Evaluierung der Ideen für das Pynguin-Semesterprojekt

Studierendengruppe 03

5. Juni 2025

Übersicht

Im Folgenden werden alle bisher diskutierten Ideen für das Pynguin-Semesterprojekt nach **Machbarkeit** (Bei fünf Wochen Aufwand, Studium nebenbei) und **Umhau-Effekt** (Innovationsgrad / Überzeugungspotenzial) bewertet. Anschließend folgt eine komprimierte Tabelle aller Varianten.

A) Minimaler Async-PoC in Pynguin

- **Kurzfassung:** Erweiterung von Pynguin, um aus einem einfachen `async def`-Modul automatisch per `asyncio.run()` einen Test zu erzeugen, der den asynchronen Pfad abdeckt.
- **Machbarkeit:** Hoch
  - Klare Teilaufgabe: AST-Erkennung `async def` + Template mit `asyncio.run`.
  - Test-Setup nur ein kleines Demo-Modul (z. B. eine `greet`-Funktion).
  - Kein Setup von Dritt-Bibliotheken außer `pytest-asyncio` oder simplen `asyncio.run()`.
  - Innerhalb von fünf Wochen realistisch, wenn wirklich nur **1–2 async-Funktionen** abgedeckt werden.
- **Umhau-Effekt:** Niedrig bis Mittel
  - \+ Zeigt, dass Pynguin asynchrone Pfade berücksichtigen kann (neuer Use-Case).
  - – Sehr eingeschränkter Scope: Nur ein PoC für eine einzelne Async-Funktion, ohne tiefere Muster (`asyncio.gather`, `async for` etc.).
  - In der Praxis ließe sich dieses Skript in einem halben Tag manuell nachbauen, daher wenig Forschungstiefe.

B) Evolutionäre Heuristik-Optimierung (Quality-Diversity / Novelty-Search)

- **Kurzfassung:** Anstatt der klassischen Fitness-Selektion (nur Coverage) wird in Pynguin ein kleines „Novelty-Archiv“ eingeführt, in dem in jeder Generation neben Top-Coverage-Individuen auch jene mit größter Diversität (z. B. anhand Coverage-Bit-Distanz) ausgewählt werden.
- **Machbarkeit:** Mittel bis Hoch
  - Änderung nur im Fitness-/Selektionsmodul von Pynguin (eine oder zwei Klassen).
  - Feature-Vektor und Distanzmetrik lassen sich auf vorhandene Coverage-Daten (z. B. `coverage.py`) aufsetzen.
  - Zeitplan (1 Woche Einarbeitung, 2 Wochen Implementierung, 1 Woche Tests/Evaluation, 1 Woche Report) passt in fünf Wochen.
  - Sehr realistisch, da keine externe Infrastruktur nötig.
- **Umhau-Effekt:** Mittel bis Hoch
  - Novelty-Search und Quality-Diversity gelten in anderen Domänen (Robotik, Spiel-KI) als innovativ; in der Python-Testgenerierung sind sie praktisch neu.
  - Messbare Ergebnisse: Zeigt, dass diversitätsorientierte Selektion tatsächlich mehr oder andere Codepfade findet.
  - Wissenschaftlicher Mehrwert: Neuer Selektionsansatz, quantifizierbar belegt.

C) Hybrid-Fuzzing mit *The Fuzzingbook* (Proof-of-Concept)

- **Kurzfassung:** Verwendung des *The Fuzzingbook*-Frameworks (Coverage-Guided Fuzzer in Python) kombiniert mit Pynguin-Seeds, um schwer erreichbare Branches zu finden.
- **Machbarkeit:** Mittel
  - *The Fuzzingbook* liefert bereits Coverage-Hook und Fuzzer-Grundgerüst in reinem Python, keine C-AFL-Infrastruktur nötig.
  - Prototyp:
    1. Demo-Funktion (`parse_json(s: str)`) mit einigen `if`/`elif`-Verzweigungen.
    2. *The Fuzzingbook*-Fuzzer erzeugt Mutationen auf `s`, deckt zusätzliche Zweige auf.
    3. Pynguin erzeugt initiale Seeds (z. B. `"{}"`, `"null"`,), *The Fuzzingbook* nutzt diese als Startpopulation.
  - Aufwandsschätzung:
    * Woche 1: *The Fuzzingbook* kennenlernen + Minimalfuzzer auf Demo-Modul.
    * Woche 2: Pynguin für Demo (`parse_json`) anpassen (AST-Hook, Seed-Erzeugung).

    - Woche 3: Hybrid-Pipeline (Pynguin → *Fuzzingbook*) implementieren, Mini-Tests (Coverage-Vergleich).
    - Woche 4: Zweites Modul (`calc_expr(expr: str)`), vergleichende Messläufe (Coverage Pynguin vs. Fuzzer vs. Hybrid).
    - Woche 5: Ergebnisse verdichten, Report, Präsentation.
  - Risiken:
    - *The Fuzzingbook*-Setup (Coverage-Hooks, Mutationsstrategie) kann verzögern.
    - Experimentelle Auswertung stark begrenzen (nur 2 Module, je 3 Runs).
  - Insgesamt ist es straff, aber machbar, wenn keine zu komplexen Funktionen gewählt werden und der Fokus auf rein stringbasierte Parameter gelegt wird.

- **Umhau-Effekt:** Hoch
  - Koppelt zwei sehr unterschiedliche Paradigmen (evolutionärer Testgenerator vs. Coverage-Guided Fuzzer).
  - Demonstriert mit Zahlen den Coverage-Zuwachs durch Hybrid-Ansatz.
  - In Python-Forschungskreisen kaum Hybrid-Fuzzing-Publikationen; Thema wirkt frisch und überzeugend.

D) Implementierung eines Papers von Lars Grunske (Testpriorisierung in Microservices)

- **Kurzfassung:** Übernahme einer aktuellen Grunskes-Publikation (z. B. Ärchitekturbasierte Testpriorisierung¨) und Integration der dort beschriebenen Heuristik in Pynguin-generierte Tests für ein kleines Demo-Microservice-Szenario.

- **Machbarkeit:** Mittel
  - Benötigt:
    1. Aufbau eines kleinen Python-Microservice-Szenarios (2–3 Flask/FastAPI-Services).
    2. Anpassung von Pynguin, damit es Testfälle erzeugt, die HTTP-Requests an diese Services senden.
    3. Implementierung der Priorisierungsheuristik aus dem Paper (z. B. auf Basis eines Service-Call-Graph).
  - Zeitplan:
    - Woche 1: Paper studieren, Demo-Services skizzieren, Pynguin für HTTP-Calls vorbereiten.
    - Woche 2–3: Pynguin-Erweiterung für HTTP-Abfragen + Heuristik-Implementierung.
    - Woche 4: Evaluation (T̈ime-to-First-Failureünpriorisiert vs. priorisiert), mind. 3 Läufe.
    - Woche 5: Report Präsentation.
  - Risiko:

    - Microservice-Setup (Debugging, Konfiguration) kann unerwartet Zeit kosten.
    - Wenn Docker/Kubernetes o. Ä. genutzt wird, sprengt das den Zeitrahmen. Empfehlung: Rein lokale Flask/FastAPI-Instanzen.

- **Umhau-Effekt:** Mittel
  - \+ Direkter Anschluss an Forschung einer lokalen Gruppe (HU Berlin), zeigt Hochschulbezug.
  - − In einem kleinen Demo-Szenario bleibt die Demonstration begrenzt, und die Priorisierungsheuristik selbst ist oft nur ein Teilaspekt.
  - Wirkt solide, aber weniger „wow“, wenn erkennbar ist, dass nur ein Mini-Nachbau statt realer Microservice-Stack gezeigt wird.

E) Differential Evolution (DE) oder Particle Swarm Optimization (PSO) statt GA

- **Kurzfassung:** Ersetzung des Standard-Genetischen Algorithmus in Pynguin durch eine Mini-DE- oder PSO-Implementierung auf Vektor-Basis, mit dem Ziel, bei niedrigdimensionalen Signaturen schneller höhere Coverage zu erzielen.
- **Machbarkeit:** Mittel
  - Notwendige Schritte:
    1. Stellen in Pynguin finden, an denen die GA-Population initialisiert, mutiert und selektiert wird.
    2. Implementierung eines einfachen DE-Kernels (PopSize 20, F=0,8, Cr=0,9) in Python.
    3. Vektor → AST-Testfall-Wrapper (z. B. $[Funktionsindex, StringIndex, IntValue]$).
    4. Vergleichs-Evaluation auf 2–3 sehr kleinen Demo-Funktionen (je 3 Läufe, Coverage-Vergleich).
  - Zeitplan:
    - Woche 1: GA-Code verstehen, DE-Prototyp für ein algebraisches Mini-Problem implementieren.
    - Woche 2: Vektor-Mapping Integration in Pynguin.
    - Woche 3: Tests mit einfachem Modul `foo(x: int)`.
    - Woche 4: Zweites Modul `bar(s: str)` + Messläufe.
    - Woche 5: Report, Diagramme (Coverage vs. Generationen), Präsentation.
  - Risiko:
    - Mapping Vektor → gültiger AST-Aufruf kann tricky sein, wenn Funktionen mehrparametrig sind. Beschränkung: max. 2 Parameter.
- **Umhau-Effekt:** Mittel
  - DE/PSO in Testfallgenerierung ist im Python-Kontext ungewöhnlich und liefert klare Erkenntnisse („DE benötigt 30 Generationen statt 50 für 100 % Coverage“).

  - Aber es bleibt ein Vergleich von Varianten eines evolutionären Algorithmus – nicht so spektakulär wie eine völlig neue Paradigmenkombination (z. B. Fuzzing + GA).

F) Gruppenidee 1: Hybrid-Fuzzing + Pynguin (Atheris-basiert)

- **Kurzfassung:**
  1. Fuzzing (z. B. mit Google's Atheris) auf ausgewählte Funktionen ausführen.
  2. „Interessante" Inputs sammeln und als Grundtests (Seeds) konvertieren.
  3. Diese Seeds als Initialpopulation in Pynguin einspeisen.
  4. Pynguin evolviert die Seed-Tests, um Coverage zu maximieren.
  5. Evaluierung: Coverage-Verbesserung, Input-Diversität, Fault-Detection, Effizienz.
- **Machbarkeit:** Niedrig bis Mittel
  - Atheris ist existierender Python-Fuzzer (C-Extensions), Learning Curve steil (erfordert `libclang`).
  - Seed-Extraktion und automatisches Wrapping der Fuzzer-Inputs in Pynguin-Tests komplex:
    - Atheris-Setup (Clang/LLVM) korrekt installieren.
    - Hook in Atheris schreiben, um bei Coverage-Neuzuwachs Inputs zu protokollieren.
    - Rohe Bytestrings → String/Int/Objekt-Umwandlung, sonst stürzt Pynguin ab.
  - In fünf Wochen mit begrenzten Ressourcen sehr risikoreich.
- **Umhau-Effekt:** Sehr Hoch (wenn erfolgreich)
  - Eine echte „AFL-style" Fuzzer-Integration in Pynguin hätte großen Neuigkeitswert.
  - Falls nach fünf Wochen nur halbfertige Experimente oder instabile Builds existieren, wirkt das Ergebnis schnell unvollständig.
- **Empfehlung:** Nur wählen, wenn bereits solide Atheris-Erfahrung besteht und Teile der Infrastruktur bereits vorhanden sind. Ansonsten sollte man wegen des hohen Risikos eher zur The Fuzzingbook-Variante (C) greifen.

G) Gruppenidee 2: Automatisches Refactoring von Pynguin-Tests

- **Kurzfassung:**
  1. Pynguin generiert unleserliche, stark verschachtelte Tests.

2. Nutzung von `Black` oder `autopep8` bzw. selbstgeschriebenes Modul, um diese Tests aufzupolieren (Einrückungen, aussagekräftigere Variablennamen, Kommentare).
3. Vergleich der Coverage vor/nach Refactoring (Coverage muss gleichbleiben).

- **Machbarkeit:** Sehr Hoch
  - Keine tiefen Eingriffe in Pynguin-Interna nötig.
  - Linter/Formatter (Black, autopep8) laufen „out-of-the-Box".
  - Minimale Evaluation, dass Coverage stabil bleibt (z. B. drei Läufe).
- **Umhau-Effekt:** Niedrig
  - Zwar hilfreich für Developer-Akzeptanz, aber kaum Forschungstiefe.
  - Ein Professor erwartet hier höchstens „OK, formatiert, Coverage bleibt", aber keine herausragende wissenschaftliche Herausforderung.
- **Empfehlung:** Nur als Zusatz-Teilaufgabe („Falls Zeit übrig bleibt, polieren wir die Tests"). Nicht als Hauptprojekt.

H) Gruppenidee 3: Priorisierung/Heuristik-Vorverarbeitung

- **Kurzfassung:**
  1. Untersuchung, wie Pynguin aktuell Testziele auswählt (Methoden, Klassen).
  2. Implementierung eines einfachen Heuristik-Preprozessors, der Methoden nach Komplexität (Anzahl `if`, `for`, `while` etc.) oder vermuteter Aufrufhäufigkeit (Call-Graph) rankt.
  3. Übergabe dieser Rangfolge an Pynguin (z. B. über `-target-methods`), sodass zuerst „schwierige"/komplexe Methoden getestet werden.
  4. Messung, ob Coverage-Konvergenz (Zeit bis zu $X\%$ Coverage) schneller ist als ohne Heuristik.
- **Machbarkeit:** Hoch
  - Pynguin bietet bereits Schnittstellen (`-target-methods`, etc.), sodass nur ein Skript für Komplexitätsanalyse geschrieben werden muss.
  - Evaluierung: 2–3 Python-Repos, *Time to 50% Coverage* mit Default vs. Heuristik (je 5 Läufe à 1 Minute).
  - In fünf Wochen gut zu schaffen (geringer Implementierungsaufwand).
- **Umhau-Effekt:** Mittel
  - Echte Verbesserung der Effizienz in der Testgenerierung.
  - Weniger spektakulär als Hybrid-Fuzzing oder Novelty-Search, aber solide und praktikabel.
  - Wenn nachgewiesen wird, dass Komplexitäts-Priorisierung Pynguin 15–20 % schneller zur gleichen Coverage bringt, wirkt das überzeugend.

Zusammenfassende Tabelle aller Varianten

Idee	**Machbarkeit in 5 Wochen**	**Umhau-Effekt**	**Kurze Begründung**
A) Minimaler Async-PoC	Hoch	Niedrig–Mittel	Schnell realisierbar, aber sehr eingeschränkter Use-Case (nur 1–2 async-Funktionen).
B) QD / Novelty-Selektion	Mittel–Hoch	Mittel–Hoch	Neuer Selektionsansatz („Diversitäts-Suchraum“) mit messbarem Mehrwert; bleibt in Pynguin-Code.
C) Hybrid-Fuzzing mit *The Fuzzingbook*	Mittel	Hoch	Innovativ, weil Fuzzing + GA kombiniert; rein in Python umsetzbar; kann in 5 Wochen realisiert werden, aber Fuzzer-Setup benötigt Sorgfalt.
D) Implementation eines Grunskes-Papers	Mittel	Mittel	Guter Lokal-Research-Bezug, erfordert aber Mini-Microservice-Setup; zeigt Praxisnutzen (Time-to-First-Failure).
E) DE/PSO statt GA	Mittel	Mittel	Vergleich von GA vs. DE/PSO auf niedrigen Dimensionen; Vektor-Mapping nötig, aber überschaubar.
F) Hybrid-Fuzzing + Atheris	Niedrig–Mittel	Sehr Hoch	Extrem innovativ, aber Atheris-Infrastruktur (Clang/LLVM, Coverage-Hooks) in 5 Wochen riskant.
G) Automatisches Refactoring	Sehr Hoch	Niedrig	Leicht realisierbar (Black/autopep8), aber kaum Forschungsinhalt (Coverage bleibt gleich).
H) Priorisierung / Heuristik	Hoch	Mittel	Effiziente Priorisierung (Komplexitätsanalyse → schneller Coverage-Konvergenz); vergleichsweise einfacher Code, solide Ergebnisse.

Empfehlung und Fazit

- **Primärer Vorschlag: Hybrid-Fuzzing mit *The Fuzzingbook* (C)**
  - Höchster Innovationsgrad, echte Kombination von zwei Paradigmen (evolutionärer Testgenerator + Coverage-Guided Fuzzer).
  - In fünf Wochen machbar, wenn Fokus auf 2 kleine Demo-Funktionen gelegt wird und experimentelle Runs (je 3 Läufe) begrenzt werden.
  - Zeigt in Python-Forschungskreisen, wie Fuzzing + GA zusammenwirken und liefert konkrete Coverage-Zuwächse.
- **Backup: Quality-Diversity / Novelty-Search (B)**
  - Ebenfalls forschungslastig: Neuer Selektionsansatz, der Diversität in die Population einführt.
  - In fünf Wochen umsetzbar (Fitness/Selektionsmodul ändern, Distanzmetrik definieren, Evaluation in 1 Woche).
  - Ergebnis: Quantifizierbare Verbesserung der Coverage oder Entdeckung neuer Pfade.
- **Alternative: Heuristik-Priorisierung (H)**
  - Erfordert nur ein Skript zur Komplexitätsanalyse und Nutzung der bestehenden Pynguin-Schnittstelle (`-target-methods`).
  - Sehr sichere Umsetzung in fünf Wochen, liefert messbare Effizienzgewinne (z. B. 15–20 % schnellere Coverage).
  - Weniger innovativ als Ansätze B oder C, aber sauber realisierbar und narrensicher.
- **Weniger empfehlenswert als Hauptprojekt:**
  - Async-PoC (A) – zu trivial, nur Proof-of-Concept für wenige Async-Funktionen.
  - Hybrid-Fuzzing + Atheris (F) – zu riskant, hohe Infrastruktur-Hürden in 5 Wochen.
  - Automatisches Refactoring (G) – kaum Forschungstiefe.