This research paper introduces **ACH (Automated Compliance Hardener)**, a system developed at Meta for automated unit test generation, focusing on identifying and preventing regressions related to specific issues like privacy. Here's a breakdown of the key concepts and ideas:

## Core Idea: Mutation-Guided LLM-Based Test Generation

Instead of traditional methods, ACH uses a novel approach:

* **Targeted Mutation Testing:** ACH strategically introduces simulated faults ("mutants") specifically related to a chosen concern (e.g., privacy violations) into the code. These aren't random mutations; they're designed to mimic real-world bugs that could lead to the targeted issue.
* **LLM-Powered Test Generation:** ACH then uses a Large Language Model (LLM) to automatically generate new unit tests that can detect (or "kill") these mutants. The goal is to create tests that expose the potential vulnerabilities represented by the mutants.

In essence, ACH uses LLMs to create "super bugs" (mutants) and then generates tests to catch them.

## Key Aspects of ACH

* **Focus on Specific Concerns:** ACH isn't a general-purpose test generator. It's designed to harden code against particular issues, such as privacy, security, or integrity.
* **Agentic LLM Workflow:** ACH employs multiple LLM-based "agents" that work together to generate faults and tests.
* **Mutation Testing Importance:** The paper emphasizes that mutation testing is superior to structural coverage criteria (like line coverage) because it can find faults even when the same lines of code are executed in a new way.
* **Wide Applicability:** While the paper focuses on privacy, the authors believe the ACH approach can be applied to various software testing problems.
* **Addressing a Fundamental Question:** ACH tries to bridge the gap between vague textual descriptions of software concerns and concrete unit tests that prevent bugs related to those concerns.
* **Risk Assessment:** ACH can use the distribution of simulated faults to estimate the risk exposure of different system components to the issue under test.
* **LLM Model:** The single language model Llama 3.1 70Bn was used in all the agents reported on.

## ACH System Architecture

The ACH system consists of several components working in a pipeline:

1. **Issue Summary:** Takes textual descriptions of concerns (e.g., previous privacy faults, regulatory requirements) as input.
2. **Fault Generation:** An LLM-based agent generates simulated faults (mutants) in the code, guided by the issue summary, existing tests, and the code under test.
3. **Equivalence Detection:** An LLM-based agent identifies and removes mutants that are functionally equivalent to the original code (i.e., they don't actually change the behavior).
4. **Test Generation:** An LLM-based agent generates new unit tests that are designed to "kill" the non-equivalent mutants.
5. **Testing and Validation:** The generated tests are built and executed against both the original code and the mutated code. Tests that pass on the original code but fail on the mutated code are considered valuable.
6. **Code Review:** Engineers review the generated tests and assess their relevance and quality.

## Key Contributions

* **Empirical Results:** The paper presents results from applying ACH to real-world software platforms at Meta, including the number of mutants generated, tests created, and the impact on code coverage.
* **Deployment Experience:** The paper shares insights from deploying and evaluating ACH at Meta, including engineer acceptance rates and feedback on the relevance of the generated tests.
* **Equivalent Mutant Detection:** The paper evaluates the performance of ACH's equivalent mutant detection agent.
* **Lessons Learned:** The paper discusses the challenges and open research questions that arose from this industrial application of LLM-based test generation.

## Prompts

The paper uses three simple prompts:

* **Make a fault:** Generates a mutated version of the class that contains a typical bug that introduces a privacy violation similar to a given diff.
* **Equivalence detector:** Determines if two versions of a Kotlin class will always do the same thing.
* **Make a test to catch fault:** Writes an extended version of the test class that contains extra test cases that will fail on the mutant version of the class but would pass on the correct version.

## Implications and Future Directions

This research suggests that mutation-guided LLM-based test generation is a promising approach for hardening software against specific concerns. The authors highlight the potential for further research in areas such as:

* More sophisticated prompting techniques for LLMs.
* Using language model ensembles to improve test generation.
* Applying the approach to a wider range of software testing problems.

## ACH Example: Data Integrity

**Concern:** Critical data fields (e.g., customer IDs) in a system must never be set to None or empty strings after initial creation.

1. **Issue Summary (Input):**
   * Textual Description: "Ensure that once a customer ID is assigned, it cannot be set to null or an empty string. This is crucial for data integrity and system functionality."
2. **Fault Generation:**
   * Imagine a Customer class with a customer\_id attribute. The mutation will try to introduce a bug where the customer\_id can be inadvertently set to None or "".
   * **Hypothetical Code:**
     + Original Code (Python):

python

class Customer:

def \_\_init\_\_(self, customer\_id, name):

if not customer\_id:

raise ValueError("Customer ID cannot be empty")

self.customer\_id = customer\_id

self.name = name

def update\_name(self, new\_name):

self.name = new\_name

def set\_customer\_id(self, new\_id):

if not new\_id:

raise ValueError("Customer ID cannot be empty")

self.customer\_id = new\_id

* + - Mutated Code (Fault Introduced - allowing null ID):

python

class Customer:

def \_\_init\_\_(self, customer\_id, name):

if not customer\_id:

raise ValueError("Customer ID cannot be empty")

self.customer\_id = customer\_id

self.name = name

def update\_name(self, new\_name):

self.name = new\_name

def set\_customer\_id(self, new\_id):

# MUTANT <START>

# Bug: Accidentally allows setting customer\_id to None or ""

self.customer\_id = new\_id # No check for empty ID!

# MUTANT <END>

1. **Equivalence Detection:**
   * The LLM equivalence detection agent would analyze the original and mutated code. If the agent determines that the change in set\_customer\_id doesn't actually lead to any observable difference in behavior (e.g., because other parts of the code prevent null IDs), it would mark the mutant as equivalent and discard it.
2. **Test Generation:**
   * The LLM test generation agent now focuses on creating tests that exploit the potential vulnerability introduced by the mutant in set\_customer\_id.
   * **Hypothetical Code:**
     + Existing Test (might not catch the mutant):

python

import unittest

class TestCustomer(unittest.TestCase):

def test\_customer\_creation(self):

customer = Customer("123", "Alice")

self.assertEqual(customer.customer\_id, "123")

self.assertEqual(customer.name, "Alice")

def test\_update\_name(self):

customer = Customer("123", "Alice")

customer.update\_name("Bob")

self.assertEqual(customer.name, "Bob")

* + - Generated Test (to kill the mutant):

python

import unittest

class TestCustomer(unittest.TestCase):

def test\_customer\_creation(self):

customer = Customer("123", "Alice")

self.assertEqual(customer.customer\_id, "123")

self.assertEqual(customer.name, "Alice")

def test\_update\_name(self):

customer = Customer("123", "Alice")

customer.update\_name("Bob")

self.assertEqual(customer.name, "Bob")

def test\_set\_customer\_id\_to\_none\_fails(self): # New test case

customer = Customer("123", "Alice")

with self.assertRaises(Exception): # this one doesnt raise an exception

customer.set\_customer\_id(None)

def test\_set\_customer\_id\_to\_empty\_string\_fails(self): # New test case

customer = Customer("123", "Alice")

customer.set\_customer\_id("") # This will run fine on original, but break on mutant

self.assertNotEqual(customer.customer\_id, "", "Customer ID should not be empty")

1. **Testing and Validation:**
   * When the test suite is run:
     + The original code will pass all tests.
     + The mutated code will fail the test\_set\_customer\_id\_to\_empty\_string\_fails test because the set\_customer\_id method no longer prevents setting the ID to an empty string. This "kills" the mutant.
2. **Code Review:**
   * A developer reviews the newly generated tests. They assess if the tests are relevant, well-written, and truly address the data integrity concern. If the developer approves, the tests are integrated into the codebase.

## Key Improvements in This Example

* **Clearer Data Integrity Focus:** This example is more directly tied to a data integrity concern.
* assertRaises**Context Manager:** The use of assertRaises demonstrates how to write tests that verify that specific exceptions are raised when invalid data is encountered.
* **Emphasis on**assertNotEqual**:** The final assertion verifies that the customer ID is not an empty string after the set\_customer\_id method is called, directly targeting the injected fault.

**Our current methodologies for testing in wells fargo**

**SonarQube (Static Analysis):**

\* Doesn't automatically generate unit tests. It provides feedback on code quality and potential issues, but developers still need to write tests manually.

**Why ACH is Better in this Context:** ACH actively creates tests to address specific concerns, reducing the manual effort required to write targeted unit tests. From which testers can select the required test cases

**Black Duck (SCA Tools):**

* **Focus:** Identifying known vulnerabilities in third-party libraries and components used in your project (e.g., in JAR files). It uses a database of known vulnerabilities (like those in the National Vulnerability Database - NVD) to scan your dependencies and flag any that have known security flaws.
* **How it Works:** SCA tools analyze your project's dependencies to create a Bill of Materials (BOM). They then compare this BOM against vulnerability databases to identify any vulnerable components.
* **Strengths:** Excellent for managing open-source risk, ensuring you're not using libraries with known vulnerabilities, and providing information on licensing and legal compliance.

**ACH (Automated Compliance Hardener):**

* **Focus:** Hardening your own code against regressions that could lead to specific issues, such as privacy violations, data integrity problems, or security flaws. It generates tests to catch newly introduced bugs in your code. ACH creates mutants and the tests that can catch them.
* **How it Works:** ACH uses mutation-guided, LLM-based test generation. It injects simulated faults (mutants) into your codebase, then uses an LLM to generate new unit tests that can detect these faults.
* **Strengths:** Proactive approach to preventing regressions related to specific concerns.

**Key Differences and Why ACH is Better in**Specific**Scenarios:**

1. **Scope:**
   * **Black Duck:** Focuses on external dependencies.
   * **ACH:** Focuses on your own code.
2. **Type of Vulnerabilities Addressed:**
   * **Black Duck:** Finds known vulnerabilities in third-party libraries.
   * **ACH:** Helps prevent new vulnerabilities (regressions) in your own code.
3. **Approach:**
   * **Black Duck:** Detects existing vulnerabilities.
   * **ACH:** Prevents future vulnerabilities by hardening your code with targeted tests.

**Why ACH is Better (in**Specific**Scenarios):**

* **Addressing Zero-Day Vulnerabilities:** Black Duck relies on known vulnerabilities. ACH can help you create tests that might catch vulnerabilities in your code before they are publicly known (zero-day).
* **Custom Code:** Black Duck won't help you with vulnerabilities in the code you write yourself. ACH is specifically designed to test your own code for weaknesses.
* **Compliance with Specific Rules:** ACH can be tailored to specific compliance requirements (e.g., GDPR privacy rules). It helps you write tests that ensure your code adheres to these rules.

**Analogy:**

* **Black Duck is like a background check:** It verifies that the libraries you're using don't have a criminal record (known vulnerabilities).
* **ACH is like a security audit of your house:** It identifies potential weaknesses in your own systems and helps you reinforce them.