Sentinel: Improving Web Security Testing with AI Assistance

Anonymous Author(s)

Abstract

By relying on numerous third-party libraries and frequent updates, web applications became increasingly complex. These libraries and updates often introduced hidden security vulnerabilities. Additonally, traditional automated security tools often flagged many potential issues without confirming whether they were real, leaving developers uncertain and increasing the risk of undetected threats. These false positives and ambiguous warnings made it difficult for organizations to prioritize security work, potentially resulting in operational disruptions, data breaches, or reputational harm. Ensuring reliable, actionable security insights was therefore critical to maintaining trust, protecting sensitive information, and supporting safe digital operations.

This report presented Sentinel, a next-generation security testing tool that provided clear, verifiable evidence of vulnerabilities in web applications. Sentinel systematically mapped applications, tested inputs, and confirmed issues with reproducible proofs, allowing developers and teams to identify real risks with confidence. It used an AI assistant in a controlled environment to suggest additional tests safely, while all results were logged and tracked for accountability. By combining automated scanning with rigorous verification and industry-aligned guidance, Sentinel reduced uncertainty, improved decision-making, and enabled organizations to address security risks efficiently and responsibly. The approach demonstrated how advanced, trustworthy tools could make web applications safer while supporting ethical, transparent, and reliable security practices.

Keywords

Secure Coding; Vulnerability Detection; Proof-of-Exploit; Reproducibility; Web Application Security; AI-Assisted Testing; Dynamic Application Security Testing (DAST); Large Language Models (LLMs)

1 Overview

Dynamic application security testing (DAST) helps teams find security weaknesses by interacting with a running web application. These tools are widely used because they test applications the same way an attacker would. One widely adopted open-source DAST tool is the Open Worldwide Application Security Project (OWASP) Zed Attack Proxy (ZAP), which provides automated scanners and a suite of manual testing tools to help developers identify vulnerabilities such as SQL injection, cross-site scripting, and insecure headers. However, even mature scanners like ZAP and commercial tools such as Burp Suite often produce warnings without clear proof that the issue can actually be exploited. Prior studies have noted this gap between academic research and practical, developer-focused tools [15].

This problem is intensified by the complexity of today's web ecosystems. Modern applications rely on large dependency chains, rapid framework updates, and frequent library changes that make assumptions about code behavior unreliable. Xu et al. show that vulnerabilities in reused third-party libraries can persist even after patching if downstream components are not re-verified [16]. Likewise, Hu et al.'s systematization of automated repair methods finds that many research prototypes fail to produce fixes that are trustworthy in real-world environments [4]. Together, these works highlight the need for tools that provide verifiable, actionable evidence, not just abstract vulnerability reports.

Sentinel is designed to address this need. Instead of simply listing potential issues, it performs structured tests and confirms each vulnerability using reproducible steps. Sentinel works in three stages. First, it maps an application's pages and inputs. Then it runs safe and targeted probes. Finally, it uses deterministic checks to verify whether a genuine issue exists. It focuses on high-impact problems such as missing security headers, unsafe cookie settings, cross-site scripting (XSS), cross-site request forgery (CSRF), and insecure direct object references (IDOR).

Each confirmed issue includes a reproducible proof-of-exploit, such as a minimal curl command or a small Catch2-based test case. Furthermore, it provides concise remediation guidance tied to common standards like the CWE Top 25 [10] and the OWASP Top 10 [13]. This ensures developers can both understand and verify every reported vulnerability.

Sentinel also fits cleanly into continuous-integration workflows. A policy-driven scoring system allows teams to decide when a build should be blocked based on the severity of confirmed findings. All results are captured in an append-only, hash-chained JSONL log, and all large language model (LLM) outputs are validated, hashed, and recorded in a structured manifest. The package includes tests, sample fixtures, and an evaluation notebook that reports precision, recall, and runtime statistics.

To remain adaptable, Sentinel uses a locally hosted LLM (via Ollama) to suggest additional test variations. Prior work by Risse and Böhme warns that machine-learning systems for vulnerability detection can behave unpredictably without strict safeguards [14]. In response, Sentinel validates all LLM-generated suggestions inside a sandbox and accepts only those that are safe, deterministic, and reproducible. Inspired by techniques from Huang et al.'s JAEX framework, which generates verified exploits using controlled dataflow analysis [5], Sentinel relies on structured templates and strict schemas to prevent unsafe test generation.

The remainder of this report is organized as follows: Section 2 surveys related research on DAST tools, automated repair systems, and web vulnerability scanners; Section 3 presents Sentinel's system design and architecture; Section 4 describes its implementation details; Section 5 reports evaluation results and performance metrics; Section 6 discusses ethical and security considerations; Section 7 reflects on lessons learned and limitations; and Section 8 outlines future extensions and integrations.

2 Related Work

Research on dynamic application security testing (DAST) has advanced significantly in the past decade, yet scanners still face persistent challenges with reproducibility, verifiability, and developer adoption. Singh et al. [15] survey contemporary DAST architectures and note that many systems rely heavily on pattern matching and heuristic rules, resulting in high false-positive rates. Widely used tools such as OWASP ZAP and Burp Suite suffer from non-deterministic behavior and lack reproducible proof-of-exploit generation, leaving developers unsure which reported vulnerabilities are genuinely exploitable [13]. These limitations highlight the need for DAST systems that provide both reliable detection and verifiable evidence.

Recent work has attempted to improve DAST effectiveness by enhancing automation and increasing execution-path coverage. Hybrid systems like FGVulDet pair fuzzing with neural vulnerability classification to reduce redundant exploration and boost detection accuracy [8]. Reinforcement learning-based approaches such as VULTURE dynamically navigate application states to uncover deeper and more complex vulnerabilities [16]. While these techniques improve coverage and efficiency, their outputs remain difficult to reproduce deterministically, and they provide limited support for integration into standard developer workflows.

Parallel research in automated vulnerability repair provides complementary insights into the challenges of interpretability and reproducibility. Hu et al. [4] demonstrate that many AI-based repair systems struggle to generalize beyond controlled benchmarks, restricting their practical use. Risse and Böhme [14] similarly show that deep learning–based vulnerability detectors often lack the explainability and auditability required for reliable decision-making. These findings reinforce the importance of tools that not only detect issues but also generate deterministic, verifiable evidence to support developer action.

The incorporation of large language models (LLMs) into vulnerability analysis introduces new opportunities and risks. Early systems demonstrate that LLMs can generate plausible exploit payloads, but their outputs are frequently unsafe or inconsistent without strong constraints. Huang et al.'s JAEX framework [5] shows that controlled dataflow analysis can enable safe, validated exploit generation, emphasizing structured and interpretable workflows. Sentinel extends this direction by integrating a local LLM pipeline through Ollama, where payload proposals pass through a sandboxed validation layer to ensure deterministic and legally compliant behavior.

Efforts to formalize vulnerability classification—such as the CWE Top 25 [10] and the OWASP Top 10 [13]—provide a shared foundation for categorizing findings and aligning security tools with industry expectations. Sentinel adopts these standards by mapping each detected issue to specific CWE and OWASP categories and by producing hash-chained JSONL reports that maintain transparent, tamper-evident logs. This combination of deterministic scanning, structured validation, and standards alignment situates Sentinel within an emerging generation of DAST systems that emphasize reproducibility, interpretability, and compatibility with continuous delivery pipelines.

3 Design and Implementation

Sentinel is structured as a modular DAST framework designed for reproducibility, safety, and auditability. Its architecture separates the scanning engine, decision logic, and reporting subsystems to support parallel development and independent testing. The design emphasizes three core properties: deterministic operation, verifiable artifacts, and safe integration with a local large language model (LLM). The LLM is treated as an assistive component rather than a decision oracle. All security decisions, scoring, and policy outcomes remain in deterministic, testable code, while the LLM is used only to propose additional test ideas and explanations that are then validated by the core engine. This reflects lessons from prior work on the limits of machine learning for vulnerability detection [14] and broader guidance on trustworthy AI [12].

At a high level, Sentinel operated in three sequential phases:

(1) **Exploration:** The crawler maps the target web application by enumerating endpoints, forms, and input parameters through safe HTTP requests.
(2) **Detection:** Each discovered input is tested using structured payloads defined in YAML templates. Responses are evaluated through deterministic oracles that identify signatures of vulnerabilities such as missing headers, reflected XSS, CSRF, and IDOR.
(3) **Verification and Reporting:** For each confirmed issue, Sentinel generates a minimal proof-of-exploit (for example, a `curl` command or Catch2 harness) and stores it in a tamper-evident JSONL log. A human-readable report is produced for developer consumption, and structured findings are exported so that CI pipelines can apply risk budgets and policies.

The system architecture consists of five main components:

(1) **Core Engine:** Implements the HTTP client, crawler, and scheduling logic. The crawler discovers endpoints while the HTTP client performs request dispatching and response collection.
(2) **Detection and Policy Layer:** Applies vulnerability-specific heuristics and risk scoring policies. The policy engine assigns severity and CI thresholds based on configured risk budgets and produces a single exit code that encodes the overall risk state of the scan.
(3) **LLM Augmentation Module:** Uses a local model (via Ollama) to suggest safe variations of payloads and test cases. The module communicates through a narrow, structured interface: prompts wrap findings in a strict JSON schema, and responses are required to conform to a compact JSON format with fields such as summary, rationale, fix, and concrete test steps. Outputs are parsed and validated; non-JSON responses or malformed objects trigger retries with stricter instructions or are discarded. Generated payloads are then validated and filtered before integration to prevent unsafe or state-changing actions, and the LLM never issues raw HTTP requests directly. In this way, the module augments, but does not replace, the deterministic detection logic and keeps model behavior bounded and auditable.
(4) **Audit Logging and Artifacts:** Each scan produces append-only, hash-chained entries to guarantee result integrity. Artifacts include all payloads, responses, and verification scripts,

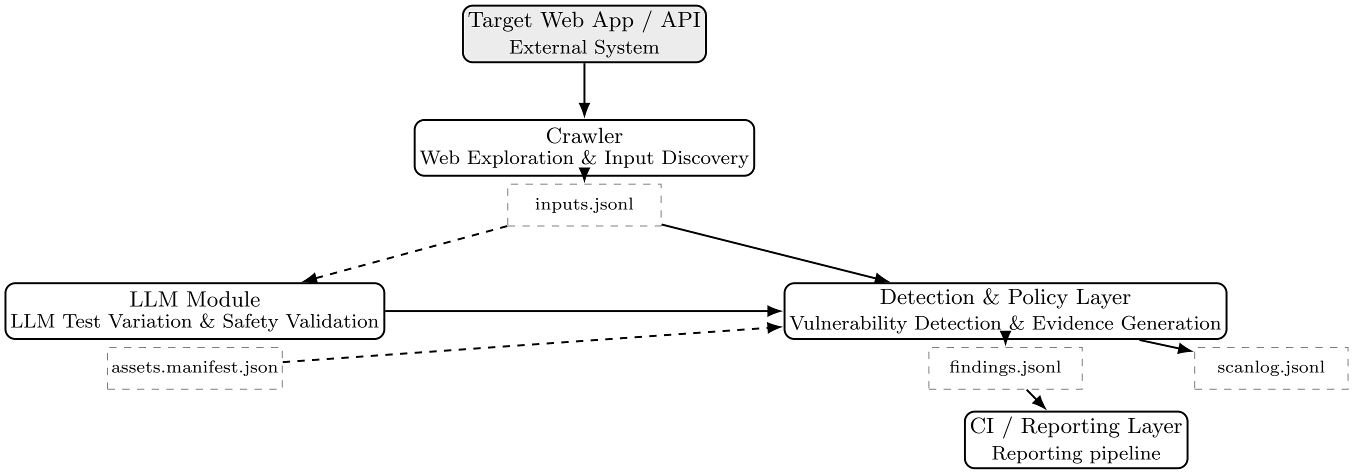

Figure 1: Overall Sentinel System Architecture

which allows scans to be replayed and independently verified. The hash chain provides a tamper-evident audit trail of the scan and is designed to align with reproducibility and accountability goals.

(5) **Reporting and CI Integration:** Results are transformed into structured JSON and rendered into HTML summaries. CI pipelines can use Sentinel's exit codes and scoring policy to automatically block or approve builds, and the per-finding artifacts and LLM-generated test suggestions help developers reproduce and validate issues locally.

The implementation follows a CMake-based modular layout, with independent test coverage for each subsystem using Catch2. Configuration files in config/ control crawler behavior, policy thresholds, payload definitions, and LLM-related settings such as model name and timeout. The Docker environment provides a reproducible runtime for local or CI-based testing and isolates the local LLM server from the rest of the system.

A block diagram (Figure 1) illustrates Sentinel's data flow:

- The Crawler discovers endpoints, forms, and inputs, and generates inputs for downstream analysis.
- The Detection Engine consumes the Crawler inputs, applies payloads, and generates findings along with verifiable scan logs.
- The LLM Module ingests both the Crawler inputs and the structure of the payloads, produces validated augmentations and proof-of-exploit suggestions, and updates payloads with safe test variations.
- All outputs flow into the CI and Reporting Layer, where results are compiled into reproducible artifacts and developer-facing reports.

This layered design ensures modularity, safety, and scalability. Each subsystem can evolve independently while maintaining a consistent interface for reproducible and verifiable DAST results, and the LLM integration remains constrained enough that its behavior can be tested, audited, and reasoned about like any other component.

4 Analysis

The development and evaluation of Sentinel provided practical lessons for building a reliable, reproducible, AI-assisted DAST framework. Early in the project, we focused on controlled experiments against known vulnerable web applications and validated each subsystem (crawler, detection engine, LLM layer, reporting) in isolation before integrating them. This made it easier to pinpoint where errors came from, improved detection accuracy and runtime behavior, and reduced the number of failures that only appeared in full end-to-end runs. In practice, frequent, small tests on individual modules turned out to be more effective than occasional large integration tests.

Data security was a focus of Sentinel's design. Sentinel regularly encounters sensitive information such as cookies, tokens, and personal data, so we designed the system to collect only what is necessary and store evidence in structured, tamper-evident logs instead of plain outputs. The hash-chained logs provide an auditable trail of what was observed and when, without forcing us to retain full payloads everywhere. The risk-budget and exit-code model then gives CI pipelines a clear, explainable signal about the level of risk in a build instead of a simple pass or fail. At the same time, the LLM components are deliberately constrained: prompts enforce a strict JSON schema, responses are parsed and validated, and outputs that do not match the expected format are retried under stricter instructions or ignored. As a result, LLM-guided tests serve as an additional source of ideas on top of deterministic checks rather than a replacement for them.

The work also revealed current limitations. Our evaluation focused on relatively small applications and a particular set of vulnerabilities, such as security headers, cookies, CORS behavior, and a small number of injection patterns. Highly stateful single-page applications, complex authentication flows, and large distributed systems remain more difficult to cover. We also found that verifying automatically generated payloads and avoiding tight coupling between modules requires deliberate design effort.

There are many possible expansions for this project moving forward. The addition of broader framework and protocol support, more detailed confidence scoring, stronger self-checks for LLM-generated tests (for example, self-consistency or adversarial prompts), and deeper CI/CD integration would allow for a significantly more robust tool. Throughout, the guiding principle is to keep Sentinel modular, conservative by default, and transparent enough that security and engineering teams can understand, rather than simply trust, the decisions it makes.

5 Legal Considerations

The use of automated web vulnerability scanning tools such as Sentinel raises several legal considerations. These considerations primarily relate to authorization, privacy, intellectual property, and liability. One such issues is that testing live web applications without explicit consent could violate the Computer Fraud and Abuse Act (CFAA) [1] in the United States, as well as similar international laws. Even unintentional system scanning could be interpreted as unauthorized access. Therefore, users of Sentinel must ensure that all tests are conducted with proper authorization and care to avoid legal repercussions.

Additionally, Sentinel could expose or process sensitive data during its scan. Privacy laws such as the General Data Protection Regulation (GDPR) [2] in the European Union have strict requirements for the handling of personal data, even when collected unintentionally. As such, the final build of Sentinel must collect only necessary data, anonymize any personal data, and securely store collected data to ensure compliance. Any findings or logs that contain personally identifiable information should be treated as protected data.

Because Sentinel uses a locally deployed LLM via Ollama, intellectual property and licensing considerations are also a concern. Outputs generated by the LLM—such as proof-of-exploit scripts or suggested remediation-could contain code fragments whose licensing status is uncertain. To mitigate potential legal risks, operators must review the Llama 3 Community License Agreement [9], and ensure that any outputs intended for distribution or reuse comply with copyright and open-source obligations.

Finally, AI-driven tools introduce potential liability. Ji et al. [7] note that AI-generated code can produce unexpected errors or security flaws. If Sentinel were to generate inaccurate findings or suggest unsafe mitigations that cause operational disruptions, determining responsibility becomes complex. Clear documentation, disclaimers, and mandatory human review steps are crucial to mitigate legal exposure. This legal exposure underscores the need to document how Sentinel uses AI components, to ensure lawful, transparent, and responsible deployment within cybersecurity operations.

6 Ethical Considerations

Several key principles of the ACM Code of Ethics and Professional Conduct [3] apply directly to the use of an LLM-enhanced DAST like Sentinel.

Principle 1.2 of the ACM Code of Ethics, "Avoid Harm", emphasizes that computing professionals must work to limit potential damages resulting from their work. Using Sentinel improperly could potentially lead to system disruptions, data leaks, or even reputational damage if scans are conducted without authorization or oversight. In accordance with this principle, Sentinel should include safeguards such as scope validation and rate limiting to prevent unintended harm to systems or users.

Principle 1.6, "Respect Privacy" is relevant as well. Vulnerability testing could uncover sensitive user information, including credentials, tokens, or personal data. Therefore, data must be handled confidentially, never be stored unnecessarily, and immediately be sanitized from reports. In order to respect privacy, Sentinel's capabilities should only serve the security of users and not expose or exploit their data.

In Principle 2.5, "Give Comprehensive and Thorough Evaluations", the ACM Code calls for honesty and accuracy in professional reports. Because LLMs could produce false positives or hallucinated vulnerabilities, ethical responsibility demands transparency about the system's limitations. Reports generated by Sentinel should clearly indicate the role of AI in analysis, identify confidence levels, and require human verification before conclusions are drawn.

Finally, Principle 3.1, "Ensure that the Public Good is the Central Concern" emphasizes the broader societal implications of cybersecurity research. While tools like Sentinel can strengthen defenses, they could also be misused by malicious actors. Responsible use means putting access controls in place, monitoring how Sentinel is used, and handling disclosures properly to make sure its benefits for security research outweigh its potential risks.

More broadly, the integration of AI into vulnerability testing introduces new ethical challenges related to bias, explainability, and accountability. Developers must ensure that humans remain ultimately responsible for all decisions made using AI-generated results. By aligning with the ACM Code's principles, Sentinel can promote ethical, transparent, and socially beneficial innovation in automated security testing

In addition to the ACM Code, Sentinel's design aligns with other professional and AI ethics frameworks. The IEEE Code of Ethics [11] and ISACA Code of Professional Ethics [6] emphasize integrity, due diligence, and public safety, reinforcing the need for responsible use. Data privacy principles, such as data minimization, consent, and auditability, require that Sentinel handle sensitive findings carefully, anonymize personal data, and maintain comprehensive logs. AI-specific ethics guidelines, including the EU Ethics Guidelines for Trustworthy AI [12], highlight the importance of fairness, transparency, and human-in-the-loop oversight, ensuring that humans remain ultimately accountable for decisions made using AI-generated results.

7 Conclusions

The development of Sentinel shows that it is possible to address some long-standing problems in dynamic application security testing (DAST) in a practical and developer-friendly way. Many existing tools flood teams with warnings and potential issues, without making it clear which ones are real vulnerabilities. Sentinel takes a different approach by tying each confirmed issue to a reproducible proof-of-exploit, a structured finding, and concrete remediation guidance. The LLM is used in a controlled, sandboxed role to suggest additional tests, while all final decisions are made by deterministic checks and policy code. Together, these choices help developers understand why a finding matters, how to reproduce it, and what to do next, while keeping the overall system predictable and auditable.

At the same time, the project makes clear that AI assistance in security testing needs to be treated with care. Sentinel deliberately avoids using the LLM as a standalone vulnerability detector, which reflects recent concerns about the reliability and stability of machine learning based security tools [14]. Instead, the model is constrained to a narrow, inspectable interface and its outputs are always subject to further validation. Hash-chained logs, minimal data collection, and a clear risk-budget model reinforce this emphasis on transparency and accountability [3]. Taken together, these design decisions suggest a path forward for AI-assisted security tooling that is cautious but still useful: automation that can be inspected, reproduced, and questioned, rather than opaque predictions that must be taken on trust.

References

[1] 1986. Computer Fraud and Abuse Act. https://uscode.house.gov/view.xhtml?req=(title:18%20section:1030%20edition:prelim)

[2] 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj

[3] Association for Computing Machinery. 2018. ACM Code of Ethics and Professional Conduct. ACM, New York. https://www.acm.org/code-of-ethics.

[4] Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. 2025. SoK: automated vulnerability repair: methods, tools, and assessments. In *Proceedings of the 34th USENIX Conference on Security Symposium.* USENIX Association, USA, Article 228, 20 pages.

[5] Xinyou Huang, Lei Zhang, Yongheng Liu, Peng Deng, Yinzhi Cao, Yuan Zhang, and Min Yang. 2025. Towards automatic detection and exploitation of java web application vulnerabilities via concolic execution guided by cross-thread object manipulation. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 429, 18 pages.

[6] ISACA. n.d.. ISACA Code of Professional Ethics. https://www.isaca.org/code-of-professional-ethics.

[7] Jessica Ji, Jenny Jun, Maggie Wu, and Rebecca Gelles. 2024. Cybersecurity Risks of AI-Generated Code. *Center for Security and Emerging Technology* (2024).

[8] Shangqing Liu, Wei Ma, Jian Wang, Xiaofei Xie, Ruitao Feng, and Yang Liu. 2024. Enhancing Code Vulnerability Detection via Vulnerability-Preserving Data Augmentation. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Copenhagen, Denmark) *(LCTES 2024)*. Association for Computing Machinery, New York, NY, USA, 166–177. doi:10.1145/3652032.3657564

[9] Inc. Meta Platforms. 2024. Llama 3 :latest (Model) – Ollama library blob 4fa551d4f938. https://ollama.com/library/llama3:latest/blobs/4fa551d4f938.

[10] MITRE Corporation. 2024. 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. Accessed: 2025-09-18.

[11] Institute of Electrical and Electronics Engineers (IEEE). n.d.. IEEE Code of Ethics. https://ieee-cas.org/about/ieee-code-ethics. Accessed: 2025-11-13.

[12] European Commission — High-Level Expert Group on Artificial Intelligence. 2019. Ethics Guidelines for Trustworthy AI. https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai.

[13] OWASP Foundation. 2021. OWASP Top 10:2021. https://owasp.org/Top10/ Accessed: 2025-09-18.

[14] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 238, 18 pages.

[15] Ravinder Singh, Mukesh Kumar Gupta, Dipak Raghunath Patil, and Sarang Maruti Patil. 2024. Analysis of Web Application Vulnerabilities using Dynamic Application Security Testing. In *2024 IEEE 9th International Conference for Convergence in Technology (I2CT)*. IEEE, Piscataway, NJ, USA, 1–6. doi:10.1109/I2CT61223.2024.10543484

[16] Shangzhi Xu, Jialiang Dong, Weiting Cai, Juanru Li, Arash Shaghaghi, Nan Sun, and Siqi Ma. 2025. Enhancing Security in Third-Party Library Reuse - Comprehensive Detection of 1-day Vulnerability through Code Patch Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, USA, 17 pages. doi:10.14722/ndss.2025.240576