🧪 Fuzzing Framework – Overview & Adoption

✅ Summary

Our fuzzing framework is a **complete, extensible solution** for validating Python applications through structured input mutation and stateful method testing. Originally developed to explore edge-case behavior in standalone functions, it has evolved into a robust platform for **object-oriented fuzzing**, integrated reporting, and automation-friendly workflows.

This framework is now ready for use across other languages and projects. Teams are encouraged to explore adaptations, and we’re available to support integration and expansion.

🔍 Features

* **Function and Class Method Detection**
  + Uses Python AST to extract standalone functions and object-oriented interfaces
* **Stateful Fuzzing with Class Instantiation**
  + Instantiates classes and fuzzes methods while tracking logic and input effects
* **Input Configuration Wizard**
  + CLI prompts allow per-method input typing (e.g. str, int, bytes, json)
* **Crash Logging and Corpus Management**
  + Captures crash inputs, stack traces, and anomalous behavior in structured logs
* **Edge Case Stress Testing**
  + Optional module runs empty strings, oversized payloads, and binary sequences
* **Visual Reporting**
  + HTML and pie chart summaries provide clear success/failure metrics
* **CLI Dashboard**
  + Unified interface for method discovery, configuration, fuzzing, and reporting
* **Auto-Bootstrap for Dependencies**
* Self-installs required Python packages for a seamless first-time setup

🚀 Platform Readiness

The framework is built in a modular fashion and is fully adaptable. The core workflow—Discover → Configure → Fuzz → Report—is language-agnostic in design.

Teams can explore:

* Adapting the same CLI structure for Java, Go, Rust, or Node fuzzing ecosystems
* Swapping Atheris for other engines like AFL++, libFuzzer, or Hypothesis
* Embedding fuzzing stages into CI/CD for service-hardening and DevSecOps
* Linking corpus files to automated replay or test triage tools

The design allows for rapid prototyping and scalable rollout, with clear separation of configuration, execution, and reporting logic.

📌 Available Components

| **Component** | **Description** |
| --- | --- |
| fuzzing\_cli.py | Main launcher for all modules |
| bootstrap.py | Auto-installs missing Python dependencies |
| detect\_methods.py | Extracts all classes and public methods |
| configure\_methods.py | Wizard for method-specific input config |
| fuzz\_runner.py | Executes Atheris fuzzing and logs crashes |
| manual\_edge\_tests.py | Runs stability checks with rare inputs |
| generate\_report.py | Summarizes and visualizes results |

🧭 Getting Started

1. **Place your source code** under an api/ directory
2. **Run fuzzing\_cli.py** and follow the menu prompts
3. **Configure inputs** per method using the CLI wizard
4. **Execute fuzz tests** and track anomalies
5. **Generate reports** for coverage insights and debugging

🤝 Team Support and Expansion

This framework is ready to be used across teams and services. We can help:

* Adapt the test harness to different languages or execution models
* Extend the configuration wizard with schema-based inputs
* Enhance reporting or integrate with analytics platforms
* Support onboarding for engineering teams adopting the framework

If you’re interested in extending fuzzing coverage to your service or project, reach out and we’ll collaborate to get you started.

Would you like a ZIP archive of all scripts or want help turning this into a versioned documentation space? I can also prepare this as an onboarding kit for your team.