs Detected: In addition to rediscovering known issues, CAFUZZ uncovers **three previously undocumented specification violations**, demonstrating its ability to expose subtle semantic errors in mature CA implementations.

In **Boulder**, CAFUZZ identifies an error-handling violation during order finalization for reserved domains (e.g., `*.example.com`). Instead of returning a permanent policy error such as `rejectedIdentifier` (HTTP403) as required by RFC8555 §6.7, Boulder responds with a transient `serverInternal` error (HTTP 500). This misclassification causes ACME clients to misinterpret policy rejections as retryable failures. We reported this issue to the Boulder developers[3].

In **Vault**, CAFUZZ discovers that successful `newOrder` responses intermittently omit the mandatory `Location` header while returning HTTP201. This violates RFC7231 §6.3.2 and RFC8555§7.4 and can break client-side state tracking. We reported this issue to the Vault project.

In **Xipki**, CAFUZZ exposes an incorrect implementation of DNS-01 challenge validation. Rather than querying `_acme-challenge.domain` as mandated by RFC8555§8.4, Xipki queries TXT records at the base domain, leading to systematic challenge failures for compliant clients. We reported this issue to the Xipki maintainers[4].

> CAFUZZ significantly enhances developer-written tests across all four CAs by achieving **100% operation and error coverage** and substantially improving statement coverage (**140.64% improvement on average**). Beyond coverage gains, CAFUZZ rediscovered **43 known** and uncovered **3 previously unknown** protocol-related bugs, demonstrating its effectiveness in closing critical testing gaps in CA implementations. (RQ3)

D. RQ4: Comparison with the State-of-the-Art

This research question compares CAFUZZ against state-of-the-art automated testing techniques for stateful network protocols. For this, we selected *AFLNet* [18] as a representative state-of-the-art coverage-guided fuzzer and additionally discuss **boofuzz** [19] as a commonly used grammar-based protocol fuzzing framework.

a) AFLNet (Coverage-Guided Stateful Fuzzing): AFLNet extends AFL with protocol-state awareness and is widely used for testing stateful network services. It represents the strongest available baseline for automated testing that combines fuzzing with limited state modeling.

To apply*AFLNet*to ACME CA implementations, we constructed a custom harness that exposes the CA's ACME HTTP interface as a fuzzing target. Because CA implementations are typically complex, multi-process systems and are not easily recompiled with compiler-based coverage instrumentation, we instrumented all CAs using QEMU-based instrumentation, as supported by AFLNet. This allows collection of edge coverage without modifying or recompiling the CA source code, ensuring a fair and realistic evaluation setting.

We additionally seeded*AFLNet*with syntactically valid ACME request templates to avoid immediate rejection at the parsing layer. Nevertheless,*AFLNet*remains fundamentally *mutation-driven* and relies on random mutations to satisfy ACME's cryptographic and state-dependent constraints.

Table VI shows that*AFLNet*achieves limited coverage across all evaluated CA implementations. On average,*AFLNet*reaches only **11.98% operation coverage** and **25.0% error coverage**, compared to **100%** for CAFUZZ. Statement coverage under*AFLNet*remains similarly low, averaging **2.5%**.

For example, in **Boulder**,*AFLNet*exercises **12.5%** of protocol operations and **25.0%** of error cases, while CAFUZZ achieves complete coverage. In **Smallstep** and **Xipki**,*AFLNet*reaches only **11.8%** operation coverage and fails to progress beyond initial account and order creation workflows. AFLNet's limited coverage is due to its reliance on random mutations over raw protocol messages, which prevents it from constructing cryptographically valid ACME requests. In practice,*AFLNet*reaches only unauthenticated endpoints such as directory discovery and nonce retrieval, and fails to exercise operations that require correctly formed JWS-signed requests, including order finalization, challenge responses, certificate download, and revocation. Consequently,*AFLNet*triggers only a small subset of shallow errors (e.g., `malformed`, `badNonce`, `badPublicKey`, `rateLimited`) and misses protocol-specific error conditions such as `badCSR`, `orderNotReady`, `incorrectResponse`, and DNS or CAA related failures, all of which CAFUZZ systematically covers.

b) boofuzz (Grammar-Based Protocol Fuzzing): Boofuzz [19] is a widely used protocol fuzzing framework that allows developers to define message grammars and sequencing logic. However, unlike AFLNet,*Boofuzz* is not suitable for testing ACME-based CAs because ACME is a *strongly stateful and cryptographically authenticated protocol* with strict semantic and external validation requirements. Each ACME request must be correctly sequenced, bound to a valid account key, signed using JSON Web Signatures, and include a fresh server-issued nonce; arbitrary mutation of message fields, as performed by*Boofuzz*, almost always breaks these invariants and causes requests to be rejected at early validation stages. Moreover, ACME relies on out-of-band challenge validation mechanisms

[1]https://github.com/letsencrypt/boulder/issues/3335
[2]https://github.com/letsencrypt/boulder/blob/main/docs/acme-divergences.md
[3]https://github.com/letsencrypt/boulder/issues/8536
[4]https://github.com/xipki/xipki/issues/343

such as DNS and HTTP, which require coordinated control of external infrastructure and cannot be exercised through simple network input mutation. As a result,*Boofuzz*-style syntactic fuzzing is filtered by TLS, JSON, and cryptographic checks before reaching core CA logic, yielding no meaningful behavioral feedback or coverage for effective testing. For this reason, we do not evaluate*Boofuzz* quantitatively and instead treat it as a conceptual baseline representative of grammar-based fuzzing approaches.

c) Why Fuzzing-Based Approaches Fall Short: Both*AFLNet*and *boofuzz* face fundamental challenges when applied to ACME.

First, ACME operations are guarded by *long, stateful preconditions*. Reaching security-relevant behaviors such as revocation or account deactivation requires successful completion of multiple authenticated protocol steps. Mutation-based fuzzing struggles to preserve these dependencies across iterations.

Second, ACME enforces strong *semantic and cryptographic constraints*, including fresh nonces, valid JWS signatures, and consistent account bindings. Violations of these constraints cause requests to be rejected early, preventing fuzzers from reaching deeper protocol logic where most semantic errors reside.

Finally, many protocol-level error conditions arise only after *nearly correct* executions (e.g., policy rejection after successful authorization). These behaviors are unlikely to be discovered through random mutation or grammar-level fuzzing but are naturally exercised by CAFUZZ's state-aware enumeration of RFC-permitted protocol paths.

> CAFUZZ outperforms*AFLNet*by achieving substantially higher coverage and detecting both known and previously unknown ACME protocol violations. By modeling RFC-defined protocol states and behaviors, CAFUZZ addresses fundamental limitations of mutation- and grammar-based fuzzing when applied to security-critical, stateful protocols. (RQ4)

VI. Discussion

Our study reveals substantial variability in how Certificate Authorities (CAs) test protocol-level behavior. Across all evaluated systems, developer effort is concentrated on unit and integration tests, while end-to-end protocol testing—particularly for error handling and stateful workflows—is limited or entirely absent. Moreover, we observe that test suites are updated far less frequently than bug fixes, creating persistent gaps in regression coverage. As a result, rare operations, error-handling paths, and adversarial inputs remain under-tested, allowing specification deviations to persist even in mature and widely deployed implementations.

CAFUZZ demonstrates that protocol-aware black-box testing can effectively complement existing developer practices. By systematically exercising protocol-defined behaviors using specification-derived models, CAFUZZ uncovers untested operations, increases protocol and code coverage, and detects semantic inconsistencies that are unlikely to surface through traditional testing approaches. Because CAFUZZ operates entirely through the public protocol interface, it can be integrated into existing CI pipelines without requiring access to internal code or intrusive instrumentation.

While our evaluation focuses on ACME, OCSP, and CRL, the underlying methodology is broadly applicable. Any standardized, stateful protocol with well-defined request–response semantics and explicit error conditions — such as OAuth, OpenID Connect, or WebAuthn — presents a similar testing challenge. The protocol modeling and input generation approach used by CAFUZZ generalizes naturally to these domains, suggesting a broader role for specification-guided testing in the validation of security-critical services.

A. Future Research Directions

This work opens several avenues for future research at the intersection of protocol specifications, automated testing, and large language models (LLMs).

First, the LLM-assisted specification extraction pipeline used in CAFUZZ can be extended beyond ACME to other standardized, stateful security protocols. Many RFC-defined protocols (e.g., OAuth, OpenID Connect, SCEP, CMP, and emerging certificate lifecycle standards) encode complex state machines and semantic constraints that are difficult to test exhaustively using manual approaches. Generalizing the LLM pipeline to automatically extract protocol state models, request schemas, and semantic constraints from diverse RFCs would enable protocol-aware testing across a broader class of security-critical systems. Beyond test generation, such structured protocol representations could support downstream tasks such as automated conformance checking, protocol documentation validation, and cross-implementation comparison.

Second, CAFUZZ's protocol models provide a natural foundation for automatic harness generation and continuous testing. Future work could integrate state-aware harnesses into continuous integration pipelines or long-running fuzzing campaigns, enabling hybrid testing workflows that combine systematic protocol exploration with coverage-guided fuzzing. Such integration would allow protocol-aware test cases to act as high-quality seeds and state reset mechanisms for fuzzers, improving their ability to reach deep protocol states over time.

Third, while CAFUZZ currently operates in a black-box setting, combining protocol-derived state models with white-box analysis presents an opportunity for deeper semantic bug detection. Future work could leverage internal program representations, control-flow information, or symbolic execution to detect inconsistencies between RFC-specified semantics and implementation behavior, such as incorrect error classification, missing state transitions, or unreachable protocol states. This hybrid approach could further bridge the gap between protocol specifications and real-world implementations, enabling more precise detection of semantic and compliance-related vulnerabilities.

Finally, researchers could go beyond automated testing to develop automated fault localization [29] and program repair [30], [31] techniques tailored to CAs, addressing challenges such as localizing faults across different CA endpoints, handlers, and server interactions, and automatically generating patches that preserve RFC specifications and backward compatibility.

B. Limitations and Threats to Validity

CAFUZZ assumes a fully observable and responsive black-box interface. Some protocol behaviors, including challenge expiration timing, background CRL generation, and asynchronous revocation propagation, are difficult or impossible to test deterministically and are therefore out of scope. In addition, our protocol models are derived from RFC specifications; implementation-specific extensions or undocumented behaviors are not explicitly modeled unless they manifest as observable deviations.

We address threats to *internal validity* by executing multiple independent test runs per CA and manually inspecting request–response traces, coverage reports, and detected deviations. Threats to *external validity* are mitigated by selecting multiple independently developed and actively maintained CA implementations spanning both public and enterprise deployments. To reduce threats to *construct validity*, we employ standard metrics—operation coverage, error coverage, statement coverage, and bug rediscovery rate-that are widely used in software testing research. Nevertheless, these metrics don't fully capture qualitative aspects such as semantic correctness, long-term maintainability of tests, or developer effort.

Finally, while CAFUZZ demonstrably improves protocol coverage and uncovers bugs missed by developer-written tests, we do not claim that it replaces manual testing. Instead, our results suggest that protocol-aware automated testing is a practical and effective complement to existing testing practices, particularly for exercising low-frequency, state-dependent, and security-sensitive behaviors.

VII. Related Work

Security of SSL/TLS Implementations. Numerous studies have investigated SSL/TLS protocol implementations to identify programming flaws, insecure defaults, and vulnerabilities in cryptographic API usage [10], [11], [12]. For example, SSLINT [10] and CRYPTOGUARD [11] use static analysis to uncover SSL/TLS misuse and misconfiguration. While valuable, these tools focus on general-purpose applications and client libraries rather than CA-side logic or certificate issuance workflows.

Certificate Validation in Clients. Prior work has demonstrated how incorrect validation logic in browsers and non-browser clients can expose users to man-in-the-middle attacks [13], [14], [32], [33], [34]. Tools like FRANKENCERT [14] and SYMCERTS [32] use differential testing and symbolic execution to uncover inconsistencies across implementations. Our work complements this line of research by evaluating how server-side CA software validates domain ownership, applies issuance policy, and tests error handling—distinct from the certificate acceptance logic studied on the client side.

Formal Analysis of ACME. Bhargavan et al. [8], [1] formally modeled the ACME protocol to prove security guarantees and uncover design-level flaws, leading to the development of ACMEv2. While these efforts strengthen the protocol's formal correctness, they do not address whether real-world ACME implementations conform to these guarantees or are adequately tested.

Protocol Fuzzing and Test Generation. Network protocol fuzzing has evolved from grammar-based approaches to coverage-guided greybox fuzzing. Tools like AFL and LibFuzzer [35] are widely used but lack state awareness. Protocol-specific fuzzers such as AFLNET [18] integrate state models to better explore transitions. However, ACME introduces unique challenges, including nonce handling, cryptographic constraints, and authenticated state machines. To our knowledge, no prior work has systematically fuzzed production-grade ACME implementations or CA logic.

State-Aware Testing of Protocol Implementations. Fuzzing frameworks based on state-machine have been developed to explore complex multi-step interactions in protocols, particularly in the context of web APIs and transport protocols [36]. However, these systems often assume unauthenticated or stateless interactions. In contrast, CA systems implement authenticated, stateful workflows across multiple resources. Our work extends state-aware testing to the domain of PKI, where inputs must adhere to cryptographic and temporal constraints and must maintain consistency across protocol phases.

Revocation, Certificate Transparency, and Misissuance Studies. Prior work has analyzed both the operational effectiveness of revocation mechanisms and the prevalence of certificate misissuance in the wild. Sosnowski et al.[37] investigates the impact of the new CRLs on certificate revocation research, while Kumar et al. [9] identified over 10,000 misissued certificates in Certificate Transparency (CT) logs by applying RFC-derived validation rules. These studies demonstrate that misissuance and revocation failures are real-world problems. However, existing work focuses on ecosystem-level monitoring or post-hoc detection, whereas our work complements these efforts by assessing whether CA systems themselves are adequately tested to prevent such failures.

Despite extensive research on protocol correctness, client validation, and fuzzing, the testing of CA systems remains largely unexplored. We bridge this gap by conducting empirical analysis of CA tests and developing protocol-aware test generation.

VIII. Contributions

We presented the first empirical analysis of how well certificate authorities are tested by developers, along with CAFUZZ, an automated protocol-aware black-box testing technique designed to complement developer-written tests. Our results underscore the value of protocol-driven test generation for critical infrastructure protocols and motivate further research into semantically aware, automated testing frameworks. In the future, we plan to automate the extraction of protocol state models from RFCs or implementation traces, improve support for non-deterministic behaviors, extend CAFUZZ to other protocols, and integrate it into CI pipelines.

References

[1] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "An In-Depth Symbolic Security Analysis of the ACME Standard," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 2601–2617.

[2] A. Ayer, "Security Vulnerabilities in Smallstep PKI Software," Dec. 2020.
[3] L. E. C. Support, "2020.02.29 CAA Rechecking Bug - Incidents," https://community.letsencrypt.org/t/2020-02-29-caa-rechecking-bug/114591, Feb. 2020.
[4] M. Bernhard, J. Sharman, C. Z. Acemyan, P. Kortum, D. S. Wallach, and J. A. Halderman, "On the Usability of HTTPS Deployment," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–10.
[5] MozillaWiki contributors, "CA/Symantec Issues - MozillaWiki," 2018. [Online]. Available: https://wiki.mozilla.org/CA/Symantec_Issues
[6] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic certificate management environment (ACME)," RFC 8555, Mar. 2019.
[7] J. Aas, R. Barnes, B. Case, Z. Durumeric, P. Eckersley, A. Flores-López, J. A. Halderman, J. Hoffman-Andrews, J. Kasten, E. Rescorla, S. Schoen, and B. Warren, "Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 2473–2487. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3363192
[8] K. Bhargavan, A. Delignat-Lavaud, and N. Kobeissi, "Formal Modeling and Verification for Domain Validation and ACME," in *Financial Cryptography and Data Security*, A. Kiayias, Ed. Cham: Springer International Publishing, 2017, pp. 561–578.
[9] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. D. Bailey, "Tracking certificate misissuance in the wild," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. San Francisco, CA: IEEE Computer Society, 2018, pp. 785–798. [Online]. Available: https://doi.org/10.1109/SP.2018.00015
[10] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL Usage in Applications with SSLINT," in *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 519–534. [Online]. Available: https://ieeexplore.ieee.org/document/7163045/
[11] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 2455–2472. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3345659
[12] M. Oltrogge, N. Huaman, S. Klivan, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting TLS (In)Security in android applications," in *30th USENIX Security Symposium (USENIX Security 21)*. Virtual: USENIX Association, Aug. 2021, pp. 4347–4364. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge
[13] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 38–49. [Online]. Available: https://doi.org/10.1145/2382196.2382204
[14] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," in *2014 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE, May 2014, pp. 114–129, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6956560
[15] C. Chen, P. Ren, Z. Duan, C. Tian, X. Lu, and B. Yu, "SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 967–979. [Online]. Available: https://doi.org/10.1145/3597926.3598110
[16] M. Luo, B. Feng, L. Lu, E. Kirda, and K. Ren, "On the Complexity of the Web's PKI: Evaluating Certificate Validation of Mobile Browsers," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 1, pp. 419–433, Jan. 2024, conference Name: IEEE Transactions on Dependable and Secure Computing. [Online]. Available: https://ieeexplore.ieee.org/document/10066507
[17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USENIX Association, 2020, p. 10.
[18] R. Meng, V. Pham, M. Böhme, and A. Roychoudhury, "Aflnet five years later: On coverage-guided protocol fuzzing," *IEEE Trans. Software Eng.*, vol. 51, no. 4, pp. 960–974, 2025. [Online]. Available: https://doi.org/10.1109/TSE.2025.3535925
[19] J. Pereyda, "Jtpereyda/boofuzz." [Online]. Available: https://github.com/jtpereyda/boofuzz
[20] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile," RFC 5280, May 2008.
[21] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and D. C. Adams, "X.509 internet public key infrastructure online certificate status protocol - OCSP," RFC 6960, Jun. 2013.
[22] L. E. (ISRG), "letsencrypt/boulder," May 2024, original-date: 2014-12-21T00:29:54Z. [Online]. Available: https://github.com/letsencrypt/boulder
[23] W3Techs, "Usage Statistics and Market Share of SSL Certificate Authorities for Websites, May 2024," 2024. [Online]. Available: https://w3techs.com/technologies/overview/ssl_certificate
[24] Chrome for Developers. (2025) Overview of CrUX | Chrome UX Report. Chrome for Developers. [Online]. Available: https://developer.chrome.com/docs/crux
[25] HashiCorp, "hashicorp/vault," May 2024, original-date: 2015-02-25T00:15:59Z. [Online]. Available: https://github.com/hashicorp/vault
[26] Smallstep, "smallstep/certificates," May 2024, original-date: 2018-11-01T03:44:28Z. [Online]. Available: https://github.com/smallstep/certificates
[27] L. Liao, "xipki/xipki," May 2024, original-date: 2014-03-29T20:16:04Z. [Online]. Available: https://github.com/xipki/xipki
[28] H.-N. Zhu and C. Rubio-González, "On the Reproducibility of Software Defect Datasets," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, pp. 2324–2335.
[29] M. Motwani and Y. Brun, "Better automatic program repair by using bug reports and tests together," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1225–1237.
[30] M. Motwani, "High-quality automated program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 309–314.
[31] ——, "High-quality automatic program repair," *UMass Dissertation*, 2022.
[32] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li, "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, May 2017, pp. 503–520. [Online]. Available: http://ieeexplore.ieee.org/document/7958595/
[33] J. Zhu, C. Wan, P. Nie, Y. Chen, and Z. Su, "Guided, Deep Testing of X.509 Certificate Validation via Coverage Transfer Graphs," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 243–254, iSSN: 2576-3148. [Online]. Available: https://ieeexplore.ieee.org/document/9240633
[34] P. Nie, C. Wan, J. Zhu, Z. Lin, Y. Chen, and Z. Su, "Coverage-directed Differential Testing of X.509 Certificate Validation in SSL/TLS Implementations," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 3:1–3:32, Feb. 2023. [Online]. Available: https://doi.org/10.1145/3510416
[35] LLVM contributors, "libFuzzer - a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation," 2025. [Online]. Available: https://llvm.org/docs/LibFuzzer.html
[36] G. V. Bochmann and A. Petrenko, "Protocol testing: Review of methods and relevance for software testing," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '94. New York, NY, USA: Association for Computing Machinery, Aug. 1994, pp. 109–124.
[37] M. Sosnowski, J. Zirngibl, P. Sattler, J. Aulbach, J. Lang, and G. Carle, "An internet-wide view on HTTPS certificate revocations: Observing the revival of crls via active TLS scans," in *IEEE European Symposium on Security and Privacy Workshops, EuroS&PW 2024, Vienna, Austria, July 8-12, 2024*. Vienna, Austria: IEEE, 2024, pp. 297–306. [Online]. Available: https://doi.org/10.1109/EuroSPW61312.2024.00038