

Parallel Stateful Security Fuzzing Algorithm in Haskell

Jenny Pan (JP3956) and Jack Bhalla (JSB2302)

Algorithm Overview and System Model

Modern authorization systems must maintain strict invariants under rapid sequences of state modifications. Two prevalent failure modes involve access persisting after revocation and delegation granting excessive authority. To detect these vulnerabilities, we propose a parallel stateful security fuzzing algorithm in Haskell that will evaluate permission frameworks against explicit formal invariants. We will model the system under test as a state transition graph where nodes represent permission configurations and directed edges represent state transitions such as resource creation, permission grants, delegation transfers, revocations, and access attempts. The proposed algorithm will explore this state space by generating stateful command sequences, executing them against a target system, and comparing the resulting state with a reference specification. When an anomaly occurs, the engine will extract and minimize the sequence into a deterministic, reproducible execution trace. To maximize search efficiency within a bounded execution budget, the engine will parallelize independent exploration batches and use a game-theoretic framework to distribute computational resources across complementary search strategies.

Theoretical Foundations and Methodology:

Our proposed framework will extend classical property-based testing [1] from static inputs to stateful operational traces, verifying whether execution sequences preserve safety invariants over time. Schema-based passive fuzzing often overlooks deep logical defects in stateful authorization layers. To overcome this limitation, we will incorporate the active checking methodology introduced in RESTler [2] by interleaving authority checks, post-revocation requests, and delegation steps into generated command sequences. Exploration will begin with bounded random walks that strictly enforce structural preconditions, requiring resources to exist before access attempts occur. Uniform random generators often suffer from coverage starvation along deep state trajectories because unconstrained command distributions favor high-probability surface operations. Swarm Testing [3] demonstrates that restricting specific generator features forces exploration into under-tested structural regimes. Inspired by this principle, our search engine will instantiate a suite of specialized policies targeting distinct structural patterns, such as deep delegation chains, high-frequency revocation cycles, or rapid resource expansion.

To allocate execution budgets effectively, we will model policy selection as a finite zero-sum game between the fuzzer and an adversary representing distinct bug classes. Preliminary batch runs will establish a payoff matrix estimating the detection probability of each policy across specific fault scenarios. Using the multiplicative-weights update algorithm [4], the fuzzer will dynamically adjust policy weights to converge toward a randomized strategy mixture. Freund and Schapire [4] establish that multiplicative-weights updates achieve low regret and converge toward a minimax equilibrium, guaranteeing that our proposed allocation maximizes the minimum expected detection probability across all targeted bug scenarios. While prior stateful protocol fuzzers such as EvoFuzz [5] apply evolutionary game theory to guide low-level byte mutation and state transitions, our framework will apply game theory at a higher structural level, optimizing resource allocation among symbolic authorization strategies in a pure functional language.

Parallel Execution Engine and Implementation:

The implementation will leverage the pure functional paradigm and parallel runtime of Haskell to execute search policies concurrently without shared-state locking. Algebraic data types will represent commands, permission configurations, and reference states, enabling pure transition functions to handle state evaluation. This deterministic foundation will guarantee exact replayability for any violating trace discovered during testing. Prior to parallelization, we plan to optimize sequential execution by refining graph representations and reducing memory allocations during state traversal. Parallel execution will rely on Haskell evaluation strategies using the parallel strategies library, dividing the total execution budget into independent batch evaluation tasks across CPU cores.

Parallelizing stateful command traces requires balancing evaluation strictness, thread scheduling overhead, and memory management. We will parameterize batch size and evaluation force, combining lazy evaluation with explicit deep evaluation, to maintain optimal throughput. Experimental evaluation will compare the proposed game-theoretic policy mixture against uniform strategy allocation and single-policy baselines across identical total operation budgets, accounting for all operations performed during security checks. Scalability experiments will run identical command batches and random seeds across 1, 2, 4, and 8 physical cores to measure runtime, speedup, and peak memory usage. All trials will include the computational cost of payoff estimation, replay detected violations for validation, and verify complete behavioral consistency between sequential and parallel executions. Because fuzzing provides incomplete coverage, terminating a run without finding a violation will indicate budget exhaustion rather than a formal proof of safety.

Synthetic Data Generation and Benchmark Evaluation:

Evaluating the fuzzing engine requires controlled authorization environments with known structural properties and injected bugs. We plan to construct a synthetic benchmark generator that will produce test instances containing parameterized sets of users, resources, permission grants, and delegation structures. Each generated instance will define an initial state configuration, random seeds, and two implementation variants consisting of a correct reference specification and a flawed target implementation. Target implementations will contain representative authorization flaws, including stale permission caches following revocation and missing authorization checks during delegation. The reference specification will calculate exact effective permissions over transitive grant graphs. To avoid false positive reports, the specification will account for disjunctive permission pathways, ensuring that revoking one grant does not invalidate access supported by an independent valid grant.

To ensure unbiased evaluation, payoff matrix estimation will use scenario configurations distinct from those reserved for final performance benchmarking. Primary evaluation metrics will include time to first violation and detection probability across repeated randomized trials. Benchmark validity will be confirmed through exhaustive state exploration on small instances and known violation sequences, while benchmark fault labels will remain hidden from the fuzzing engine during evaluation.

References

[1] Claessen and Hughes. QuickCheck, A Lightweight Tool for Random Testing of Haskell Programs. ICFP, 2000.

[2] Atlidakis, Godefroid, and Polishchuk. Checking Security Properties of Cloud Services REST APIs. ICST, 2020.

[3] Groce et al. Swarm Testing. ISSTA, 2012.

[4] Freund and Schapire. Adaptive Game Playing Using Multiplicative Weights. Games and Economic Behavior, 1999.

[5] Wang, Yu, and Yang. EvoFuzz, Enhancing State Space Exploration and Seed Prioritization in Stateful Protocol Fuzzing Using Evolutionary Game Theory. ISC 2025, proceedings dated 2026.