GitFL: Automated fault localization for environments where code-changes by multiple developers are tested simultaneously

J.E. van Dorth tot Medler
Student Number: 4453344
April 26th 2023

Master of Science Robotics
Delft University of Technology
Adyen N.V.

Robotics Master Thesis Project (RO57035)
Department of Mechanical, Maritime and Material Engineering
March 2022 - January 2023

Supervisors:
Prof. dr. ir. J.C.F. (Joost) de Winter (Delft University of Technology)
Kelvin Elsendoorn (Adyen N.V.)

Defense commitee:
Prof. Dr. Ir. J.C.F. (Joost) de Winter (Delft University of Technology)
Prof. Dr. A.E. (Andy) Zaidman (Delft University of Technology)
Dr. Ir. Y.B. (Yke Bauke) Eisma (Delft University of Technology)
Kelvin Elsendoorn (Adyen N.V.)

CONTENTS

GitFL: Automated fault localization for environments where code-changes by multiple developers are tested simultaneously

Jan van Dorth tot Medler
Delft University of Technology
Adyen N.V.
Delft, The Netherlands
j.e.vandorthtotmedler@student.tudelft.nl

***Abstract*—**
Background: For rigorous software testing, integration and end-to-end tests are essential to ensure the expected behavior of multiple interacting components of the system. When software is subjected to integration or end-to-end tests, it is often unfeasible to test every code change individually, as the runtime of these tests is usually significantly larger compared to unit tests. For this reason, batches of code changes from multiple authors are often tested simultaneously. **Problem:** An issue with testing multiple changes simultaneously is that it can be unclear which change form which author caused the failure when tests fail, as all changes from all authors included in the test can be at fault. **Design:** To solve this, a new automatic fault localization algorithm called GitFL is introduced, which combines state-of-the-art fault localization with version control history information for enhanced performance. GitFL was evaluated on a C++ repository at Adyen where tests are considered to be end-to-end. **Findings:** It showed that the addition of version control history information significantly increases the performance of fault localization for systems where multiple changes are tested simultaneously. **Societal implications:** This work provides insights on improved fault localization for these systems, which could enable organizations which develop these systems to speed up their testing and development processes. **Originality:** This work contributes by focusing on fault localization specifically for systems where multiple changes are tested simultaneously, which was not researched before.

***Index Terms*—software fault localization, software testing, EXAM score, software debugging, survey**

I. Introduction

Software testing is an essential element of building and maintaining software systems. It ensures that software meets its requirements and does not crash. Usually, three types of test can be distinguished, being unit-, integration-, and end- to-end tests. A single unit test tests a small part of code in isolation, for example by ensuring that a method returns the right output or by asserting that a variable attains the right value. Integration tests test the combination of multiple modules working together and end-to-end tests (or systems tests) test the software by simulating real-life use cases, requiring the entire system to operate and interact without failure [1].

End-to-end tests can attain many forms, depending on the type of system that is under tests. Systems that contain embedded systems through which user interaction is expected, cannot be fully automated end-to-end tested by using software-only testing solutions, as physical interaction is required. To still provide automated end-to-end testing for these scenarios while refraining from manual testing, robotic testing can be deployed. In the previous years, robots became more enhanced and easily accessible, partly by innovations in additive manufacturing (3D printing) [2]. These developments lowered the barriers of using robotics for automation within many applications within many different industries, including hardware testing. With these robots, user interaction with embedded systems can be simulated, enabling automated end-to-end testing. For example, for systems which require interaction through a touch-screen, a robot-arm can mimic a human arm to provide the required input.

Unit tests are usually executed by the developer (author) after making a change in the code. For integration tests and especially end-to-end tests, testing after every code change is unfeasible due to these tests often being slow. An entire system with all of its components takes more time to be built and run, compared to the often dependency-less unit-testing [3], which causes its testing to be more time-consuming. For example, for robotic end-to-end tests, pressing multiple buttons as input takes a significantly larger amount of time compared to almost instantly providing the button values in a software-only test. For these time-consuming scenarios, code changes of multiple authors are tested simultaneously.

When a batch of code changes from multiple authors is tested and tests fail, it can be unclear which change (and thus which author) caused the failure, as opposed to unit testing, where inherently only one author can be at fault. When it is unclear which author caused the failure, either more developers have to search for bugs or developers have to search outside of their own domain, increasing

the time required for debugging. Increased debugging time is expensive. Namely, when bugs (also referred to as faults) are not located early in the development cycle, locating them becomes more time-consuming [4]. Next to that, developers do not like debugging, especially when debugging takes a large amount of time [5].

A widely-used approach for shortening debugging-time is automatic fault localization (FL). Automatic FL uses coverage information (e.g. data indicating which statements are executed) combined with test outcomes as input [6, 7, 8, 9, 10] and outputs a ranked list of statements suspicious of containing bugs. This output, inspected by the developer, can contain many statements with a similar or even equal suspiciousness [11], making it difficult to localize the bug. Additionally, with low coverage test suites, which often tests more modules simultaneously, code is usually not tested finely-grained. As FL performs better with tests that test more fine-grained parts of the code, it performs worse for lower coverage test suites [12].

Version control has been essential for software development in the last decades and is used throughout most organizations [13]. These version control systems hold extensive data on when, how, and by who the code-base has been altered. When searching for bugs, every bit of information about the code-base history could be valuable. Thus, the data from version control systems could hold untapped potential when combined with automatic fault localization, especially for low coverage test suites, with the goal of narrowing down the number of statements for the developer to inspect. In this paper:

- We present a novel method to automatically localize faults using version control data for systems where batches of code changes are tested simultaneously,
- We evaluate the performance of this method on both artificial and real-life test suites.

II. Background

A. Fault localization

Automated FL is the practice of localizing software faults (e.g. bugs) automatically. Instead of requiring a developer to manually search for bugs, execution coverage and test results are combined and fed to an algorithm that predicts which statements are most likely to contain bugs. More precisely, the execution coverage resembles the specific statements that are executed during the tests. The test outcome indicates if a test has passed or failed. To illustrate the concept, suppose that a simple program is the subject of several tests, of which some passed and some failed. If a specific statement appears to be executed during every failed test and never during passed tests, logically, this statement has a high probability of containing the bug.

This concept is further explained by introducing a characteristic fault localization method named Tarantula [14], which belongs to the family of methods called Spectrum-Based Fault Localization (SBFL). This seminal paper, which introduced this type of method, is cited over 1400 times. The Tarantula formula is shown in figure 1 with its parameters in table I. For every statement (indicated with 'e'), the number of times it is executed during passed test and failed tests is used, as well as the total number of both passed and failed tests. This results in the suspiciousness number, ranging from zero to one, indicating the probability of the statement containing a bug.

TABLE I: Parameters in spectrum-based formulae [15]

e_f	Number of failed tests that execute the program element.
e_p	Number of passed tests that execute the program element.

$$suspiciousness(e) = \frac{\frac{e_f}{e_f+e_p}}{\frac{e_p}{e_f+e_p} + \frac{e_f}{e_f+e_p}} \quad (1)$$

Fig. 1: Formula of the Tarantula method [6] (edited).

Next to executed statements, many types of execution information can be used, such as paths (the order in which statements are executed) [16, 17], branches/predicates (statements that return either true or false, for example the condition that is checked in an if-statement) [9, 18], in-variants (program states that should hold during the entire life of the program) [19], program traces (logs about program execution), data-dependencies (when statements refer to data of preceding statements), outputs (values returned by the program) [20] and combinations of them [21].

SBFL can be extended with other information sources to obtain better performance. For example, the novel method Histrum-based Spectrum Fault Localization (HSFL) [22] combines spectrum-based fault localization (SBFL) with the version control history (VCH) of the code repository (the VCH contains all changes to the code with their timestamp and author), to increase FL performance. This method is based on the following principle: if a changed statement belongs to a set of changes which cause tests to fail, it is assumed that this statement should have an increased suspiciousness.

The equation belonging to HSFL is shown in equations 2 and 3. The suspiciousness (here called HSFL) is obtained by combining standard spectrum-based FL and the histrum. The histrum is calculated in the following way. First, it is determined which commits are so-called 'bug-inducing' (also called 'inducing'), e.g. commits before which tests passed and after which failed. Then, for each statement (s), it

is determined by how many $inducing$ and $non-inducing$ commits it is changed. Then, SBFL and the histrum are combined using a weighted combination with weighting factor α.

$$Histrum(s, c_i) = \frac{induce(s)}{\sqrt{N_I * (induce(s) + noninduce(s))}} \quad (2)$$

$$HSFL(s) = \begin{cases} (1-\alpha) * SBFL(s) & s \in A \wedge s \notin S_c \\ (1-\alpha) * SBFL(s) + \alpha * Histrum(s) & s \in A \wedge s \in S_c \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where:

α = weight factor determining the ratio of SBFL versus Histrum
$Histrum$ = the histrum

B. Version control

Version control (also known as revision control and source control) is used by developers to keep track of code changes. Every change that is recorded in the version history, can be retrieved. Therefore, developers gain the possibility to roll back to previous versions at any time. All changes of all files, together with the project structure are stored in the so-called repository. With version control, it is possible to branch, e.g. to split off at a certain point in the version history to for example work on a new feature that is not ready to be included in the main project yet [23]. Several existing version control tools are Git [24], CVS [25], and Bitkeeper [26].

III. Related works

The first FL methods were introduced in 1982, with slice-based methods [27]. Slice-based FL isolates all statements that could affect a specific variable chosen by the developer [28], limiting the number of statements a developer has to examine to find the bug. The developer could, for example, choose this variable to be a certain program output variable that returns an irregular value, isolating all statements of the program that can affect this output variable. These isolated statements grouped together are called a slice. This method, introduced by Weiser [27, 29], is called static slicing. It produces slices that contain all statements that could affect a variable, regardless of the program input and test outcome. Korel and Laski [30] introduce dynamic slicing, which uses slices that exist of only those statements that are executed for a specific input. The number of statements in a dynamic slice is always less than or equal to the size of the static slice, as the static slice includes the statements for all possible inputs.

Another slice-based method, called execution slicing, creates a slice that holds all executed statements of a program, disregarding input and test results and without the need for the developer to specify a specific variable to inspect [31]. Static- and dynamic slicing require the specification of a variable for which to create slices, in contrast to execution slices. All types of slicing are language-independent, as they do not depend on language-specific properties, but only on which lines of code are executed. Slicing does not leverage test outcomes and only isolate statements for inspection as opposed to providing a ranking of statements to inspect first. Because of this, it can be argued that this family of methods is not fully automatic.

Execution slices are often used as the input for a family of automated FL methods, called spectrum-based FL (SBFL). SBFL uses execution slices (e.g. execution information) of the program combined with test outcomes to localize bugs. One of the first introduced spectrum-based methods is Set Union [6]. The formula of Set Union is shown in figure 2. Set Union works as follows: the statements executed (E) by a single failed test (f) are considered. From those statements, the union of the statements executed by all passed tests (Ep for all p in P) is subtracted. The remaining statements are thus only executed by failed tests and never by passed tests. These remaining statements should be examined first by the developer. Next to Set Union, similar basic methods exist [6, 32].

$$E_{initial} = E_f - \bigcup_{p \in P} E_p \quad (4)$$

Fig. 2: Set union formula [6].

For more detailed results, the execution information of the program combined with test outcomes can also be combined with formulae that compute the suspiciousness of parts of the program (e.g. how likely is it for the statement to contain the bug). Most spectrum-based methods use execution information consisting of which statements were executed [33]. There exists many methods similar to Tarantula, all with varying formulae [8, 34, 7, 35]. Where execution slices are program-independent, other execution information, such as predicates, can be language-dependent, making data collection potentially a more complex effort. A widely acknowledged shortcoming of spectrum-based methods is the problem of ties. When multiple statements have the same suspiciousness score, they have the same rank. In other words, they are tied. It is then unclear which of these tied statements should be examined first [36].

Execution slices are also used as input for Machine Learning-based FL methods (MLFL). MLFL treats execution slices as features and test results (either passed or failed) as labels, with which a model can be trained, as shown by Zheng et al. [37]. There exist several methods,

all based on different machine learning models. BPNN [37] uses a (shallow) neural network, DNN [38] uses a deep neural network and CNN-FL [39] uses a convolutional neural network. Other methods focus on using clustering, such as RBF [40], which makes use of a radial basis function. LingXiao [41] also makes use of clustering, to find bug-related predicates.

Next to execution slices, other inputs for FL methods exists. One of those is code mutations, which entails the modification of statements. Mutation-based methods (MBFL) leverage the mutation of statements to localize bugs. MUSE, introduced by Moon et al. [42], is based on the concept of trying to make failed tests pass by mutating statements. For example, a mutant of the statement $var = x$ can be $var = -x$. If this mutant causes previously failing tests to pass, the suspiciousness of the original statement containing a bug will be high. Papadakis and Le Traon [43] introduce Metallaxis, which is similar to MUSE. However, Metallaxis works by observing if mutants change the program output compared to the original program version, instead of only observing if a mutant switches a test result from failed to passed.

Another input for another FL family is changed program states. State-based FL methods make use of observing and changing program states. Program states refer to the contents of memory locations at any time during the execution of the program. Relative Debugging [44] assumes that there are several known reference program states linked to correct program behavior. The program states of the version of the program that is being tested are then compared to the reference program states and when they do not match up, it should be investigated. Delta Debugging [45] replaces program states of failed tests with the corresponding program states of passed tests and reruns the test. If the test passes, these program states are no longer suspicious of holding the bug. Cause Transitions [46] aims to locate moments in time where variables end being failures and transition to another variable. Predicate Switching [47] switches the outcome of a predicate value during the execution of the program, trying to cause a successful program execution. If a switched predicate caused the program to succeed instead of fail, this could hint at the location of the bug. A disadvantage is that, for state-based methods to work properly, program states need to be able to be changed during runtime, which is not always possible, for example with secured embedded systems. The memory of these systems is either executable or writable, which means that executable code cannot be altered during runtime, inhibiting the change of program states.

Next to the families of methods listed above, more families of methods exists, such as statistical debugging [9, 18, 48, 49] and model-based FL [50, 51, 52, 53].

IV. Design

Combining FL with version history control information can be done by using HSFL. For environments were batches of code changes from multiple authors are tested simultaneously, this poses a challenge. For HSFL, it is required to know which changes are inducing and non-inducing, e.g. which changed statements belong to a set of changes which cause tests to fail and which do not. To determine which changes are inducing and non-inducing, the execution of tests after every change is required. For environments where batches of changes are tested simultaneously, this is impossible. For these environments, a different approach is required.

To improve existing fault localization for environments where batches of changes are tested simultaneously, we introduce GitFL. GitFL is a fault localization method that combines traditional fault localization (FL) with the version control history (VCH).

$$sus(e) = \begin{cases} (1-\alpha) * FL(e) + \alpha * VCH(e) & \text{if } FL(e) \neq 0 \\ 0 & \text{if } FL(e) = 0 \end{cases} \tag{5}$$

In formula 5, $sus(e)$ is the suspiciousness for the executed statement e. $FL(e)$ is the suspiciousness for a statement returned by an arbitrary standard FL algorithm using coverage information and test outcomes. $VCH(e)$ is the suspiciousness for a statement based on the version control history. The weighted combination of $FL(e)$ and $VCH(e)$ is determined by the weighting factor α. When a statement is never executed during failing tests, it has no reason to be suspicious and thus should not be considered suspicious. Accordingly, when $FL(e)$ is equal to zero, $sus(e)$ is set to zero, to prevent a non-zero $sus(e)$ caused by VCH. To calculate $VCH(e)$, several assumptions are made.

For calculating VCH, one could include all changes ever made in the repository with equal weights. With this approach, all code-changes (now referred to as 'changes') are treated as potentially failure-inducing changes. Failure-inducing changes are changes that cause test(s) to fail by introducing bugs. This would add the same suspiciousness value for every line of code, as every line of code was changed (adding statements are also changes) at some point, not strengthening the standard fault localization method.

An improvement can be made with the assumption that bugs are fixed shortly after they are discovered. This assumption is expected to hold, as fixing bugs in an early stage of development is crucial for effective software development [4]. It can be deducted that a bug in an older

failure-inducing change is more likely to have already been fixed, compared to a bug in a newer failure-inducing change. Thus, newer failure-inducing changes should increase the suspiciousness more than older ones. This relationship can for example be linear or exponential. $VCH(e)$ is calculated using a decaying exponential relation, meaning that new changes will result in an exponentially higher suspiciousness, resembled by the exponential function in equation 6 and visualized by the plot in figure 3 in the range $DBIT$ to c_f. A disadvantage of this approach is that the entire change history of the repository needs to be evaluated, which can be a time-consuming task when the repository is large.

This disadvantage can be mitigated with the assumption that there is recent point in time that is free of failing tests (e.g. all tests passed), which limits the evaluated change history to a more recent, smaller time range, likely significantly reducing the number of changes. This assumption can be tricky to make, since it is not necessarily required for all tests to pass before the next release can be delivered. Not all tests are as important for realizing a sufficiently functioning system, possibly resulting in unsolved failing tests before a release. These tests could be neglected due to development prioritization. However, as releasing a non-functioning system would be unwise, it can be assumed that most tests pass with only the critical tests in the worst case. Additionally, all tests passing does not necessarily mean that no bugs are present, as adding more tests cases will not ensure the localization of all bugs [3]. A more realistic approach is to pick a point in time where the majority of tests passes (including all critical tests). From this point in time, all changes are considered potentially failure-inducing, up until the last change made. All changes before that point in time, are considered as non failure-inducing changes. This is resembled by the otherwise case in equation 6 and the range before $DBIT$ in figure 3. If, for a specific repository, assumption two cannot hold, then assumption three also cannot hold, as there will be old unfixed bugs present.

$$VCH(e) = \begin{cases} e^{-d_f \frac{c_f - c_b}{DBIT}} & \text{if } c_f - DBIT \geq c_b \geq c_f \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Here, c_f is the date of the final change, e.g. the last change made in the current release. c_b is the date of the last change that changed the statement. $DBIT$ stands for days back in time and determines up until which date changes should still be considered to be potentially failure-inducing. d_f is the decay factor. When the value of d_f is high, the suspiciousness of new changes will be high compared to old changes. d_f can be adjusted to the environment. For example, when it is known that bugs are usually fixed later after they are discovered, the value of d_f should be lowered, to allow older changes to still significantly influence the suspiciousness.

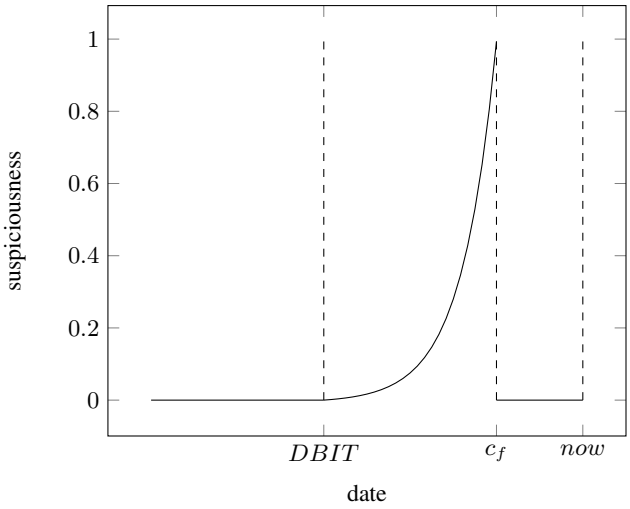


Fig. 3: Decaying exponential relation between change date and suspiciousness with a decay factor of 5. The final commit date c_f is 7 days ago relative from now. $DBIT$ is 37 days ago. In between $DBIT$ and c_f, newer changes receive an exponentially higher suspiciousness compared to older changes. For changes outside this range, the suspiciousness is zero.

TABLE II: Sample program for finding the n-th number in the Fibonacci sequence.

```
Line  Code
1     unsigned int fib(unsigned int n) {
2       if (n == 0) {
3         return 0;
4       }
5       else if (n == 1 || n == 2) {
6         return 1;
7       }
8       else {
9         return fib(n - 1) + fib(n - 2);
10      }
11    }
```

TABLE III: Unit tests for testing the Fibonacci sample program.

```
Line  Code
1     TEST_METHOD(T1) {
2       Assert::AreEqual(0, fib(0));
3     }
4     TEST_METHOD(T2) {
5       Assert::AreEqual(6, fib(8));
6     }
```

TABLE IV: Changes made to the sample program by a developer. They refactored the code to make it more compact and added a logging statement. By refactoring, they introduced a bug, namely the if statement always returns 1 but should return 0 for n=0.

```
unsigned int fib(unsigned int n) {
  if (n == 0) {
    return 0;
  }
  else if (n == 1 || n == 2) {
    return 1;
  }
  else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

Line	Code
1	`unsigned int fib(unsigned int n) {`
2	`  if (n == 0 || n == 1 || n == 2) {`
3	`    return 1;`
4	`  }`
5	`  else {`
6	`    std::cout << logging statement`
7	`    return fib(n - 1) + fib(n - 2);`
8	`  }`
9	`}`

TABLE V: Overview of the concept of GitFL, indicating which statements are changed, which statements are executed by the failed test case T1 and which statements are executed by the passed test case T2. In the last column, the suspiciousness is shown, indicated by –, -, 0 , + or ++. When a statement is changed or executed by a failed test case, the suspiciousness raises. When a statement is executed by a passed test case it lowers.

Line	Code	Changed	T1	T2	Suspiciousness
1	`unsigned int fib(unsigned int n) {`		✓	✓	0
2	`  if (n == 0 || n == 1 || n == 2) {`	✓	✓	✓	+
3	`    return 1;`	✓	✓		++
4	`  }`	✓	✓	✓	+
5	`  else {`			✓	-
6	`    std::cout << logging statement`	✓		✓	0
7	`    return fib(n - 1) + fib(n - 2);`			✓	-
8	`  }`			✓	-
9	`}`			✓	-

TABLE VI: Output of GitFL consisting of a list of statements ranked on their suspiciousness. Statement 3, the buggy statement, is ranked as the first statement that the developer should examine.

Line	Code	Suspiciousness
3	`    return 1;`	++
2	`  if (n == 0 || n == 1 || n == 2) {`	+
4	`  }`	+
1	`unsigned int fib(unsigned int n) {`	0
6	`    std::cout << logging statement`	0
5	`  else {`	-
7	`    return fib(n - 1) + fib(n - 2);`	-
8	`  }`	-
9	`}`	-

The tables II, III, IV and V illustrate the concept of GitFL. Table II shows a sample program for calculating the n-th number from the Fibonacci sequence. Two test cases exist for the program and are described in table III, one asserting that the 0th Fibonacci number returns 0 and one asserting that the 6th Fibonacci number returns 8. A developer refactors the program with the goal of making the code more compact. These changes are shown in table IV. They merge the two if-statements and accidentally introduce a bug. Namely, returning 1 for both the 0th, 1st and 2nd Fibonacci number, while the 0th number should return 0. The developer did not notice they introduced a bug and runs the two tests T1 and T2 described in table III. They notice that test T1 fails but do not know the reason for it to fail.

To locate the buggy statement, they use GitFL. The combination of traditional fault localization and the version control history is visualized in table V. The traditional fault localization FL is resembled by the columns 'T1' and 'T2', indicating if the statements are executed during these failing and passing tests. The version control history VCH is resembled by the column 'changed', indicating if the statements are changed by the developer. When a statement is changed or executed by a failed test case, the suspiciousness raises. When a statement is executed by a passed test case it lowers. The suspiciousness is described by a –, -, 0, +, ++, resembling its value.

After calculating the suspiciousness of each statement,

TABLE VII: Visualization of placeholder for deleted lines.

```
else if (n == 1 || n == 2) {
  return 1;
}
```

Line	Code
1	`unsigned int fib(unsigned int n) {`
2	`  if (n == 0) {`
3	`    return 0;`
4	`  }`
5	`  std::cout << executing deleted lines`
6	`  else {`
7	`    return fib(n - 1) + fib(n - 2);`
8	`  }`
9	`}`

all statements are ranked according to their suspiciousness, starting with the statement with the highest suspiciousness. Now, the developer examines the output of GitFL, being the ranked list of statements. They start with the statement with the highest suspiciousness and find the bug. Note that this example was made purely with the goal of explaining the concept of GitFL.

The term 'change' refers to the addition, modification or deletion of statements. For both addition and modification, GitFL operates as expected, as the affected statements are still present after addition or modification. For including deletions in GitFL, we chose to use placeholders. A placeholder consists of a statement which does not affect the program logic but is always executed when the delete lines of code would be executed, to ensure detection by GitFL. This can for example be a statement that generates a log message, which is visualized in table VII. Here, statements 5-7 of the original program are deleted and replaced by statement 5 in the new program, which outputs the text 'executing deleted lines' to the console. For every consecutive block of deleted statements, one placeholder is required.

V. Implementation

A. Environment

Adyen is a financial service provider which, among other others, implements solutions for in-store payments. Customers can pay in the stores of merchants by using payment terminals. One of the available terminal models used in many stores, the Verifone P400Plus, is shown in figure 4. Customers can, for example, pay by inserting or tapping their card or phone and enter their pin-code on the numeric pad, if required. To enable these payments, software needs to run on the terminal, which handles user interaction and payment processing. This software is developed and maintained by Adyen.

The terminal software is continuously maintained and improved upon through updates. However, before deploying updates to terminals, it is essential to ensure that the new software behaves as expected. If bugs are present, which potentially cause terminals to crash or to not properly

Fig. 4: Verifone P400Plus terminal, provided by Adyen to merchants for in-store payments. This specific model is used for the evaluation of GitFL.

being able to process transactions, the situation could occur wherein customers are not able to pay at merchants. To prevent this, the terminal software is extensively tested. Next to unit-tests which test individual components, end-to-end tests are used to ensure that the whole system operates as expected. To end-to-end test the terminal, real-life scenarios are tested, simulating a paying customer. For example, by mocking a merchant by initiating a payment request for an arbitrary amount, after which a card is presented, a pin-code is entered and the transaction is processed. Such an end-to-end test implicitly tests many parts of the code. However, presenting a card and entering a pin-code on the numeric pad are physical actions which cannot be mocked by only using software. The solution for this is using robots to simulate real-life user interaction by physically testing the terminals and thus implicitly testing the terminal software.

The robot testing setup at Adyen is shown in figure 5, 6 and 7. The robots are primarily made out of additive manufactured parts and servo motors. Each robot has a terminal mounted on the base, a robotic arm with a finger for pressing buttons and interacting with a touch-screen, and they have modules for three payment methods installed,

Fig. 5: Robot test setup at Adyen for the end-to-end testing of terminals by simulating real-life user interaction. The robot has the ability to insert, swipe or tap cards and provide input on the numeric pad and screen.

namely, inserting a card in the terminal, swiping card though the side of the terminal and tapping the top of the terminal. A camera is mounted on the top bracket to enable remote surveillance of the robot.

During these robotic end-to-end tests, batches of code-changes from multiple authors are tested simultaneously, as running many robot tests after each change is unfeasible due to robot capacity and time. To illustrate, consider a test suite with hundreds of robot tests. If all these robot tests need to be executed after every code change, the tests are probably not finished before the next change is introduced, which results in an ever-growing testing queue. However, because multiple changes from multiple authors are tested simultaneously, it can be unclear which change (and thus which author) caused a robot test to fail. To combat this, GitFL was implemented at Adyen to locate these buggy changes.

B. Data collection

To localize faults in this particular environment, a tailored testing and data collection pipeline was required. The reason for this was that implementing FL for an embedded system was more complex compared to software-only systems.

Fig. 6: Front view of the robot test setup at Adyen.

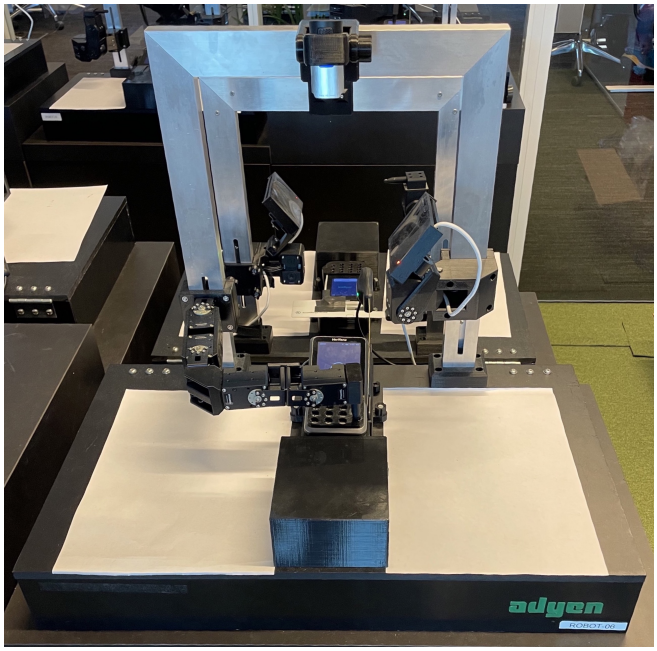


Fig. 7: Close-up view of the robot interacting with the terminal during a test.

For software-only systems, coverage information collection tools are often already integrated in the IDE. However, for these embedded systems, coverage information collection tools had to be integrated manually.

The routine is shortly described here and is written as pseudo-code in algorithm 1. First, an individual test from the collection of tests is executed. When that individual test is finished running, the next test in the list is not immediately executed. Before doing so, the coverage information related to the just-executed test, stored in the memory of the target (which, within this environment, resembles the terminal), is written to the flash memory of the target. Then, the coverage files are downloaded from the target to the local machine (which, within this environment, resembles a laptop) for storage. These files will be processed after finishing executing all tests.

Algorithm 1 Pseudocode of the data collection pipeline.

```
for each test in test case list do
    Schedule test
    Start robot
    if test is not completed yet then
        Wait on test to finish
    end if
    Save coverage information on target
    Extract coverage information from target to local machine
end for
```

For obtaining the coverage information, tools are available. For example, there is Gcov [54], for C++ code, gcovr [55] for Python and Jcov [56] for Java. There are no restrictions for most other popular programming languages, for which tools are available too [57, 58, 59].

For the setup at Adyen, coverage information was obtained with Gcov. Gcov was enabled by setting specific flags before compiling the source code for the target. To log the coverage information, Gcov generates two type of files, being *.gcda* files and *.gcno* files. The *.gcno* files are generated at compile time and placed in the same directory as the program source code (on the machine used for compiling the code). These *.gcno* files hold the instructions for logging the coverage information at runtime. During runtime, Gcov logs the coverage information to the target's memory, specifying which statements are executed and how often they are executed. When the method $gcov\_dump$ is called (which is automatically called at a restart), the coverage information on memory is written to a *.gcda* file on the target's storage. After every test, the *.gcda* files need to be downloaded from the target to the local machine, to avoid replacing the data in the file with the data from the next test. Then, the *.gdca* and *.gcno* files need to be combined into a *.gcov* file. The *.gcov* file holds human-readable coverage information logs, indicating which statements are executed and how often.

To ensure obtaining complete coverage information, compiler optimizations needed to be turned off. For example, if a code change causes an if-statement to always evaluate to true, it would be unnecessary to evaluate it and the compiler optimizations would generate instructions that result in never evaluating the if-statement. However, if this buggy if-statement is never executed, it cannot be detected by GitFL.

After executing all tests, the *.gcda* files were combined with their accompanying *.gcno* files to generate the readable *.gcov* files. However, the *.gcda* files were located on the local machine, while the *.gcno* files were located on the machine used for compilation, where they were generated. Normally, this would be the same location, being the local machine. However, in this specific case, the code could not be compiled on the local machine due to the local compiler not supporting the Gcov tool. For this reason, the code was compiled on a cloud-based server. The Gcov tool can only successfully merge the *.gcda* and *.gcno* files when both files are located in the original source directory used for compilation. This required moving the *.gcda* files to the server. Furthermore, it was required that the *.gcda* and *.gcno* of the same file were placed in the exact same directory, for the Gcov tool to be able to merge them. This is described in algorithm 2.

Algorithm 2 Pseudocode of the algorithm that finds, matches and moves *.gcno* and *.gcda* files.

```
find all .gcda files on local machine
find all .gcno files on buildserver
for each .gcda file do
    path ← get path to file
    path ← remove part of path belonging to buildserver
    .gcno path ← path
    copy .gcda file to path of .gcno file path
end for
```

An example of a .gcov file is shown in VIII. It starts with the version number of Gocv. On the next line, the path to the source file is listed, indicated by the 'file' keyword. Below, the executed functions are listed, indicated by the 'function' keyword. The first two numbers resemble the start and end statements of the function. The third number indicates how many times the function has been executed, followed by the name of the function. Continuing, the 'lcount' keywords indicates the individual statements. The first number represents the line number of the statement, the second number is the number of times the statement has been executed and the third number indicates

TABLE VIII: Example of .gcov file contents.

```
version:8.3.1 20190311 (Red Hat 8.3.1-3)
file:path/to/source/file
function:37,41,3,function1
function:45,47,6,function2
lcount:37,3,0
lcount:38,3,0
lcount:39,3,0
lcount:40,3,1
lcount:43,2,0
lcount:44,6,0
lcount:45,6,0
lcount:47,6,0
file:path/to/first/header/file
function:2,4,1,function1 lcount:2,1,0
lcount:4,1,0
lcount:5,0,0
lcount:6,0,0
lcount:83,0,0
lcount:87,0,0
file:path/to/second/header/file
function:46,50,2,function1 lcount:46,2,0
lcount:47,2,0
lcount:48,2,0
lcount:50,2,0
lcount:83,1,0
lcount:87,1,0
```

if there is a not-executed block present within the statement.

Consequently, executed header files linked to the source file are listed. For simplicity, it was assumed that the header files were free of bugs, thereby ignoring their coverage information. Additionally, only the 'lcount' rows are required to be parsed, as GitFL works with statement-level coverage (meaning GitFL requires data describing the coverage information of all statements), therefore it is not necessary to include the 'function' rows for parsing. The parser described in algorithm 3 parses the coverage information in .gcov file format to usable data saved in a database. It does this by iterating over all lines in the .gcov file. To ensure that only the first file is processed, it checks if the keyword 'file' appeared before. If that is the case, it continues to the next .gcov file. The executed statements are added to the 'Coverage' table in figure 8.

The GitFL algorithm needs access to the change history in order to work properly. More specifically, it requires the exact statements which are changed within the date range of interest. In other words, if the date range of interest is 30 days, GitFL requires all statements that have been changed in the last 30 days together with their change date. Within the Adyen environment, Git is used for version control history. Because of this, Git blame [60], a command belonging to Git, can be used to fetch the changed statements.

Git blame takes a filename as input and outputs all statements of that file, accompanied by the latest changes of those statements. This indicates when and by who a statement is last changed. However, for GitFL, only the changes within the date range of interest are required. These changes are used as input for GitFL while the changes outside of the date range are excluded. Algorithm 4 describes the data collection routine with Git blame. For each file that is executed at least once (present in the 'Files' table in figure 8), the version control history is fetched using Git blame. Then, it loops over the lines of the Git blame output, which resemble the statements of the file with their last change date. If a statement is changed within the date range of interest, it is added to the list of changed statements. These statements, together with their change date, were added to the 'ChangedLines' table in figure 8.

Algorithm 3

```
for each .gcov file do
    for each line in file do
        if line contains 'file' then
            if 'file' is already seen before then
                continue to next .gcov file
            end if
            save path to database
        end if
        if line contains 'lcount' then
            save line to database
        end if
    end for
end for
```

Algorithm 4 Pseudocode of the version control history data collection.

```
Initialize list of changed statements
for each file executed by any test do
    Retrieve changes of the file with Git Blame
    for each statement in the Git Blame output do
        if statement is changed within the determined date
range then
            Add statement to list
        end if
    end for
end for
return list
```

Next to coverage information and changed statements, GitFL required the test outcomes. With the test outcomes, GitFL can determine if a statement is executed by either a passed or a failed test. The framework used for executing the robot tests was already in place. Next to executing tests, the framework also kept track of the test outcomes. After executing all tests, the test outcomes were extracted using the API of the testing framework. The API response was not in the correct format used in the database in figure 8, which required the response to be parsed. The parsed test outcome data was added to the 'Tests' table in figure 8.

C. Data structuring

All relevant coverage information and test data was stored in a relational database with multiple tables, visualized in figure 8. The 'Files' table holds the names and paths of all source files that contain at least one statement which is executed during testing. The 'Tests' table hold the names and outcomes of all tests that were executed. The test outcome can attain either 'passed' or 'failed', which is represented by a one or a zero, respectively. The 'Coverage' table holds all statements which were executed, linked to the corresponding test by which they were executed and which file they belong to. The 'ChangedLines' table holds all statements that were recently changed within the date range of interest, accompanied by the exact date and time indicating when the statement was last changed. Finally, the 'Sus' table holds the suspiciousness returned by GitFL for all executed statements.

GitFL requires a specific format as input, visualized in table IX. For the input, each row represents data for an individual executed statement. This includes the file id and the line number of the statement, by how many passed and failed tests the statement was executed and if the statement was changed and, if so, on which date. This data format is obtained from the database in figure 8 with the following pseudo-query:

```
for each executed statement in every
test in the coverage table, count how
many times it is executed by failed and
passed tests by referencing to the tests
table. Determine if the statement was
recently changed and if so, when, by
referencing to the changedlines table.
```

TABLE IX: Data structure of the input of GitFL.

FileID	LineNumber	Passed	Failed	Changed	Change date
1	1	14	2	0	Null
1	36	12	6	1	4 days ago
2	15	3	20	1	1 day ago
3	2	15	2	0	Null
3	3	15	2	0	Null
3	143	3	24	1	4 days ago
...	...	...	...	...	...

D. GitFL

The GitFL algorithm is described by algorithm 5. It resembles an implementation of formula 5. It loops over all executed statements and first computes its partial suspiciousness for traditional FL with the Tarantula formula shown in equation 1. If this value is zero, the final suspiciousness of this statement becomes zero, as it is never executed by a failed test and thus has no reason to be suspicious. Otherwise, the partial suspiciousness for VCH is computed with equation 6. Finally, both partial suspiciousness values are combined in a weighted combination to obtain the final suspiciousness value for the statement. The resulting list of statements with their suspiciousness is then sorted in a descending order to ensure the most suspicious statements being at the top of the list, which is the output of GitFL.

Algorithm 5 Pseudocode of the GitFL algorithm, which returns a list of statements, ranked on their suspiciousness.

```
Initialize empty list SusList to store suspiciousness of statements
for each statement in executed statements list do
    Sus ← 0
    FL ← compute FL
    if FL is zero then
        Sus ← 0
        SusList ← Sus
        skip to next iteration
    end if
    VCH ← compute VCH
    Sus ← (1 − α) ∗ FL + α ∗ VCH
    SusList ← Sus
end for
Sort SusList on suspiciousness, descending
Return SusList
```

VI. Evaluation

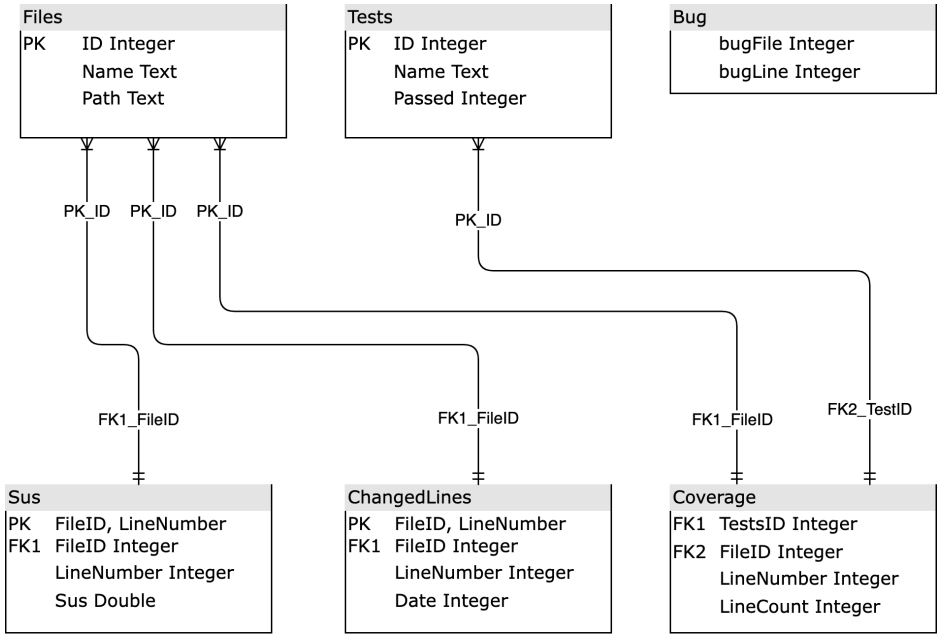
The research question posed in the introduction was "What is the effectiveness of GitFL for environments where batches of changes are tested simultaneously?". To answer this research question, an empirical study was conducted to evaluate the effectiveness of the newly introduced method GitFL. Specifically, this evaluation was done on the Adyen case.

As GitFL combines existing FL with the version control history, an existing FL method needed to be selected for evaluation. The performance of the method had to be good enough to produce usable results but did not necessarily have to be the best available, as the main goal was to produce insights on the contribution of version control history information. This requirement resulted in the selection of Tarantula [6], a popular and simple to implement FL method.

A. Artificial cases

The artificial test suite used for evaluation was manually generated. A recent build, for which all end-to-end tests passed, was chosen. With this build, multiple faulty versions were generated by introducing mutations in the code. A faulty version is a unique build with a unique bug or mutation. These mutations had the objective to cause test failures, which classifies them as bugs. A selection of mutation operators from the Java mutation testing tool PIT

Fig. 8: Relational database for storing coverage and test data, to be used as GitFL input, later on.



from Coles et al. [61] was used. The selection was made with the objective of causing test failures in mind. For example, the probability of the negate condition operator ($==$ to $!=$) to cause failures is expected to be significantly greater than the condition boundary operator ($<$ to $\leq$), as boundaries are usually tested in unit tests instead of end-to-end tests. PIT's mutations were extended by additional mutant operators, such as null replacement and switch case. The selected mutation operators are shown in table X. To illustrate how mutations can affect test outcomes, example code is provided in the appendix (XII) and walked through.

All mutations in table X were applied to several source files, belonging to the collection of code for a specific transaction protocol. Most mutations were applied multiple times on different locations in the source code. Only files affecting this transaction protocol were considered, to reduce the scope of the test suite. The mutations were chosen in such a way, that they primarily affected the business logic of the programs. For example, by omitting a certain response field for a transaction. Affecting only business logic was done to ensure the programs did not cause the embedded system to crash, which would result in the inability to further log coverage information and prevent GitFL to perform as expected. Although these mutations, which cause the system to crash, are also considered to be bugs, they fall out of the scope of this project, as coverage information is essential to GitFL.

When a mutation was made, it was added to the version control history, similar to a developer making a change. Every change in the version control history has a date. This

TABLE X: Mutation operators selected from Coles et al. [61] (edited)

Name	Transformation	Example
Negate Condition	Negates one relational operator (single negation).	$==\rightsquigarrow!=$
Return Values	Transforms the return value of a function (single replacement).	$return\ 0 \rightsquigarrow return\ 1$
Method Call	Deletes a call to a non-void method.	$int\ m() \rightsquigarrow$
AOD	Replaces an arithmetic expression by one of the operand.	$a + b \rightsquigarrow a$
ROR	Replaces the relational operators with another one. It applies every replacement.	$<\rightsquigarrow\geq$
Remove Condition	Replaces a cond. branch with true or false.	$if(...) \rightsquigarrow if(true)$
Null replacement	Replace variable by null	$a = 5 \rightsquigarrow a = null$
Switch case	Add, swap or delete a switch case	

gave rise to the following question: which date should the artificial change get? The change date should by all means be in the $DBIT$ range, as changes outside this range are not recognized by GitFL. With the assumption that bugs are usually fixed shortly after they are discovered, changes that induce not-yet-fixed bugs are more likely to appear closer to the date of the final change. Based on this reasoning, the date of an artificial change was sampled from a exponential decay distribution with a decay factor similar to the decay

factor in the GitFL parameters. This distribution aims to increase the realism of bug-inducing change dates by favoring dates closer to the final change date.

To research the influence of the decay factor parameter of the GitFL algorithm, the decay factor was varied. At this point, the influence of the weighting factor α was not known yet. For this reason, both the decay factor d_f and the weighting factor α were varied simultaneously. For each combination of α in the range of zero to one with a step size of 0.1 and d_f with the values 0.1, 0.3, 0.5, 0.7, 1, 2, 4, 8, 15, the performance of GitFL was evaluated. The performance was determined by inspecting the rank of the buggy statement returned by GitFL for each combination of values. For each value of α, the best performing value of d_f was listed. Inspecting this value for all values of α gave insight in the relation between d_f and α. It often occurred that the rank was equal for multiple values of d_f for a specific value of α. When this occurred, the suspiciousness value was inspected to differentiate between the tied statements.

B. Real-life cases

Although the amount of real-life test data was too limited for proper evaluation, it was not discarded. Several cases using real-life data were used for additional evaluation to test the generalization of GitFL. Compared to the artificial cases, the scope of the collection of the real-life cases was significantly broader, meaning that a larger part of the system was involved.

C. Finding the GitFL parameters that lead to optimal performance

The GitFL algorithm has several parameters that can be varied, resulting in a varying output and performance. Determining analytically which set of properties results in the best performance is difficult (what this best performance entails is defined in section VI-D). Each environment (e.g. code-base) requires different parameters for the best possible performance. In addition, the set of all possible parameters variations is large, which means solving for optimal parameters is expected to be time-consuming. For this reason, it was empirically determined which set of properties resulted in the best performance.

Combination weight (α): α determines the weighted combination of standard FL (FL) and the version control history (VCH). This property is arguably the most interesting, because varying it has the potential to show how both FL and VCH perform stand-alone ($\alpha = 0$ and $\alpha = 1$, respectively) and what the contribution of VCH to FL looks like in terms of performance. It is therefore varied within the range of zero to one.

Days back in time (DBIT): the value of DBIT determines how far back in time the algorithm will search for changes. For example, if DBIT is set to seven, GitFL will include all changes made up to seven days ago. DBIT is often implicitly linked to the number of changes that the algorithm considers. Only when no changes are made in a time range, the number of changes stays the same with a varying value for DBIT. The influence of this property is difficult to determine analytically. When DBIT is increased, often more changes are included. Those additional changes could potentially increase the suspiciousness of non-buggy lines, causing them to rank higher than the buggy line, influencing the performance of GitFL. The value of DBIT was kept constant, equal to the point in time were no tests were failing.

Decay factor (d_f): the exponential decay relationship between the change date and the suspiciousness is determined by the decay factor d_f. A higher decay factor results in a stronger exponential relation, meaning older changes will contribute less influence on the suspiciousness. Apart from varying the decay factor, it was investigated if other relations performed better, such as a constant- or linear relation.

D. Analysis

In the FL literature, new methods are often evaluated using the EXAM score [7]. The EXAM score consists of the percentage of code to examine vs the percentage of bugs in faulty versions located. The percentage of code to examine refers to the number of ranked statements, outputted by the FL method, which the developer has to examine before finding the bug. For example, suppose that a certain program contains 100 statements and one of those statements contains a bug. This program is tested with several tests and the executed statements are logged for each test. An arbitrary FL method is used to locate the bug and the output of the algorithm is a ranked list of all statements. The higher the rank, the higher the probability that the statement contains the bug. If the buggy statement is ranked twelfth, the developer first has to examine the eleven higher ranked statements (which do not contain the bug) before finding the bug in the twelfth ranked statement. The developer has to examine twelve percent of the code to locate the bug.

For every faulty version of both the artificial and real-life cases, the GitFL algorithm was applied several times, each time with varied algorithm properties, to empirically determine which properties resulted in the best performing algorithm. The percentage of code to examine (equal to as the rank of the buggy statement) is visualized in a table for each GitFL variation for each build.

With a large number of faulty versions and algorithm variations, the table with the ranks of buggy statements becomes difficult to interpret. A solution that comes to mind would be to take the average of the percentage of code to examine. However, this would result in a loss of information. This is where the EXAM score comes in, which visualizes which percentage of code needs to be examined for multiple faulty versions. The result is an accumulative plot, where it is shown which minimum percentage of code needs to be examined for which number of faulty versions. If the percentage to examine is smaller for a larger number of faulty versions, the performance is considered to be better. The x-axis is logarithmic and indicates the percentage of code examined and the y-axis indicates the number of faulty versions where the fault is successfully located.

Algorithm 6 Pseudocode for finding the rank of buggy statement.

```
SusList
BugList
initialize rank
counter ← 0
for each statement in SusList do
    for each bug in BugList do
        if statement location is equal to bug location then
            rank ← counter
            break
        end if
    end for
    counter increments 1
end for
```

For the analysis of GitFL, the rank of the buggy statements in the output of GitFL needed to be determined. Due to knowing beforehand which statements are buggy (saved in the 'bug' table in figure 8), their rank can be determined. The output of GitFL consists of a list of statements, sorted descending on their suspiciousness. To obtain the rank of the buggy statement in this list, algorithm 6 is used. The algorithm iterates over the sorted suspicious statements and checks if the current statement is a buggy statement. If this is the case, the rank is equal to the iterator number plus one, because the iterator starts at zero while the rank starts at one. When a build has multiple buggy statements, the rank will be the rank of the first buggy statement encountered. The list of statements is always sorted on suspiciousness, which results in the rank always being the highest rank of all buggy statements.

VII. Results

The evaluation resulted in table XI and XII, for artificial and real-life builds, respectively. Figure 9a shows the EXAM scores of the multiple variations of the GitFL algorithm, which was constructed with the data from table XI. For the real-life results from table XII, no EXAM score was plotted. As these consisted of a couple of builds, they interpret without the need for visualization.

A. Artificial test suite

For all values of α, the best performing value of d_f was 0.1. This indicates that α and d_f were not related. E.g. adjusting α did not require adjusting d_f for constant results.

It can be observed in figure 9a that for eight evaluated GitFL variations (excluding $\alpha = 0.0$ and $\alpha = 1.0$), the EXAM score plot shows a steep ramp from zero to approximately half to two thirds of the number of faulty versions, approximately at 0.005 percent of code examined. From that point on, the EXAM score starts to diverge for the different GitFL variations.

GitFL with $\alpha = 0.0$ was under-performing relatively to the other variations. Here, only FL was activated by setting VCH to zero. In other words, GitFL was equal to the traditional FL of choice, being Tarantula, ignoring any version control history information. In figure 9a, the plot of $\alpha = 0.0$ is significantly shifted to the right compared to the other variations. Especially when compared to $\alpha = 0.4$ to $\alpha = 0.9$, where the rank of the buggy statement is often within the top-5 statements. This shows the additional performance to traditional FL when the version control history is included, for this specific evaluation.

GitFL with $\alpha = 1.0$, meaning only VCH is enabled by ignoring traditional FL, performs significantly better than $\alpha = 0.0$. Compared to the other variation, at first it seems to perform worse. However, at 0.1 percent of the code examined, it found the bug for all faulty versions, outperforming α = 0.1, 0.2, 0.3. 0.1 percent of the code examined is roughly equal to 25 statements to examine before finding the bug.

The best performing variation is $\alpha = 0.7$, outperforming all variation at any point in the graph. $\alpha = 0.7$ resembles a combination of FL and VCH with a slight inclination to VCH, meaning the suspiciousness of the version control history is favored above traditional FL. More detailed performance in a small range of percent of code examined can be found in 9b

B. Real-life test suite

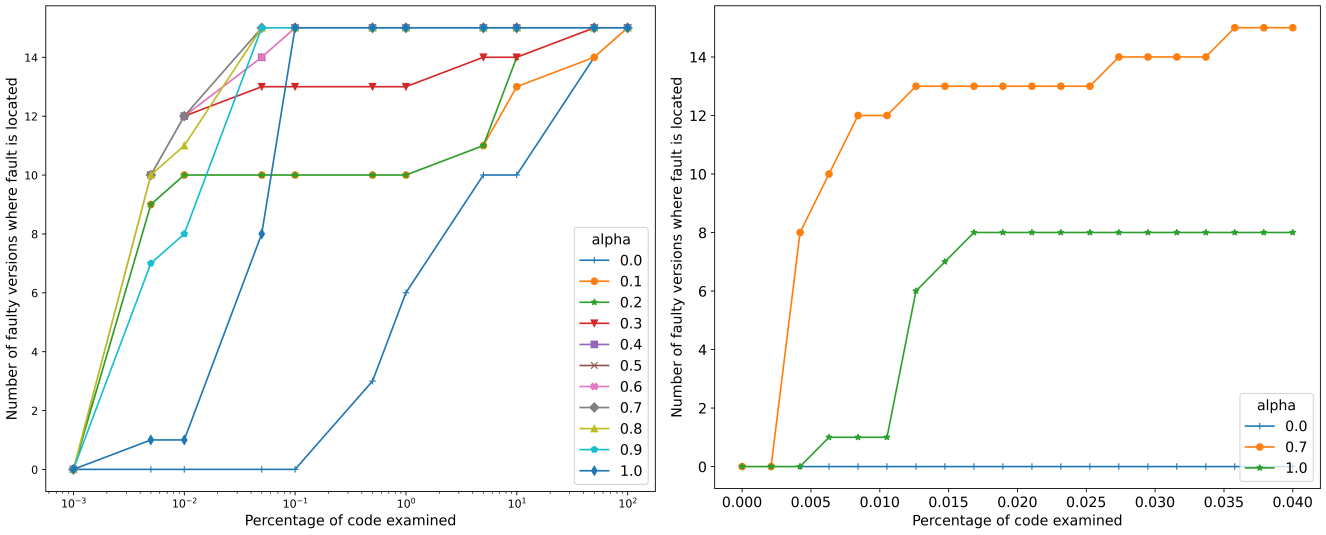
The first real-life case consistently returned a rank of two, except for $\alpha = 0.0$. In other words, when VCH is enabled, the developer has to examine two statements to locate the buggy statement. The second real-life case showed a rank of five, for all variations except α = 0.0 and 0.1, meaning that the developer found the bug after inspecting the first five statements suggested by GitFL.

TABLE XI: Performance evaluation of GitFL, evaluated on an artificial test suite consisting of 15 builds each with a unique fault-inducing mutation, for eleven values of α ranging from 0.0 to 1.0. For each build and value of α, the rank of the buggy statement is displayed. The rank resembles how many statements a developer has to examine before examining the buggy statement.

	value of α											total number
Build	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	of lines
1	3105	1	1	1	1	1	1	1	1	1	1	22304
2	190	1	1	1	1	1	1	1	1	3	25	25641
3	162	1	1	1	1	1	1	1	1	1	3	24198
4	31	2	2	2	2	2	2	2	2	2	4	24947
5	1011	643	473	1	1	1	1	1	1	1	3	22798
6	8843	2112	1388	740	6	6	6	3	3	3	3	25867
7	3760	2753	1737	2	2	2	2	2	4	4	23	25418
8	703	1	1	1	1	1	1	1	1	3	16	25667
9	68	1	1	1	1	1	1	1	1	1	3	25370
10	60	1	1	1	1	1	1	1	1	1	3	25766
11	533	1	1	1	1	1	1	1	1	1	14	25650
12	20368	14608	7062	6044	16	16	15	9	9	7	3	25547
13	7250	2139	1862	7	7	7	7	7	9	7	23	25755
14	298	1	1	1	1	1	1	1	1	9	25	26134
15	195	1	1	1	1	1	1	1	1	1	25	25228

TABLE XII: Performance evaluation of GitFL, evaluated on a real-life test suite, for eleven values of α ranging from 0.0 to 1.0. For each build and value of α, the rank of the buggy statement is displayed. The rank resembles how many statements a developer has to examine before examining the buggy statement.

	value of α											total number
Build	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	of lines
1	1707	2	2	2	2	2	2	2	2	2	2	23023
2	631	5	5	5	5	5	5	5	5	5	16	24271



(a) 0 to 100 percent of code examined with logarithmic scaling.

(b) 0 to 0.04 percent of code examined with linear scaling.

Fig. 9: Performance evaluation of GitFL, evaluated on an artificial test suite consisting of 15 builds each with a unique fault-inducing mutation, for eleven values of α ranging from 0.0 to 1.0, with $DBIT = 30$ and $d_f = 0.1$. The performance is visualized using the EXAM score, resulting in an accumulative plot, where it is shown which minimum percentage of code needs to be examined for which number of faulty versions. If the percentage to examine is smaller for a larger number of faulty versions, the performance is considered to be better.

VIII. Discussion

A. Discussion of results

The EXAM scores obtained through the evaluation of GitFL are significantly higher compared to state-of-the-art fault localization methods, such as D-star [7], Ochiai [8], and DNN [38]. An explanation for this could be that GitFL makes use of version control history, as opposed to D-star, Ochiai, DNN, and most other FL methods. Logically, combining a state-of-the-art FL method with information from the version control history of the code-base at hand should narrow down the possible bug-inducing statements immensely.

Within the Adyen case, the evaluation showed a clear difference in performance between only using FL and combining it with the version control history. Namely, the addition of the version control history significantly outperforms standard FL, which was Tarantula. It can be argued that the GitFL output for standard FL-only was unusable by developers, according to the reasoning that developers will discard the tool when they have to inspect at least 30+ statements before discovering the buggy statement [62, 63]. While this observation shows the contribution of GitFL, it raises the question why the Tarantula-only variation performed sub-par. A potential explanation has to do with the inner workings of Tarantula and the test suite coverage. End-to-end tests are focused on testing higher-level functionality and are usually more time consuming. This results in less-granular testing, and the number of tests that can be executed within the same timeframe is likely to be lower compared to unit tests. Because of this, their test coverage is likely to be lower compared to the test coverage of unit tests. Lei et al. showed that more passing tests executing non-faulty statements, which is a result of high coverage, increase FL performance [12]. This highlights the value of the addition of version control history information for environments where test coverage is limited.

The second real-life case was wrongly selected, as the test failure was not related to the specific terminal being tested in this scenario. However, the change holding the fix also hold a change specific to the terminal in this scenario. This showed that, although the failure had nothing to do with the terminal, the influence of the version control history part in GitFL was strong enough to present this statement as the most suspicious one.

Instead of manually generating bugs and evaluating on an artificial test suite, future work could evaluate GitFL on a substantially larger real-life data-set, by considering a more suitable code base or test environment for the objective in mind. Multiple solutions come to mind. Firstly, integration tests could be used instead of end-to-end tests. The integration tests should only test one system, being the terminal, with no varying external factors, such as a live back-end or a separate database. This way, failed tests will always be caused by the terminal, providing an ideal evaluation environment for GitFL. Secondly, the environment could be changed, moving away from the robot testing setup. For example, by applying GitFL to the integration tests of a back-end repository. This code runs on servers, removing the factor of the software running on embedded systems, which is arguably a more complex system and is thus more time-consuming to test due to longer build and run times [3].

B. Algorithm assumptions

The version control history part of GitFL only considers changes inside the $DBIT$ range and discounts changes based on age. These characteristics lead to the possibility of several unwanted scenarios. For example, suppose a bug is not fixed shortly after it is discovered, which can occur when fixing it takes a significant amount of time. The longer it is left unfixed, the lower its suspiciousness will be. The worst case is when the bug is left unfixed while advancing to the next release, the bug will be out of scope and receives a suspiciousness of zero for the VCH part of GitFL. While this bug could be the root cause of many test failures, it is not deemed as highly suspicious by GitFL because of its age, potentially resulting in bad performance of GitFL. A potential solution could be to mark these long-lasting bugs as highly suspicious, without the need to immediately fix them. When running GitFl the next time, these marked bugs are included in the calculation, preserving the high suspiciousness of those bugs.

The results showed that a lower value of d_f resulted in better performance. It can be reasoned that this result was expected, as a lower decay factor always raises the y-values of an exponential decay function. With this observation, it can be argued that a constant function for VCH performs similar to an exponential function with a low decay factor, eliminating the need for an exponential decay function. However, the exponential decay function was introduced in the design with a specific situation in mind, namely, when there are multiple changes within the time window which have received the same FL suspiciousness score. Usage of an exponential decay function prevents these statements to be tied, as the suspiciousness values are affected by the date. This is under the assumption that bugs are fixed shortly after discovery, meaning that the most recent change is more likely to hold the buggy statement. This specific situation did not occur in the evaluation, which potentially implies that it does not occur often, eliminating the need for an exponential decay function.

Suppose that a hidden bug (e.g. a bug which does not cause failure at the moment) was introduced in the previous release and was revealed by another change in the current release. GitFL will assign a high suspiciousness to the change revealing the bug, but not to the bug itself. Since the

change revealing the bug can impact the execution of many other non-buggy statements, the bug cannot necessarily be easily deducted from the revealing change. The bug could also be non-hidden, causing tests to fail, indicating that the previous release was not necessarily free of test failures. Increasing the days-back-in-time parameter (DBIT) for GitFL can result in better performance for hidden bugs and scenarios where the assumption that the previous release is free of failing tests does not hold. This way, the influence of significantly older bugs on the suspiciousness is very low but never zero. However, the consequence of this increase is that more changes are included, potentially drowning out other bug-inducing changes.

Within the evaluation, it became apparent that the lowest value of the decay factor always resulted in the best performance. It can be reasoned that this result was obvious, as a smaller decay factor inherently raises the suspiciousness value of changes, due to the working of the exponential decay function. When they decay factor is lowered infinitely, the function attains the constant value of one. This renders the use of an exponential decay function obsolete, as it can be replaced by an equal suspiciousness value for all changes within the range. One could argue that this result can be generalized to all environments, instead of just the specific evaluation scope of this paper. However, an edge case exists. When multiple statements obtain the same suspiciousness value by FL and are both changed recently, it can be unclear which statement is at fault. With an exponential decay function, the statements can be differentiated, with the most recent change attaining a higher final suspiciousness. This assumes that the assumption that bugs are fixed shortly after discovery still holds. The exception to this, is when multiple statements obtain the same suspiciousness value by FL and are both changed at the exact same time. In this scenario, differentiation is impossible, resulting in a tied suspiciousness value.

C. Tooling

During the evaluation of GitFL, it became apparent that the selected coverage information collecting setup had a significant shortcoming. Namely, it did not log which lines were executed in lambda functions. If a lambda function was executed, only the first and last line were logged. Whether this was caused by the coverage information tool Gcov, the compiler or another factor was unclear. It is important to realize that the combination of FL with version control history is flawed with lambda functions. If a developer changes statements in a lambda function but leaves the first and last line of the function intact, the traditional FL and version control history for this change can never be combined, as FL only registers the execution of the first and last line of the lambda function.

The requirement of using placeholders for deleted statements was essential for the evaluation of GitFL. Without placeholders, even though it was known beforehand which statements were deleted, these could never be located by GitFL because they were never executed, making it impossible to determine the performance. However, outside of evaluation, GitFL does not necessarily require to locate the exact deleted statements. With deleted statements, GitFL should locate suspicious statements which are closely connected to the deleted statements. For example, if the input of a method was originally provided by now deleted statements, the call to this method will be marked as highly suspicious by GitFL. The developer can use this method as starting point and investigate to which statements it is closely connected. A disadvantage of this approach is that it increases developer effort. Another disadvantage of not being able to exactly locate deleted statements is that the statements returned by GitFL, closely connected to the deletion, is not expected to be changed in the same change as the deletion. Thereby, it is not possible to automatically determine which author caused the bug.

D. Evaluation hurdles

While the input data for GitFL was assumed to be available within this paper, obtaining it was a lengthy process which took around three to four months. To obtain the data, a robust pipeline needed to be built to ensure continuous data collection during testing. All exceptions needed to be handled correctly to prevent the process of crashing, potentially resulting in invalid input data, thereby influencing the evaluation. At the start of the project, the evaluation environment, being Adyen terminal testing, was considered a good fit. However, as described earlier, this introduced several hurdles. The most significant one being to build a robust pipeline to gather coverage information from the terminal, which posed a great amount of challenges, as this process appeared to be more complex for embedded systems compared to server-like systems. Additionally, the pipeline was required to be integrated with the robot testing framework in place to enable the collection of coverage information during testing.

Furthermore, the Adyen environment turned out to provide very limited input data. In theory, there should be many occasions of real-life bugs causing robot tests to fail, over the course of months. However, a large percentage of failed robot tests was not caused by the terminal software itself, but by external systems interacting with the terminal. As those test failures were not caused by the terminal software, they could not be used for evaluating GitFL. This made it difficult to obtain a real-life test suite of a reasonable size. To overcome this problem and to still be able to evaluate GitFL, bugs were manually introduced in the terminal software. This was done by mutating code to create bugs, with the purpose of making tests fail. While still evaluating on a limited amount of real-life data, this larger artificial test suite provided more data for a

substantiated evaluation.

IX. Threats to validity

The validity of the evaluation of GitFL is limited by several threats. These, together with a potential solution, are listed below.

- GitFL was evaluated on the Adyen case, where only a subsystem was tested. Arguably, this is a specific environment. Because of this, it is difficult to determine how GitFL will generalize to other environments. To determine its generalizability, it is recommended that GitFL is tested on multiple varying (open source) environments.

- Both the artificial and real-life test suites were single-fault test suites, meaning they contain exactly one bug. This means that GitFL is not evaluated on multiple-bug scenarios (where multiple bugs are present in the test suite) and that it is unclear how the performance of GitFL will generalize to multiple-bug scenarios. In the fault localization literature, several authors indicate that single-bug evaluations can be generalized to multiple-bug scenarios [34, 22, 64, 65], although it is unclear what the exact performance difference is. To determine multiple-bug performance, it is recommended that GitFL is tested on multiple-bug scenarios.

- Compared to the test suites used in the FL literature, the test suite for the Adyen case is limited in size. Logically, it could be reasoned that a larger test suite should be likely to provide more accurate results, as with a larger number of tests, more bug-free statements can be labeled as not-suspicious, drawing a contrast to the suspicious statements. However, Lei et al. [12] argue with their study that there is no strong correlation between test suite size and the effectiveness of fault localization. Testing GitFL on a larger test suite could clarify this.

- The limited time of this project influenced the extent to which GitFL parameters were found that lead to optimal performance. Because of this, a limited set of parameter values was chosen. A more extensive study with a larger set of parameter values could result in finding parameters that cause better performance.

- In the evaluation, Tarantula was used as the standard fault localization technique. Yet, GitFL accepts all fault localization techniques. Although Tarantula was considered to be a fitting representation of most used FL methods, it is unclear how GitFL performs with other FL methods. To determine this, it is recommended that GitFL is tested on several other FL methods.

- When a test suite caused the system to crash, coverage information could not be retrieved. Coverage information is required for GitFL, rendering GitFL, in this environment, unable to produce results when crashes happen.

- For the artificial test suite, the change date of changes introducing bugs was sampled from an exponential decaying distribution, with the goal of simulating the assumption that bugs are fixed shortly after they are discovered. This means that the likelihood of older bugs existing would be smaller. It remains unclear whether this assumption resembles reality. To clarify this matter, research can be conducted (i.e. on open source projects) to obtain a more substantiated change date distribution.

X. Conclusion

When batches of code changes from multiple authors are tested at once, it can be unclear which changes from which authors cause test failures. To solve this problem, we introduced GitFL, an Algorithm that combines state-of-the-art FL with version control history information. GitFL was implemented and evaluated at Adyen, where both real-life and artificial test suites were used. We found that GitFL performed significantly better compared to state-of-the-art FL methods within the specific low-coverage test suite GitFL was evaluated on. Frequent end-to-end robot testing can belong to this low-coverage scenario, with its often-limited testing capacity. This means that GitFL can aid in FL for end-to-end robot testing environments.

A deployed FL system outputs suspicious lines. To be of any use, these suspicious lines need to be communicated to developers in such a way that they are probable to consider them during debugging. Additionally, they need to trust the algorithm to provide helpful insights. Possible implementations could, for example, consist of an IDE plugin, where suspicious lines automatically highlighted, enabling developers to observe them with a quick glance. Another implementation could be an integration in the version control system. Here, the top-N suspicious lines for a particular code-change could be automatically commented. This does not require downloading a plugin and therefore has an arguably lower boundary for usage for developers, for which using the version control system is standard practice. This aspect of the design of GitFL is not in the scope of this project. To design an effective method of communicating algorithm results with developers, substantiated by literature from the human computer interaction field, further research on this topic is required.

XI. References

[1] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[2] Ray Noel Medina Delda et al. "3D Printing Polymeric Materials for Robots with Embedded Systems". In: *Technologies* 9.4 (2021), p. 82.

[3] Mauricio Aniche. *Effective Software Testing: A Developer's Guide*. Simon and Schuster, 2022.

[4] Zeba Khanam and Mohammed Najeeb Ahsan. "Evaluating the effectiveness of test driven development: advantages and pitfalls". In: *International Journal of Applied Engineering Research* 12.18 (2017), pp. 7705–7716.

[5] Jim Shore. "Fail fast [software debugging]". In: *IEEE Software* 21.5 (2004), pp. 21–25.

[6] James A Jones and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique". In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 2005, pp. 273–282.

[7] W Eric Wong et al. "The DStar method for effective software fault localization". In: *IEEE Transactions on Reliability* 63.1 (2013), pp. 290–308.

[8] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. "An evaluation of similarity coefficients for software fault localization". In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE. 2006, pp. 39–46.

[9] Ben Liblit et al. "Scalable statistical bug isolation". In: *Acm Sigplan Notices* 40.6 (2005), pp. 15–26.

[10] W Eric Wong et al. "A survey on software fault localization". In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[11] Sangharatna Godboley and Arpita Dutta. "PRFL: Predicate Rank based Fault Localization". In: *2021 IEEE 18th India Council International Conference (INDICON)*. IEEE. 2021, pp. 1–6.

[12] Yan Lei et al. "How test suites impact fault localisation starting from the size". In: *IET software* 12.3 (2018), pp. 190–205.

[13] Ali Koc and Abdullah Uz Tansel. "A survey of version control systems". In: *ICEME 2011* (2011).

[14] James A Jones, Mary Jean Harrold, and John T Stasko. "Visualization for fault localization". In: *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer. 2001.

[15] Daming Zou et al. "An empirical study of fault localization families and their combinations". In: *IEEE Transactions on Software Engineering* 47.2 (2019), pp. 332–347.

[16] Thomas Reps et al. "The use of program profiling for software maintenance with applications to the year 2000 problem". In: *Software Engineering—Esec/Fse'97*. Springer, 1997, pp. 432–449.

[17] Trishul M Chilimbi et al. "Holmes: Effective statistical debugging via efficient path profiling". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 34–44.

[18] Chao Liu et al. "Statistical debugging: A hypothesis testing-based approach". In: *IEEE Transactions on software engineering* 32.10 (2006), pp. 831–848.

[19] Brock Pytlik et al. "Automated fault localization using potential invariants". In: *arXiv preprint cs/0310040* (2003).

[20] Mary Jean Harrold et al. "An empirical investigation of program spectra". In: *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 1998, pp. 83–90.

[21] Raul Santelices et al. "Lightweight fault-localization using multiple coverage types". In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 56–66.

[22] Ming Wen et al. "Historical spectrum based fault localization". In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2348–2368.

[23] Nayan B Ruparelia. "The history of version control". In: *ACM SIGSOFT Software Engineering Notes* 35.1 (2010), pp. 5–9.

[24] *Git*. URL: https://git-scm.com/.

[25] *CVS - Open Source Version Control*. URL: https://www.nongnu.org/cvs/.

[26] *BitKeeper*. URL: https://www.bitkeeper.org/.

[27] Mark Weiser. "Programmers use slices when debugging". In: *Communications of the ACM* 25.7 (1982), pp. 446–452.

[28] Frank Tip et al. *Generation of program analysis tools*. Universiteit van Amsterdam, Inst. for Logic, Language and Computation, 1995.

[29] Mark Weiser. "Program slicing". In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357.

[30] Bogdan Korel and Janusz Laski. "Dynamic program slicing". In: *Information processing letters* 29.3 (1988), pp. 155–163.

[31] Hiralal Agrawal et al. "Fault localization using execution slices and dataflow tests". In: *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE. 1995, pp. 143–151.

[32] Manos Renieres and Steven P Reiss. "Fault localization with nearest neighbor queries". In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE. 2003, pp. 30–39.

[33] Xiaoyuan Xie et al. "Metamorphic slice: An application in spectrum-based fault localization". In: *Information and Software Technology* 55.5 (2013), pp. 866–879.

[34] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. "A model for spectra-based software diagnosis". In: *ACM Transactions on software engineering and methodology (TOSEM)* 20.3 (2011), pp. 1–32.

[35] W Eric Wong, Vidroha Debroy, and Byoungju Choi. "A family of code coverage-based heuristics for effective fault localization". In: *Journal of Systems and Software* 83.2 (2010), pp. 188–208.

[36] Qusay Idrees Sarhan and Árpád Beszédes. "A Survey of Challenges in Spectrum-Based Software Fault Localization". In: *IEEE Access* 10 (2022), pp. 10618–10639.

[37] W Eric Wong and Yu Qi. "BP neural network-based effective fault localization". In: *International Journal of Software Engineering and Knowledge Engineering* 19.04 (2009), pp. 573–597.

[38] Wei Zheng, Desheng Hu, and Jing Wang. "Fault localization analysis based on deep neural network". In: *Mathematical Problems in Engineering* 2016 (2016).

[39] Zhuo Zhang et al. "A study of effectiveness of deep learning in locating real faults". In: *Information and Software Technology* 131 (2021), p. 106486.

[40] W Eric Wong et al. "Effective software fault localization using an RBF neural network". In: *IEEE Transactions on Reliability* 61.1 (2011), pp. 149–169.

[41] Lingxiao Jiang and Zhendong Su. *Automatic isolation of cause-effect chains with machine learning*. Tech. rep. Citeseer, 2005.

[42] Seokhyeon Moon et al. "Ask the mutants: Mutating faulty programs for fault localization". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 153–162.

[43] Mike Papadakis and Yves Le Traon. "Metallaxis-FL: mutation-based fault localization". In: *Software Testing, Verification and Reliability* 25.5-7 (2015), pp. 605–628.

[44] David Abramson et al. "Relative debugging and its application to the development of large numerical models". In: *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. IEEE. 1995, pp. 51–51.

[45] Andreas Zeller and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input". In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.

[46] Holger Cleve and Andreas Zeller. "Locating causes of program failures". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE. 2005, pp. 342–351.

[47] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. "Locating faults through automated predicate switching". In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 272–281.

[48] W Eric Wong, Vidroha Debroy, and Dianxiang Xu. "Towards better fault localization: A crosstab-based statistical approach". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.3 (2011), pp. 378–396.

[49] Jiajun Jiang et al. "Combining spectrum-based fault localization and statistical debugging: An empirical study". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 502–514.

[50] Wolfgang Mayer and Markus Stumptner. "Evaluating models for model-based debugging". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2008, pp. 128–137.

[51] Wolfgang Mayer et al. "Prioritising model-based debugging diagnostic reports". In: *Proceedings of the 19th International Workshop on Principles of Diagnosis*. Citeseer. 2008, pp. 127–134.

[52] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. "Spectrum-based multiple fault localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2009, pp. 88–99.

[53] Tom Janssen, Rui Abreu, and Arjan JC Van Gemund. "Zoltar: A toolset for automatic fault localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2009, pp. 662–664.

[54] Free Software Foundation, Inc. *Gcov (Using the GNU Compiler Collection (GCC))*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[55] Sandia Corporation. *gcovr*. URL: https://gcovr.com.

[56] *Jcov*. URL: https://github.com/openjdk/jcov.

[57] *cov - Rust*. URL: https://docs.rs/cov/latest/cov/.

[58] *cover command - cmd/cover - Go Packages*. URL: https://pkg.go.dev/cmd/cover.

[59] *Istanbul, a JavaScript test coverage tool.* URL: https://istanbul.js.org/.

[60] *Git blame*. URL: https://git-scm.com/docs/git-blame.

[61] Henry Coles et al. "Pit: a practical mutation testing tool for java". In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 449–452.

[62] Pavneet Singh Kochhar et al. "Practitioners' expectations on automated fault localization". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 165–176.

[63] Xin Xia et al. ""Automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 267–278.

[64] Nicholas DiGiuseppe and James A Jones. "On the influence of multiple faults on coverage-based fault localization". In: *Proceedings of the 2011 international symposium on software testing and analysis*. 2011, pp. 210–220.

[65] Spencer Pearson et al. “Evaluating & improving fault localization techniques”. In: *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03* (2016).

XII. Appendix

A. Examples of code mutations

To illustrate how mutations can cause tests to fail, an example is provided. Consider the code in table XIII, which resembles several methods from a very basic financial API for processing transactions. The first method, `check_if_zero_amount`, checks if a transaction amount is zero and returns true if that is the case. Otherwise, it returns false. The second method, `assign_amount_to_json_field`, assigns an amount value to a json field to be used later on. The third and final method, `add_amount_to_response`, adds the json object amount to the json object response, which is later on used as a response for a specific API call.

To purposely make the code fail, mutations are introduced. In the example, the first three mutations from table X are applied. Negate Condition is applied to the first method, `check_if_zero_amount`. The comparison operator `==` from line 2 in table XIII is switched to `!=`, resulting in the code in table XV. When an amount is actually zero, the method will now return false when checking for a zero amount. Other methods depending on this check will fail. For example, a method that stops the transaction if the amount is zero will not be executed correctly, resulting in the wrong behavior later on, resulting in tests to fail. The second mutation operator, Return Values, replaces `return true` with `return false` from line 3 in table XIII, resulting in the code in table XVI. This leads to the same behavior described for the first mutation operator.

The third mutation operator, Method Call, is applied to `assign_amount_to_json_field`. This operator completely deletes that method, line 8 to 10 from table XIII, resulting in the code in table XIV. In the resulting code, the double amount is never converted to a json object. The method `dd_amount_to_response` does not work properly, because it expects a non-existent json object. When called, the method will fail and tests will fail.

TABLE XIII: Several methods from a sample financial API, for showing mutation examples.

Line	Code
1	`bool check_if_zero_amount(double amount) {`
2	`  if (amount == 0) {`
3	`    return true;`
4	`  }`
5	`  return false;`
6	`}`
7	
8	`json_object assign_amount_to_json_field(double amount) {`
9	`  return new json_object amount_json = to_json_object(amount)`
10	`}`
11	
12	`bool add_amount_to_response(json_object amount_json) {`
13	`  response.add(amount_json)`
14	`}`

TABLE XIV: Resulting code after the Negate Condition mutation from table X is applied

Line	Code
1	`bool check_if_zero_amount(double amount) {`
2	`  if (amount == 0) {`
3	`    return true;`
4	`  }`
5	`  return false;`
6	`}`
7	
8	`bool add_amount_to_response(json_object amount_json) {`
9	`  response.add(amount_json)`
10	`}`

TABLE XV: Resulting code after the Return Values mutation from table X is applied

Line	Code
1	`bool check_if_zero_amount(double amount) {`
2	`  if (amount != 0) {`
3	`    return true;`
4	`  }`
5	`  return false;`

TABLE XVI: Resulting code after the Method Call mutation from table X is applied

Line	Code
1	`bool check_if_zero_amount(double amount) {`
2	`  if (amount == 0) {`
3	`    return false;`
4	`  }`
5	`  return false;`