Testing Software Systems Against Realistic User Errors

THÈSE N^{O} 6009 (2014)

PRÉSENTÉE LE 31 JANVIER 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DES SYSTEMES FIABLES

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Silviu ANDRICA

acceptée sur proposition du jury:

Prof. A. Wegmann, président du jury
Prof. G. Candea, directeur de thèse
Dr E. Cecchet, rapporteur
Prof. T. Nguyen, rapporteur
Dr P. Pu Faltings, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

Abstract

Le persone organizzano sempre di più ogni aspetto della propria vita attraverso l'utilizzo di sistemi software, dal lavoro al trasporto alla salute. Possiamo essere sicuri che i programmi dai quali ci accingiamo a dipendere sono affidabili ? Tali programmi possono tollerare maltrattamenti degli utenti?

Questa tesi affronta il problema di verificare i sistemi software rispetto ad errori introdotti dagli utenti. Gli errori introdotti dagli utenti sono inevitabili. Ciò significa che gli sviluppatori non possono vedere gli utenti come esseri perfettamente razionali che scelgono sempre la migliore linea di azione e non commettono mai errori. Perciò, i sistemi software devono poter essere robusti ad errori umani.

In questa tesi si propone una tecnica che verifica i sistemi software rispetto un ampio spettro di comportamenti erronei umani realistici. Tale tecnica permette agli sviluppatori di quantificare l'impatto che gli errori degli utenti hanno sull'affidabilità del loro sistema, e.g. crash di sistema, cos che possano sapere dove e' meglio agire per migliorare il loro sistema.

Le tecniche per rendere i sistemi software robusti ad errori umani sono tre: 1) prevenzione, 2) "coping", 3) verifica e validazione. Le tecniche di prevenzione permettono di allontanare gli errori degli utenti attraverso l'identificazione e la rimozione di opportunit per gli utenti di compiere errori. Purtroppo, queste tecniche richiedono manodopera esperta che le rende molto costose e sono tuttavia limitate alla protezione dei sistemi solo rispetto ad errori conosciuti in anticipo. Le tecniche di "coping" assicurano la possibilità di mitigare o invertire gli effetti di errori umani. Queste tecniche hanno tuttavia lo svantaggio di incrementare la complessità del sistema, che lo rende complesso da gestire e permette l'insinuazione di difetti al suo interno. Infine, gli sviluppatori usano tecniche di verifica e validazione per mettere in evidenza deficienze nell'implementazione del sistema. Tali problemi possono essere corretti applicando una delle tecniche precedentemente menzionate.

Questa tesi presenta una tecnica per verificare i sistemi software rispetto ad errori introdotti dagli utenti che è contemporaneamente automatica e collaborativa. Attraverso l'automazione del processo di verifica del sistema, in questa tesi si mostreranno i benefici dovuti alla riduzione del lavoro manuale richiesto e del numero di test eseguiti. Con il termine "realistico" s'intende che il comportamento erroneo dell'utente è rappresentativo del comportamento tipicamente esercitato dagli utenti che interagiscono con il sistema. Con il termine "rilevante" s'intende che l'attività di verifica è guidata verso le parti del sistema sulle quali gli utenti si affidano di più. Con il termine "collaborativo" s'intende che gli utenti del "mondo reale" possono partecipare al processo di verifica, o come sorgente dei test o come ospitanti dell'esecuzione del test.

La tecnica proposta è stata applicata ad applicazioni web, applicazioni smartphone, e a server web. In particolare, tale tecnica è stata in grado di generare test che rilevano difetti non scoperti in ogni categoria di sistemi analizzati. Tali test mostrano che gli errori degli utenti causano l'esecuzione di nuovo codice, il quale contiene difetti. Inoltre la tecnica proposta riesce a preservare l'anonimia degli utenti che partecipano nella collaborazione e non ostacola

l'esperienza degli utenti durante il collezionamento dei dati.

Parole chiave: sistema, errori umani, crowdsourcing, applicazioni web, applicazioni smartphone, server web, anonimia degli utenti.

Abstract

Increasingly more people manage every aspect of their lives using software systems, from work to transportation and health. Can we be sure that the programs on which we came to depend are dependable? Can they handle users mistreating them?

The problem we address in this thesis is that of testing software systems against user errors. User errors are inevitable. This means that developers cannot view users as perfectly rational beings who always choose the best course of action and never make a mistake. Therefore, software systems must be resilient to them.

In this thesis, we develop a technique that tests software systems against a wide spectrum of realistic erroneous user behavior. The technique enables developers to quantify the impact user errors have on their system's dependability, e.g., the system crashes, so that they know where to improve their system.

There are three classes of techniques to make systems resilient to user errors: 1) prevention, 2) coping, and 3) testing and verification. Prevention techniques keep user errors away from the system by identifying opportunities for users to make mistakes and removing them. Alas, these techniques require expert human labor, making them expensive, and are limited to protecting systems only against anticipated errors. Coping techniques ensure that if a user makes a mistake, its negative effects can be reversed or mitigated. These techniques have the drawback that they increase the complexity of the system, making them difficult to manage and enabling bugs to creep in. Finally, developers use testing and verification techniques to point out deficiencies in their system's implementation, which they then fix by applying one of the previous techniques.

This thesis presents a technique for testing software systems against user errors that is automated, realistic, relevant, and collaborative. By automating the testing process, we reduce the amount of manual labor and reduce the number of tests required to run against the system. By "realistic" we mean that the erroneous user behavior is representative of the behavior actual users exert when interacting with the system. By "relevant" we mean that testing is guided toward the parts of the system that users rely on most. By "collaborative" we mean that real-world users can participate in the testing process, either as sources of tests or as hosts for test execution.

We applied our technique to web applications, smartphone applications, and web servers. The tests the technique generates uncovered bugs in each category of systems and show that user errors cause new code to execute, which can contain bugs. The technique manages to preserve the anonymity of contributing users and collecting usage scenarios does not hinder user experience.

Keywords: testing, human error, crowdsourcing, systems, user anonymity, web applications, smartphone applications, web servers.

Acknowledgments

I would like to thank my advisor, George Candea, who always had faith in me, and was more than just an advisor.

I would like to thank my parents, Mariana and Liviu Andrica, who set aside the pain caused by having their son leave the home country so that he could make a future for himself.

I owe my gratitude to Dr. Pearl Pu, Dr. Emmanuel Cecchet, and Dr. Thu Nguyen who accepted to be members of my PhD defense committee, and to Dr. Alain Wegmann who served as the jury president for my PhD defense.

I wish to thank my DSLab colleagues, who cheered me up when things looked grim and always had great feedback.

I want to thank Nicoletta Isaac, DSLab's administrative assistant, for helping me navigate through the intricacies of EPFL bureaucracy.

I would like to thank my friends Roxana, Victor, Iulian, and those who I shamefully forgot to mention for cheering me up and having faith in me.

I wish to express my gratitude toward the PocketCampus guys, Amer, Loïc, and Florian, who provided me with an evaluation target and made me feel part of something important.

I want to thank Francesco Fucci who helped me translate the thesis abstract into Italian.

Contents

Chapter 1

Introduction

Software systems permeate every aspect of the lives of people living in developed or developing societies. People use software systems to read news, send emails, get work done, or have fun by keeping in touch with friends on online social networks. People use programs to manage even sensitive aspects of their lives, such as health or finance.

One's willingness to surrender control over one's life to a software system suggests one expects that that system is correct. One has confidence that nothing bad can happen. When a user makes a mistake, he is sure that somebody anticipated it and made sure that the system correctly handles each exceptional situation. This is a property of appliances—they just work—and users have the same expectation of software systems.

This thesis presents a technique that enables developers to assess the effects that a wide spectrum of user errors have on the dependability of their system. Armed with this information, developers can decide how to best protect their system and its users.

1.1 Problem Statement

The problem we address in this thesis is that of developers being unable to ensure that the software system they are building is resilient to erroneous user behavior. We say that a system is resilient to user errors if they cannot hinder the system's dependability. Dependability is that property of a system which allows reliance to be justifiable placed on the service it delivers. Dependability has four properties: availability, reliability, safety, and security [1].

For example, a user of an e-banking application should be able to transfer money (availability), and tapping the wrong button in the application should not cause the application to crash (reliability), transfer all the user's funds away (safety), or make the user's account publicly accessible (security).

Tolerating user errors is an important system property. Studies show that user error is pervasive, e.g., Wood [2] reports that each participant in an experiment committed on average 5 errors, so every system will encounter them once deployed. If a system is not resilient to user errors, users stop using it or, worse, it can cause irreparable damage to users. For example, would you use GMail if every time you type in a wrong email address, the application crashed?

People use software systems to complete a task, e.g., write an email. Their interaction with the system consists of issuing a sequence of commands to the system. User errors are commands that stop users from progressing toward task completion, e.g., inputting the wrong email address to send an email to.

To avoid their system becoming irrelevant or a liability, developers need to build into it defense mechanisms against user errors. The first step is to identify how user errors manifest. Alas, it is impossible to manually complete this task.

The reason is that user errors can occur every time a user interacts with a system. The number of ways in which users can interact with the system is proportional to the number of paths through its code, which is exponential in its size or infinite, if the program has an unbounded loop.

In this thesis, we present a technique and tools that make explicit the realistic user errors a system will encounter during its lifetime. By focusing on realism, our technique achieves practicality, because it reduces number of program executions and user errors developers need to test their system against.

Section 1.2 shows the effects user errors had on a system and its users, while Section 1.3 briefly reviews techniques to make systems resilient to user errors.

1.2 Impact of User Errors

We show that user errors are a threat to systems' dependability. We review five examples of user errors and their consequences on a system, its users, or people affected by actions taken based on the system's output. The first two examples show that user errors can manifest as valid program input, not cause the system to enter an abnormal state, but cause it to perform an unintended action.

Recently, the administrator of EPFL's email server wrongfully edited the server's configuration, causing email meant for one email list to be delivered to another, wrong list. At EPFL, email lists can be moderated, i.e., an administrator must approve an email sent to the list before it is delivered to its members. EPFL hosts an IEEE public email list, and its owners asked that moderation be removed for emails coming from a particular email address, i.e., if sent from this address, an email will be immediately delivered. This prompted the system administrator to add an exception to the moderation rules. Later on, another email list was created that also required email delivery without moderation for email coming from a particular address. To accommodate this requirement, the email server administrator had to add a second moderation exception, similar to the one for the IEEE list, to the email server's configuration. The system administrator used the line containing the exception for the IEEE list as a template, but forgot to change the name of the list to deliver email to, keeping the IEEE public one [3]. This resulted in emails sent to the list belonging to an EPFL class be delivered to members of the IEEE public list.

Another example of user error is that of a Wall Street stock trading system clerk who mistakenly entered the correct number in the wrong text field [4]. This resulted in an abnormally large transaction that drove automated trading systems to start selling shares, causing the New York Stock Exchange to lose all gains made earlier in the day.

The next example describes consequences of allowing erroneous user input to enter a system. In 2010, Facebook went down for two and a half hours [5]. The root cause was manually setting a configuration parameter in a database to an invalid value. This value was propagated to database clients. An automated system for verifying configuration values detected the value as being invalid and instructed the clients to reread it from the database. This lead to a flood of requests to the database. But, the database contained the invalid value, so requests continued to accumulate. To stop the flood, the database had to be taken offline, and Facebook became unavailable.

The fourth example shows user errors causing bugs in a system to surface. The Therac-25 [6] was a radiation therapy machine with two operation modes: one in which it emits a beam of high power particles that hits a thick metal plate to produce an x-ray beam, and a lower power electron beam mode, in which the beam is not obstructed by the metal plate. Alas, the machine had a data race that caused it to irradiate patients with lethal doses of radiation [6]. The data race was triggered when the machine operator mistakenly selected the x-ray operation mode, which used the machine's entire power, but then corrected the operation mode, setting it to use the lower power electron beam mode, all within 8 seconds. This caused the machine to maintain the high power mode, specific to the x-ray mode, but

remove the metal plate, specific to the lower power mode. This resulted in radiation poisoning in patients, which lead to their deaths.

The fifth example shows how user errors can lead to violating people's privacy. In 2010, the UK Security Service (MI5) wrongfully acquired subscriber data in relation to 134 incorrect telephone numbers. "These errors were caused by a formatting fault on an electronic spreadsheet which altered the last three digits of each of the telephone numbers to '000'." [7].

We expect user errors to continue having potentially devastating effects on system dependability.

1.3 Summary of Previous Approaches

There are three main techniques to make systems resilient to user errors: 1) prevention techniques, 2) coping techniques, and 3) testing and verification techniques.

1.3.1 Prevention Techniques

Prevention techniques keep user errors outside the system. There exist four ways to achieve this [8]: train users, so that they do not make mistakes; remove opportunity for errors through automation or designing user interfaces that are aware of user errors; and delay execution of user commands, so that if a user commits a mistake, they have time to fix it. These techniques require extensive manual labor, therefore they are expensive. For example, eliminating the possibility for user errors to manifest requires an iterative refinement process in which an expert fleshes out the interaction protocol between the user and the system, identifies where user errors can occur, and then improves the protocol to make errors less likely. Another drawback of prevention techniques is that they are limited in the scope of the errors they prevent. Developers cannot think of all the errors users can make, and thus, cannot set in place mechanisms to guard the system against them. An in-depth review of related work on this topic is presented in Section 2.2.

1.3.2 Techniques for Coping with User Error

Coping techniques ensure that, if a user makes a mistake and the system already processed the erroneous user command, the negative effects of that mistake can be reversed or mitigated [8]. A first coping technique is spatial replication, where there exist multiple replicas of a system and the user interacts with a single replica, with the other replicas acting as backups. Thus, if the user makes a mistake, its effects can be removed by setting the state of the affected replica to that of the other replicas. A second coping technique is temporal replication, where a history of a system's state is used to rollback the system to a state previous to the user error, so its effects disappear. Another technique is temporal replication with reexecution, where the state of the system is rolled back to a prior state and the actions the user took since that state are replayed, to bring the system up to date. Coping techniques have the drawback of requiring extra resources, e.g., additional hardware to host the replicas of a system, are difficult to manage, or are difficult to implement correctly [8]. An in-depth review of related work on this topic is presented in Section 2.3.

1.3.3 Testing Techniques

The previous two classes of techniques offer solutions to handling user error. However, their implementations might be incorrect or incomplete and not cover all possible user errors. Thus, developers need to assess their quality by using

testing and verification techniques. These techniques act as a feedback mechanism for developers, who use them to identify deficiencies, then fix them, and then recommence the cycle.

Testing techniques concern themselves with assessing the system's correctness with respect to some usage scenarios. Verification techniques aim to prove that the system is correct in all situations, which subsumes its ability to correctly handle user errors, assuming one can provide a specification of what it means to be resilient to user errors. On the one hand, testing techniques cover only certain aspects of a system's functionality, but can be automated and can be used on systems of all sizes. On the other hand, verification techniques need correctness properties to be expressed in formal languages, which is laborious and error prone, and cannot yet handle large bodies of code, but if they succeed, they provide a proof that the system is correct. An in-depth review of related work on this topic is presented in Section 2.4.

1.4 Properties of the Ideal Solution

The techniques for preventing or coping with user error are expensive to employ, making them inaccessible to all developer teams. Prevention techniques require extensive manual effort, while coping techniques require extensive resources.

Instead, testing is easy and cheap to employ, so we choose to pursue this avenue. We believe that a solution to testing software systems against user errors must have the following four properties:

- *Accuracy*. An accurate technique is realistic, relevant, effective, and efficient.
  - *Realism*. It is impractical to test a system against every input, because that is equivalent to verification. Thus, to be practical, the solution needs to focus on a subset of important inputs. In this thesis, we focus on testing a system's resilience to user errors, i.e., against those inputs that have been affected by user errors. Alas, such testing is equivalent to testing all a system's inputs, because one can argue that any input can be the consequence of user errors. Instead, we limit testing against those user errors that are representative of the erroneous behavior of real-world users. Realism is what makes the solution practical, because it limits the scope of testing.
  - *Relevance*. The technique should ensure that the tests it generates focus on system components users most rely on.
  - *Effective*. The technique should be able to discover bugs triggered by user errors.
  - *Efficiency*. The technique should use the minimum number of tests to achieve maximum effectiveness.
- *Automation*. Employing the technique and its associated tools should require minimal developer effort. There exist two dimensions to this:
  - *Automated test generation*. Developers should not have to specify tests.
  - *Automated test (re)execution*. Developers should be able to automatically run tests. For this to work reliably, the technique must generate reproducible tests.
- *Scalability*. The technique should scale along two dimensions:
  - *System size*. The technique should be able to test software systems that are comprised of millions of lines of code.

  - *Portability*. A portable testing technique is able to express user errors independently from the system it is testing, making the technique applicable to a large set of systems.

- *Protect users' anonymity*. If the technique leverages real users' interactions with a system to generate tests or guide the testing effort, then one should not be able to determine based on the generated tests who is the user who contributed the interaction seed.

1.5 Our Proposed Solution

This thesis presents Record-Mutate-Test, a relevant, portable, efficient, crowdsourced, and automated technique for testing software systems against realistic user errors while maintaining users' anonymity.

Record-Mutate-Test has three steps. In the first step, *Record*, it records the interaction between *real* users and the system under test in an interaction trace.

Next, during the *Mutate* step, the technique applies a set of user error operators on the interaction trace and generates new interaction traces. These are tests that simulate how real-world users interact with a system. The errors it injects are inspired by observations of user errors that arose from studies performed by human error specialists.

During the final step, *Test*, the technique replays the mutated interaction traces against the system under test and reports the discovered bugs to the system's developers.

Record-Mutate-Test is collaborative in that it enables users to contribute usage scenarios for test generation or donate their machine for test execution. Prior to a user contributing an interaction trace, Record-Mutate-Test ensures that the trace cannot identify its source user.

1.6 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 reviews prior work related to defining user errors, classifying them, protecting systems against them, handling user errors when they cannot be prevented, and testing techniques that uncover bugs in handling user errors.

Next, Chapter 3 describes in details the technique used to test systems against realistic user errors that is the subject of this thesis. Chapter 4 describes how the technique enables users to submit usage scenarios, to be used as seeds for tests, without violating the users' anonymity. Chapter 5 presents a new code coverage metric that developers can use to make best use of their testing resources.

Chapter 6 describes four tools that are embodiments of Record-Mutate-Test. One tool targets web applications, a second one focuses on smartphone applications, while the other two tools target system administrator misconfiguration. Chapter 7 shows that Record-Mutate-Test possesses the properties of the ideal testing technique.

Finally, Chapter 8 concludes this thesis.

Chapter 2

Prior Work

In this chapter, we review prior work on defining user errors, classifying them, protecting software systems against user errors, handling them when they cannot be prevented, testing techniques that ensure that the handling mechanisms function correctly, and techniques to protect users' anonymity. Each category of related work provides ideas that we incorporate in Record-Mutate-Test.

2.1 User Error

People use software systems to complete a task. Human-computer interaction researchers model this interaction as a sequence of actions [9, 10].

Humans can fail to complete a task either because the sequence of actions is incorrect or inappropriate for the task, or because the sequence is correct, but they make mistakes in executing its actions [10]. Human error has been used to define the cause of both failure conditions.

Two definitions of human error have emerged. Human error can be seen as a *special class of actions* that cause unwanted consequences or as the *cause* for those actions [10]. For example, if a user clicks on the wrong button on a website, human error can denote the click action or the cognitive failure that caused the user to click on the button.

2.1.1 Definition of User Error Used in This Thesis

In this thesis, we define user error as the erroneous action committed by a human when interacting with a software system. That is, we adopt the definition in [10] and apply it to human-machine interaction. An erroneous action is one that leads to unwanted consequences, e.g., the system crashing. Said differently, user error is human error applied to human-machine interaction.

By focusing on how errors manifest, not what causes them, we can simulate realistic user errors without having to simulate how the human brain works.

In the rest of this section, we review two classes of taxonomies. The first class focuses on what causes humans to make mistakes. The second class focuses on how user errors manifest. If one views the cognitive processes that govern human actions as a program, then the taxonomies in first class are bugs in that program, while the second class of taxonomies describes how the bugs affect the program's output.

Ideally, we would simulate the reasons why people err, so that our technique covers all possible realistic user error manifestations. Alas, this is practically impossible, because their usage is conditioned on the existence of a program that models the human brain. Currently, to the best of our knowledge, there is no such program that passes Turing's test. Thus, we restrict ourselves to simulating how user errors manifest.

Nevertheless, it is worth reviewing what causes people to err, because the information they provide enables Record-Mutate-Test to generate more realistic errors.

2.1.2 Taxonomies of User Error as Cause of Erroneous Actions

Norman [11] identified two types of errors: slips and mistakes. Slips cause automatic behavior that is not consciously controlled to fault. Mistakes, on the other hand, are conscious and deliberate wrong courses of action. There are six types of slips:

- *Capture errors.* These errors manifest when a user wishes to complete a task that is similar to another one that is more common, and cause the user to perform the sequence of actions corresponding to the latter task, rather than the intended one.

- *Description errors.* These errors occur when users under-specify on what subject they want to perform an action, and several possible subjects match the description. These errors cause users to perform the right action on the wrong subject.

- *Data-driven errors.* These errors cause a person to use wrong, but readily available information to perform an action. For example, "I was assigning a visitor a room to use. I decided to call the department secretary to tell her the room number. I used the telephone in the alcove outside the room, with the room number in sight. Instead of dialing the secretary's phone number–which I use frequently and know very well–I dialed the room number." [11]

- *Associative activation errors.* These errors cause a human to mix together actions belonging to different, but similar tasks."My office phone rang. I picked up the receiver and bellowed 'Come in' at it." [11]

- *Loss-of-activation errors.* These errors cause people to forget to perform an action required to complete a task.

- *Mode errors.* These errors are specific to using technology. They are fostered by a device having more functionalities than controls (e.g., buttons). This requires the device to provide modes of functioning, and for each mode, a control executes a different functionality. Errors occur when people mistake the current mode and trigger the device to perform a different-than-intended functionality.

Mistakes happen when users consciously devise a sequence of actions that will not help them complete their task. Norman alludes to one of the sources for a wrong choice being previous experience, which seems to fit the current task, when in fact it does not. This indicates that people act according to learned rules, yet the rules are not always applicable. This concept of misapplying rules is extended in the work of James Reason, whose human error taxonomy we describe next.

James Reason developed the Generic Error-Modeling System (GEMS), a conceptual framework within which to describe the basic human error types [12]. GEMS is based on the three levels of human performance: skill-based level, rule-based level, and the knowledge-based level. Errors occur when functioning at each of these levels.

The skill-based level handles human actions that are frequently performed and controlling them does not require attention, for example typing in one's GMail address. After the typing intention is formed, the body caries out the action without conscious control over how one's hands move.

There are two types of errors at the skill-based level: slips and lapses. Slips are the same as in Norman's taxonomy, while lapses involve failures to remember an action or some information. Some forms of slips and lapses are [12]:

- *Double-capture slips*. Such errors cause a person to forget to check if the execution of a sequence of actions is progressing correctly and prevent a more practiced sequence from taking over the execution of a less practiced one. This type of errors is similar to Norman's capture errors. One account of such an error is "I intended to stop on the way home to buy some shoes, but 'woke up' to find that I had driven past the shoe store." In this example, we can see that the two sequence of actions, driving home and shopping, have a common prefix, but the person forgot to branch to the less practiced sequence, and instead carried on with the more frequent one.

- *Omissions following interruptions*. These errors cause a person to forget to check if the execution of a sequence of actions is resumed correctly after an interruption, and cause the person to forget to perform an action. An example is "I picked up my coat to go out when the phone rang. I answered it and then went out of the front door without my coat." The interruption caused by the telephone made the person forget to perform an action.

- *Omissions*. These errors cause a person to overestimate the progress of a sequence of automatically-performed actions, which causes them to miss performing some actions. Note that one can model the three error examples described so far in the same way, by deleting an action from a sequence, even though their causes are different.

- *Reduced intentionality*. These errors manifest when there is a delay between the formulation of a sequence of actions and performing them and cause people to mix actions belonging to different tasks. An example is "I intended to close the window as it was cold. I closed the cupboard door instead."

- *Perceptual confusions*. These errors occur when an action is performed on the wrong subject, but which is very similar to the correct one. An example is "I intended to pick up the milk bottle, but actually reached out for the squash bottle."

- *Interference errors*. These errors occur when people multi-task, i.e., when multiple sequences of action are active at the same time, which can result in a person combining them. An example is "I was beginning to make tea, when I heard the cat clamoring at the kitchen door to be fed. I opened a tin of cat food and started to spoon the contents in the tea pot instead of his bowl." Again, we see that this error and the ones for reduced intentionality and perceptual confusions can be modeled by replacing the correct subject with another one.

- *Repetitions*. These errors cause a person to misjudge the progress of a sequence of actions, making them perform an action already carried on.

- *Reversals*. These errors cause a person to undo a previously performed action. An example is "I got the fare out of my purse to give it to the bus conductor. A few moments later I put the coins back into the purse before the conductor had come to collect them."

The rule-based level works by recognizing a problem situation and applying a known solution, i.e., it is a pattern matching system consisting of `if (problem situation) then apply(solution)` rules. These rules are acquired through experience or learned through training or by reading documentation. It is worth noting that multiple rules may match the problem statement, and people have to choose which one to apply.

Finally, humans work at the knowledge-based level when they have to handle problems or tasks that have not been previously encountered. This level makes humans use their analytical prowess and stored knowledge.

Mistakes are errors committed while performing at the rule-based or knowledge-based level. They cause a person to elaborate an inadequate sequence of actions that will not help the person complete their task even if all actions are carried out correctly.

There are two types of mistakes at the rule-based level: misapplication of good rules, when a rule that proved its utility in the past is used in the wrong situation, and application of bad rules, when there is a deficiency in describing the problem to which a solution is applicable and the current problem wrongfully falls under this description. We review some mistakes caused by the misapplication of good rules:

- *First exceptions*. These mistakes cause a person to use a tried-out rule that fits the problem situation only to discover that its solution is inadequate. This prompts the person to create a new, more specific rule that captures this exception and is necessary to cope with the range of situational variations.

- *Signs, countersigns, and nonsigns*. Signs are rule inputs that satisfy some or all of the conditional aspects of a rule. Countersigns are inputs that indicate that the rule is inapplicable. Nonsigns are information about the problem statement that do not relate to any rules. These mistakes cause people to prefer a tried-out rule despite not being an exact fit to the description of the problem they are trying to solve. Users argue away the countersigns pointing to the misfit.

- *Rule strength*. These mistakes occur when there is no rule that completely matches the current problem situation, but there are multiple rules that partially match the situation. These mistakes cause a person to wrongfully choose the most frequently employed rule over the correct one.

Mistakes committed at the knowledge-based level appear either due to bounded rationality (people make rational decisions based on the facts they know and within the time they have) or due to the incorrect or incomplete mental model of the problem they are trying to solve. We describe some of these mistakes:

- *Selectivity*. These mistakes cause a person to give undue attention to irrelevant aspects of the problem to solve instead of the important ones.

- *Confirmation bias*. These mistakes cause a person to favor the first problem description he or she identified and then disregard future evidence disproving the description.

- *Out of sight, out of mind*. These mistakes cause people to give undue weight to facts that come readily to mind, while ignoring those that do not.

Researchers estimate that 60% of general human errors occur when operating at the skill-based level [13, 14]. Errors at the rule-based level account for 30%, and the rest of 10% correspond to errors committed at the knowledge-based level. However, the picture reverses when considering the error ratios. Humans commit errors at the knowledge-based level more frequently than at the other levels. The consequences of human error differ among errors committed at the various performance levels. Errors at the skill-based level have the least worst consequences, while errors at the knowledge-based level have the worst.

These error taxonomies influence what Record-Mutate-Test records when a user interacts with a system. For example, to simulate description errors, our technique must record what is the target of a user's interaction, e.g., the

button labeled "Submit," and must take into account the environment in which the user interaction takes place, e.g., what are the other available buttons.

The taxonomies described in the next section focus on the mechanics of user errors, but lack the power to sort errors based on how realistic they are. We argue that combining elements of both classes of taxonomies enables Record-Mutate-Test to practically simulate realistic user errors.

2.1.3 Taxonomies of Human Errors Based on Error Manifestation

There exists previous work that focuses on characterizing how human errors manifest, i.e., how errors affect the sequence of actions a user performs. Such taxonomies are oblivious to what causes humans to make mistakes. These taxonomies provide us with error models that we can apply to a sequence of actions. They form the basis of Record-Mutate-Test. Here, we review some of the proposed taxonomies.

Swain and Guttman's error taxonomy [15] distinguishes between three main forms of human error: errors of omission, errors of commission, and extraneous actions. Errors of omission manifest as users forgetting to perform an action. Research shows that they are prevalent, at least in safety critical systems [12].

Errors of commission manifest as users incorrectly performing required actions. Some of these errors are:

- *Selection errors*, when users interact with the wrong subject, e.g., click on the wrong UI widget
- *Sequence errors*, when users execute an action out of order
- *Timing errors*, when users interact with a system at the wrong time, e.g., too early or too late
- *Qualitative errors*, when users misjudge the extent to which they should perform the action, e.g., they turn a dial too much

Users need not perform extraneous actions in order to complete a task.

Hollnagel develops a more detailed error taxonomy [10] that distinguishes between the psychological causes for error, which he terms genotypes, and the way errors manifest, which he terms phenotypes, and describes a taxonomy for phenotypes. The taxonomy contains two types of phenotypes that cover most of the human errors reported in the literature:

- *Zero-order phenotypes* refer to errors that can be discovered by looking at a single action in a sequence
- *First-order phenotypes* refer to errors that can be discovered by using multiple zero-order phenotypes

There are two categories of zero-order phenotypes: one for the timing of actions and one for the order of actions. The phenotypes referring to action timing are:

- *Correct action*, when no error occurs
- *Premature start of action*, when a user starts an action too early
- *Delayed start of action*, when a user starts an action after its starting deadline
- *Premature finishing of action*, when a user starts an action at the correct time, but finishes too early
- *Too short duration*, when a user starts an action after its starting deadline and finishes too early

- *Delayed finishing of action*, when a user starts an action at the correct time, but finishes too late
- *Too long duration*, when a user starts an action before its starting deadline and finishes too late
- *Omission*, when a user starts an action after its latest finishing time

The phenotypes referring to the order of a sequence of actions are:

- *Correct action*, when no error occurs
- *Omission*, when a user skips the current action and execution continues with its successor, e.g., jumping from action i to action $i + 1$
- *Jump forward* is a generalization of omission and corresponds to omitting multiple consecutive actions, e.g., jumping from action i to action $i + 3$
- *Jump backward*, when a user performs an already-executed action, i.e., it is the reverse of the jump forward phenotype, e.g., jumping from action i to action $i - 3$
- *Repetition*, when action execution does not advance to the next step, e.g., action i is followed by itself
- *Intrusion*, when a user performs an action not part of the sequence of actions

By combining multiple zero-order phenotypes, one can define new errors and differentiate among similar errors. These are called first-order phenotypes, and we list them here:

- *Spurious intrusion*, when a user performs an unnecessary action, but then reverts to executing the correct action
- *Jump, skip*, when a user skips one or more steps in the action sequence
- *Place losing* is characterized by the lack of order in the sequence of actions
- *Recovery*, when the user returns to a previously skipped action
- *Side-tracking*, when the user executes an inappropriate subsequence of actions, but which is part of the correct sequence
- *Restart*, when the user starts reexecuting a sequence of actions from the beginning
- *Capture*, when the user executes a subsequence of actions that is part of an incorrect sequence
- *Branching*, when two sequences of actions have a common prefix and the user executes the wrong sequence. This is similar to Norman's capture errors and Reason's double-capture slips.
- *Reversal*, when the user executes two consecutive actions in the reversed order
- *Time compression*, when the user executes a sequence of actions in a shorter-than-expected time

Hollnagel defines second-order phenotypes based on combining multiple first-order phenotypes, but remarks that the first two classes of phenotypes are sufficient to cover most of the human errors described in the literature. Our experiment, presented in Section 3.9.1 confirms this. Thus, we do not discuss second-order phenotypes here.

The taxonomies reviewed in this section define formulas one can apply to a sequence of actions to generate user errors. Alas, they are not precise in what errors are more realistic than others. For example, Hollnagel's *intrusion* error does not specify what is the extraneous user action. On the other hand, the taxonomies described in the previous section give us hints of what these actions might be, e.g., an action performed on a similar subject. We believe that combining elements of both classes of taxonomies enables Record-Mutate-Test to generate more realistic user errors.

2.2 Techniques for Preventing User Errors

Error prevention techniques aim to eliminate user errors. In this section, we review four error prevention techniques: training 2.2.1, automation 2.2.2, error-aware user interfaces 2.2.3, and delaying operations 2.2.4.

2.2.1 Training and Documentation

A first line of defense against user errors is training. Training teaches one how to properly interact with a system and how to react in case problems arise. Another way to see training is as a means to "proceduralize" user actions and, thus, reduce the variability of human performance.

Training is incomplete, as shown by studies that reveal that even highly trained humans will encounter situations their training does not cover. For example, nuclear power plant operators, who are well-trained and whose actions are safety-critical, are responsible for the largest fraction of significant reactor incidents [12]. When confronted with unforeseen situations, humans perform at the knowledge-based level and the chances of error are highest.

Another drawback of training is that it may be outdated. Amazon and Google release new versions of their systems multiple times a day [13, 16]. Thus, it may be the case that the practices a system administrator was taught become less efficient or, worse, damaging with every new release.

Documentation, such as user manuals, can be thought of as training and, thus, it suffers from the same drawbacks. Documentation has the added drawback that it is non-interactive, potentially leaving readers with unanswered questions or, worse, with an incorrect model of how a system functions and what are its safe operation margins.

In [4], Casey describes an incident that happened at a military nuclear power plant and resulted in casualties. The accident involved a maintenance crew who had to replace some of the reactor's rods. Their work involved disassembling the top of the reactor, replacing the rods, and then reassembling the top. The crew was given documentation that detailed the steps necessary to disassemble the top of the reactor. When it came to reassembling it, the documentation specified that the crew needed to follow the steps for disassembling, but in the reverse order. However, this was impossible and prompted the maintenance crew to improvise and unwillingly take the nuclear power plant outside its safe operation parameters. This caused an increase in energy production that lead to an explosion that killed all crew members.

The conclusion is that training cannot prevent humans from making mistakes. At best, it can reduce the probability of error. There will always be tasks not covered during training that a person must complete. Such tasks require one to reason at the knowledge-based level, which has high probabilities of error.

2.2.2 Automation

A second means of preventing user errors from affecting the dependability of a system is by automating that system's functioning, i.e., take human interaction out of the control loop. This approach is most suitable for industrial processes,

such as electric power networks and telephone systems, where computer systems take over the human operator's activities of control, planning, and problem solving. However, all attempts to automate the tasks of the human operator fail due to the ironies of automation [17].

There are two challenges with automation. First, the burden of ensuring the system is running correctly is shifted from the human operator to a system designer. Second, it is not possible to completely automate a large system.

The problem with tasking a system designer to automate an industrial process is that she/he has less experience with the process, how it works, and where the process is fragile, compared to the human operator the software control system is replacing. So, designers may commit errors, which can be a major source of operating problems [17]. Designers may also not know how to automate a task and rely on human operators [17].

Thus, human operators are still needed to watch over the automated system's well functioning and possibly take over when the system starts faltering. Ironically, researchers point out that the more automated a system, the higher the skill the human system operator must have, and the more likely the operator is to commit an error [17, 13]. Since the human operator intervenes only when the system starts behaving abnormally, which is rare, the operator's skills deteriorate, which negatively impacts his/her ability to respond to the system's malfunctioning [17].

A further problem with automation is that it demotes human operators to monitoring the output of a system, yet psychological tests have shown that it is impossible for humans to maintain visual attention, for more than half an hour, to a source of information that rarely changes [17]. This means that operators tune out, potentially missing warnings issued by the automated control system. One way to solve this issue is to trigger warnings more frequently. Alas, this leads to operators starting to distrust the system, which worsens their performance.

Another problem of automation is that it conditions the operators' actions on the correctness of the automated system: if the system does not report the state of a controlled process, operators will make mistakes. Operators were deemed responsible for the Three Mile Island nuclear incident [18]. They misdiagnosed a nuclear reactor problem because the monitoring system did not provide all the information necessary to reach the right diagnose.

A less intrusive form of automation is one in which the human operator is in control of a process, but she/he is aided by a computer system. There are two types of aids [12]: decision aids help operators devise plans of actions, while memory aids help operators execute a plan by, for example, reminding operators what steps need be followed.

Unfortunately, relying on automated systems can instill into people a false sense of safety and understanding of the system. Casey [4] describes the crash of Airbus 320's first public flight. The cause of the accident was the pilot disconnecting the automatic flight control system, which altered the way the plane behaved and surprised the pilot.

Moving away from industrial automation to everyday users, we can observe automation in program installation wizards. Alas, the wizards may not configure the program to fit the user's requirements, so the user has to manually tune the installation procedure. This creates opportunity for errors to creep in.

The conclusion is that automation cannot prevent user error. While the goal of automation is commendable, in practice, automated systems are incomplete and still require human operators, who will commit errors.

2.2.3 Error-aware User Interfaces

In this section we review techniques that system designers can employ to design human-machine interfaces that reduce the likelihood of, prevent, or lessen the effects of user errors. The main idea of such techniques is to understand where errors can occur in an interface, what fosters them, and change the design to remove users' possibility to err.

A first technique is to avoid uniformity of control knobs. Norman [11] presents the example of the error-aware interface developed by automobile manufacturers. Cars require different fluids in order to run properly, for example

windscreen washer solution and engine oil. Different receptacles have different shapes and openings, and different fluids are colored differently, so as to prevent drivers from pouring the wrong fluid into the wrong receptacle.

A second technique is to force users to do the correct action or to not forget to perform an action [11]. Automated teller machines, for example, require users to remove their card prior to dispensing money. This is to prevent people from walking away without their card.

A third technique is to provide users with context information, feedback on the effects of their actions, and support for fixing the errors they committed [11]. Consider the autocorrect functionality present today in text editors. If a user mistypes a word, that word gets underlined, and the user has the possibility to select the correct spelling.

A new user interface for setting file-access permissions in Windows XP proposed by Maxion and Reeder [19] gives users feedback on what are the effective permissions they set on files and folders. Their results indicate a 300% increase in successful task completion, a 94% reduction in commissions of a category of errors, and a nearly 300% speed up in task completion time.

Kiciman and Wang [20] describe an automated technique to discover correctness constraints for a system's configuration. Their technique uses presumed-to-be-good configurations to learn properties of correct configuration parameters that refer size, domain, equality, or reference. Their experiments show the technique was able to detect 33% of a set of known configuration errors. One can imagine using the technique on-line, as a system is being configured, to forewarn system administrators of possible errors.

A fourth technique is to prevent users from performing irreversible actions [8]. A naïve implementation is to ask users for confirmation when they perform an action that cannot be easily undone, such as deleting a file. Alas, this solution is not perfect, because users will stop caring about the displayed warning and will blindly agree to it or they will misinterpret the warning as pointing out the action to be performed rather than its subject. To fix this problem, Norman proposes that irreversible actions be removed [11]. Nowadays, there exist alternatives that enable users to undo some of their actions, but not all. We cover them in the next sections.

2.2.4 Delaying Operations

People quickly realize they made an error. Thus, one way to prevent user errors to propagate to an entire system is to delay performing an action to give people a chance to realize their error and fix it.

Norman [11] describes a problem occurring in a research lab that involved people throwing away their notes only to realize the next day that they needed them. The solution was to employ a different trash can for each day of the week and delay emptying a trash can to the day before it was reused. Thus, people had one week at their disposal to realize they needed the notes and had the chance to retrieve them.

More recently, delaying (or buffering) operations has been employed to reduce the error rate for operators in charge of dispatching tickets that describe system problems to technicians who can solve them [21].

Delaying operations is also encountered in real world systems. For example, GMail provides users the possibility of undoing sends. However, a more factual description is delayed send, because user have up to 30 seconds to undo sending an email.

2.2.5 Summary of Techniques for Preventing Human Error

The conclusion of this section is that human error is unavoidable, and that no prevention mechanism, or their combination, is sufficient. Human creativity is boundless, and it reflects in the way people misuse computer systems.

James Reason argues that "that the most productive strategy for dealing with active errors is to focus upon controlling their consequences rather than upon striving for their elimination." [12]. Therefore, in the next section, we present prior work which focuses on coping with the effects of user error.

2.3 Techniques for Coping with User Errors

A software system can provide users a natural means to recover from an error by enabling them to *undo* it [8]. There are five major ways to implement undo [22], and we review them in the following sections. All of the undo techniques suffer from the drawback that there exists a class of errors they cannot undo, e.g., disconnecting the power cable.

2.3.1 The Complete Rerun Technique

This technique achieves undo by resetting the state of a system to its initial state, then replays each user command, but stops just before executing the erroneous command, which erases its effects.

The technique assumes that all user interactions are logged, that there exists a means to replay the logged interactions, and that the system is deterministic with respect to the logged actions, i.e., replaying the logged actions consistently leads the system to the same state.

The major drawback of this technique is its poor efficiency: if the log contains numerous commands, replaying it is time-consuming. Another drawback is that if the logged commands cause the system to externalize its state, then replaying the log causes the system to externalize its state again.

2.3.2 The Full Checkpoint Technique

The idea behind the full checkpoint technique is to provide checkpoints of the state of a system at different times. If a user makes an error, they can restore a checkpoint, and the effect of the error vanishes.

Taking a checkpoint before every user interaction is best, because the user can travel back in time to any program state. The downside of this variant is its vast storage space requirements. One way to mitigate this is to keep a limited number of checkpoints, log the user interaction between two checkpoints, and after the state is rolled back, replay the logged interaction, to bring the state of the system up to date.

Brown [23] describes a technique to allow system administrators to fix their mistakes. The technique introduces two steps after undo. In the repair step, system administrators fix their system. In the redo step, the state of the system is brought up to date by replaying the logged interactions.

2.3.3 The Partial Checkpoint Technique

The idea behind partial checkpoint techniques is to reduce the storage space requirements of the full checkpoint techniques by making each checkpoint consist of only the part of the state that will be changed by executing a user's action. To undo a user's action, the technique copies back the checkpoint.

2.3.4 The Inverse Command Technique

This undo technique relies on executing the "inverse" of a command. Although this looks as an intuitive means to provide undo functionality, it has the major drawback that multiple user actions do not have inverses, e.g., overwriting

a configuration parameter.

2.3.5 Spatial Replication

Another technique to undo user errors is to have multiple system replicas that run in parallel and whose states are synchronized [8]. If a user interacts with only one replica at a time and the user makes a mistake, the state of the affected replica can be recovered from the other replicas. An example of applying this technique is systems based on byzantine fault tolerant protocols [24]. For example, if a system administrator corrupts the state of a replica, then the other replicas will detect this and the affected replica can be discarded and rebuilt from the others [8].

This technique can cope with system administrator errors during the maintenance of a single machine. However, if errors affect all replicas, this technique cannot undo them, e.g., errors committed by end users of services running on top of spatial replicated systems, assuming all replicas run the same code, or system administrator actions that affect all replicas, such as upgrading the operating system, because all replicas will have the same, erroneous, state.

A different way to use replicas to cope with user errors is to confine the user's actions to a part of the system, check for problems, and if none are found, then propagate the actions to the entire system. This technique was applied by Nagaraja et al [25] to enable system administrators of Internet services to test their actions. In their work, a system is split into two slices: an online, live slice that serves requests originating from real users and a validation slice that is a replica of a part of the live slice. The validation slice is fed live requests and the outputs of the two slices are compared. If the comparison succeeds, then the validation slide can be transparently moved to the live slice, without having to perform any configuration change, eliminating the chance of introducing errors during migration.

2.3.6 Summary of Techniques for Coping with Human Error

The techniques described in this section enable users to undo the effects of the errors they commit. Their main idea is to roll back the state of a system to a previous, correct state. However, doing so discards the effect of benign user commands that occurred since the checkpoint, so one must replay them or their effects. Techniques differ in what the correct state is and how to bring the state of the system up to date.

We will reuse the idea of combining checkpoints with replaying logs when describing Record-Mutate-Test. This helps our technique achieve efficiency, by reducing the time required to run a test.

2.4 Testing Techniques Against User Errors

Testing is a form of quality assurance process that feeds inputs to a program and checks its outcome against a predefined one. In this section, we review testing techniques that can be used to point out a system's deficiencies when it comes to handling user errors. Table 2.1 shows the testing techniques we cover in this section and whether they possess the properties of the ideal testing technique, as described in Section 1.4.

The first two rows, below the table header, correspond to testing performed by a developer, either manually or using code. The third row describes a testing technique that involves real users. The fourth row shows a technique that reuses errors observed during real-user testing. The fifth row shows a technique to increase the number of users that take part in the testing process. The last row is a technique that leverages execution information to generate tests.

The first two columns, after the header, indicate whether the technique is automated, i.e., if developers need to manually write tests and if the tests are reusable. The third column specifies whether a techniques is efficient, i.e.,

Technique / Property	Non-developer test generation	Reusable	Efficient	Realistic	Relevant	Portable
Manual Developer Testing	✗	✗	✗	✗	✗	✗
Code-Driven Testing	✗	✓	✗	✗	✗	✗
Real-User Testing	✗	✗	✗	✓	✓	✗
Playback	✗	✓	✗	✓	✓	✗
Crowdsourced Testing	✓	✗	✗	✓	✓	✗
Statistical Testing	✓	✓	✓	✗	✓	✓

Table 2.1: The testing techniques that can be used to test a system against user errors and how they stack up with respect to the properties of the ideal testing solution.

whether it generates redundant tests or not. The next column shows whether the technique uses behavior representative of real users. The fifth column indicates if the technique focuses on parts of the system that users find relevant. The final column specifies if the techniques can reuse errors across multiple systems.

2.4.1 Developer Testing

Developer testing is the form of testing in which the developer defines the entire test suite. In this section we review two approaches to developer testing: manual testing and code-driven testing.

2.4.1.1 Manual Testing

Manual testing is the form of testing that requires fewest preparations. Developers use the system and, unavoidably, commit errors. This type of testing is expensive, because developers need to manually run the tests. It may also not be realistic, because developers commit errors that may not be representative of those real users commit. Manual testing may not be relevant, because developers and users may disagree about which are the important parts of the system. However, this testing is relevant for the dependability of a system, because developers know where the system is fragile and what program inputs trigger error handling, which is code that runs seldom, but when it runs, it has to run perfectly because it is saving the system from the abyss of failure. This type of testing is neither portable nor reusable.

2.4.1.2 Code-Driven Testing

Code-driven testing encodes manual tests as test cases, making tests reproducible. Successful embodiments of code-driven testing are the xUnit (jUnit [26], nUnit[27]) set of frameworks, the Robotium [28], TestDroid [29], and MonkeyTalk [30] frameworks for testing Android applications, and the Selenium [31] framework for testing web applications. Continuous integration systems, like Jenkins [32], automatically run a test suite on every code change.

Alisauskas [33] describes a community-based testing framework that enables Android developers to share their devices with other developers so that they can run code-driven tests on a multitude of Android devices. Such a framework is useful in getting coverage of Android devices that run various versions of the Android OS, have different performance characteristics, and different screen sizes.

There exist commercial solutions for running code-driven tests targeting smartphone and web applications on real devices, such as Xamarin Test cloud [34], TestDroid [29], and PerfectoMobile [35], or testing services that use emulators, such as CloudMonkey [36].

Record-Mutate-Test generates tests that can be used together with test automation tools, to reduce developer effort.

2.4.2 Real-User Testing

In this type of testing, the tester asks a set of users to solve a set of predefined tasks. The tester and the users are collocated. Users may be audio- and video-taped. Related to this thesis, such testing enables developers to test their system against real user errors.

Brown and Patterson [37, 8] argue that software system dependability benchmarks should include human factors, specifically system administrators. The reasoning behind this claim is that human operators are required to manage a system and their presence impacts the system's dependability: they can help a system perform better and recover faster, but they can also cause it to misbehave. The authors report that that the variability of human performance has a significant impact on the benchmarking results. They propose a two-pronged approach to managing variability: have each operator take part in a single run of the benchmark, and choose operators with similar skill and expertise levels. Their results suggest that at least 5 operators are needed to overcome human performance variability.

Whitten and Tygar [38] studied the usability of PGP 5.0 in an experiment with twelve participants. Participants were asked to perform routine tasks involving sending cryptographically secure emails by using PGP to encrypt them. The study revealed that a quarter of participants failed to send the encrypted email, one participant did not manage to encrypt the secret, and only a third of the participants managed to send correctly encrypted email.

Maxion and Reeder [19] performed an experiment with 24 participants that targeted user errors made when configuring access file permissions in Windows XP.

Real-user testing has the benefit of confronting systems with erroneous behavior representative of their users. We wish for the tests our technique generates to be as representative.

2.4.3 Playback

In this type of testing, researchers process the audio and video recordings of real-user experiments described in the previous section and devise means to reproduce the observed errors. This helps amortize the cost of running the experiment over multiple usages of the discovered errors. The similarity to Record-Mutate-Test is in the testing's focus on reproducing representative user errors.

Nagaraja et al [25] studied the effects of operator mistakes in Internet services. They proposed that operators use a validation environment prior to deploying their changes to an entire system. In order to test the effectiveness of their validation environment, they recorded detailed traces of operators errors, manually derived scripts that simulate them, and then replayed them. The drawback of this approach is that the errors are specific to a system. In subsequent work, Nagaraja et al [39] propose a similar validation environment for actions performed by database administrators, but in this case, the administrator errors are manually created based on database administrator manuals and books.

Vieira et al[40] focus on the recoverability aspect of database management systems. To that end, they extend the TPC-C benchmark to inject database administrator errors. The errors are reproduced by mimicking wrong operator commands using the same means of interacting with the system as the human operators did.

2.4.4 Crowdsourced Testing

We define this type of testing as one in which a system is made available to multiple users who use it in their environment, as opposed to a laboratory setting.

Nielsen [41] proposed the idea of distributing to users software that can log user interaction. We follow the same idea in Record-Mutate-Test. When users complete a set of assigned tasks, the developer receives the logs and

uses them to understand how users performed the assigned tasks, what difficulties they encountered, whether they committed errors, and how were bugs triggered.

Researchers have proposed tools for testing of web [42, 43, 44] and desktop applications [45, 46, 47, 48, 49]. One way to record user interaction with a web application is to use a proxy (e.g., Fiddler [42]) that logs all network traffic between a web browser and an application server. Another possibility is to use proxies to inject JavaScript code into HTML pages to track user interaction (e.g., Mugshot [43] and UsaProxy [44]).

One can record a user's every click and keystroke when using a desktop application by modifying the operating system (e.g., RUI [45] and AppMonitor [46]), by using hooks provided by the operating system (e.g., GUITESTER [47]), or by integrating the testing tool tightly with a program's runtime (e.g., KALDI [48]).

Virtual machines can be used to record user interaction. In this approach, an entire OS runs inside a virtual machine, which captures a program execution and enables developers to replay it later [50, 51].

There exist commercial solutions where users test a program and provide feedback [52, 53]. uTest [52] is an intermediary between developers and professional testers. Developers receive feedback in the form of bug reports, test cases, screenshots, or videos. AppLover [53] is a similar testing service for Android smartphone applications.

Overall, the benefit of crowdsourced testing is that it enables developers to test their systems in real-world conditions, with real users using their own devices. We leverage this idea in Record-Mutate-Test.

2.4.5 Error Injection

The goal of such techniques is to test how systems react when there are hardware errors, or when external functionality used by a module returns an error, or when a system is exposed to wrong configurations. But it can also be used for testing against user errors.

Software faults, an error injection technique, focuses on injecting the effects of programmer errors in a program's binary. Christmansson and Chillarege [54] analyzed the bugs in an IBM operating systems and developed an error model that specified what errors to inject, where in the program's binary, and when. Later work [55] shows that the injected errors cover less than 60% of all possible programmer errors.

Mutation testing [56] is an error injection technique in which programmer errors are introduced in the source code of a program, one error at a time. Each injection begets a new version of the program called a mutant. Developers run a test suite against each mutant. If the tests fail, then the mutation is said to be killed. As such, mutation testing is a means to asses the adequacy of test suites, i.e., how able are they to discover bugs.

The challenges faced by mutation testing in reducing the time required to run a test suite against all mutants are of interest to this thesis. There are two solutions: reduce the number of injected errors and reduce the execution cost.

To reduce the number of generated mutant programs, Jia [57] suggests one can apply techniques such as:

- *Mutant sampling* [58, 59], where one runs a test suite against a percentage of randomly chosen mutants
- *Mutant clustering* [60], where mutants are clustered based on killable test cases
- *Selective mutation* [61], where only certain error types are injected
- *High-order mutation* [62], where instead of injecting a single error, a compounded error is injected

To reduce test execution time, one can apply techniques such as:

- *Weak and firm mutation*, when the check if the outcome of the original and the mutant program differ is performed close to when the injected error executed

- *Runtime optimization techniques* include determining the impact of the mutation directly from the source code
- *Parallel execution* techniques distribute executing tests over the available processors

ConfErr [14] is an automated testing tool that injects errors into configuration files. The errors it injects are representative of the errors system administrators can make. ConfErr injects three types of errors:

- *Spelling mistakes*. These errors represent typos committed during hurried typing. ConfErr injects five types of spelling mistakes:
  - *Omissions*, when a letter is forgotten
  - *Insertions*, when a letter is introduced in a word
  - *Substitutions*, when a letter is replaced with another one that is adjacent on the keyboard
  - *Case alterations*, when a letter is replaced with another one that is obtained by altering the state of modifier keys, e.g., `Shift`
  - *Transpositions*, when two consecutive letters in a word are swapped
- *Structural errors*. Configuration files are generally structured as sections and directives. Structural errors modify the structure of the configuration file in fashions similar to spelling mistakes, but applied to sections or directives. Another structural error is to configure one system using a configuration directive from a different system.
- *Semantic errors*. There are two types of semantic errors: errors that cause configuration files to violate configuration constraints, and errors due to the operator not knowing exactly the meaning of a given parameter and using it to configure a similar but different aspect of the system.

ConfErr laid the groundwork for part of the work presented in this thesis and served as inspiration for the rest of it.

2.4.6 Statistical Testing

Whittaker and Thomason [63] propose a testing technique that generates test program inputs using a probability distribution rooted in the actual or expected use of a program. They propose to use Markov chains to determine what program inputs to generate and when to stop testing. In this section, we cover only using the Markov chain for generating tests, while the testing stopping criteria is described in Section 2.5.2.

Their technique generates tests by following a path in a Markov chain. The chain contains the states of the program. Arcs correspond to user actions and are labeled with the probability of users performing that action. The authors specifically mention using real usage data to determine the probability of each arc. The crowdsourcing technique presented in Section 3.7 can be used to compute these probabilities.

2.5 Testing Quality Criteria

A program's reliability is considered to be roughly proportional to the volume and quality of testing done on that program. Software vendors, therefore, use extensive test suites to reduce the risk of shipping buggy software.

Software engineers invented quantitative ways of assessing the quality of a test suite, called testing metrics. It is common for development organizations to choose a target metrics value and, once this is met, to declare the software ready to ship. We use testing metrics as a proxy for measuring the effectiveness of the tests Record-Mutate-Test generates. It is, therefore, interesting to review some of these metrics.

2.5.1 Code Coverage Criteria

This section describes three code coverage metrics: line coverage, branch coverage, and path coverage. These metrics measure how well tests explore a system's code. Whether the tests correspond to functionality specified in the system's requirements or not is irrelevant. The testing process stops once the tests generate satisfactory values for the employed code coverage metrics.

In the rest of this section, we illustrate code coverage metrics by referring to the example code below (Figure 2.1) and discuss what coverage value one achieves by running `example(0)`.

```
   int example( int a )
   {
1    int res=0;
2    if (a==0) {
3      res=1;
4      libraryCallA();
5      libraryCallB();
     } else {
6      crash();
     }
7    if (a==1) {
8      crash();
     } else {
9      res--;
10     libraryCallB();
11     libraryCallA();
     }
12     return res;
   }
```

Figure 2.1: Example program.

The `example` function takes as parameter an integer `a`. If the value of `a` is zero, then the application sets the value of `res` to one. Otherwise, it crashes. Next, if the value of `a` is one, then it crashes. Otherwise, it decrements `res`. Finally, the program returns the value of `res`.

2.5.1.1 The Line Code Coverage Metric

The most popular code coverage metric is *line coverage*. This code coverage metric measures the percentage of lines of code in a program that were executed at least once during testing [64]. There exist code coverage metrics that are similar to line code coverage, but differ in the unit of measure, e.g., statement coverage or basic block coverage. Note that although identical in principle, line code coverage, statement code coverage, and basic block coverage yield different values for the same program and the same tests.

The largest value differences are between statement coverage and basic block coverage, because the latter groups all consecutive, non-branching program statements into a single unit of measure. In so doing, it abstracts the code of a system into a control flow graph and counts how many of its edges were executed.

In our example, calling `example(0)` exercises all lines except 6 and 8 (so, 10 out of 12 lines), yielding a line code coverage of 83%. An organization that uses 80% coverage as a criterion for shipping would consider this code to be sufficiently well tested, despite the two undiscovered crash scenarios.

The value of the basic block coverage metric for our example is 71%, because 5 out of the 7 basic blocks are exercised: line sets {1,2}, {3,4,5}, {7}, {9,10,11}, and {12}.

2.5.1.2 The Branch Coverage Metric

Branch coverage [64], also known as decision coverage, is often considered a more informative metric, compared to line code coverage. It reports the fraction of branch outcomes taken during a test, i.e., to what extent the branch conditions evaluated to both `true` and `false`.

In our example, there are 4 condition branches and the `example(0)` test takes 2 of them (2→3 and 7→9), resulting in 50% branch coverage. Similar to the earlier metrics, this one views the branch points as being independent, with the sequence in which branches are taken not affecting the coverage metric at all.

2.5.1.3 The Path Coverage Metric

The path coverage metric computes the percentage of program paths that have been explored [64]. A program path is the sequence of branches from the program entry point to its exit point.

One cannot always compute meaningful values for the path coverage metric. A program that has loops that have an unbounded number of iterations, e.g., because the loop expects some user input, has an infinite number of paths. Therefore, it is futile to compute path coverage and use it to assess the quality of a test suite, because any test suite will achieve 0% path coverage. One can curtail the number of iterations considered for a loop, to make computing the metric feasible.

In our example, there are 4 potential execution paths (of which 2 are feasible); the test exercises only one of them (sequence <1,2→3,4,5,7→9,10,11,12>), resulting in 25% (50%) path coverage.

2.5.2 Analytical Stopping Criteria

The work of Whittaker and Thomason [63] on using Markov chains to generate tests also proposes an analytical testing stopping criteria. The idea behind this criteria is to use tests to reconstruct the Markov chain describing a system's usage, called a usage chain. Initially, a copy of the usage chain is created, called the testing chain, but all its arcs have probability zero. Every time a test is executed, the sequence of user actions is used to increment the values of the arcs it involves. Testing stops when the two chains are indistinguishable, up to a threshold. Whenever a failure occurs, a new state, corresponding to the failure, is added to the testing chain. This requires more tests to be generated so as to compensate for making the testing chain and the usage chain different.

2.6 The Challenge of User Anonymity

Modern software often provides automated testing and bug reporting facilities that enable developers to improve the software after release. Alas, this comes at the cost of user anonymity: reported execution traces may identify users. In this section, we review previous work that aims to protect users' anonymity.

K-anonymity is a technique to provide users with a limited, quantifiable level of anonymity. A data set satisfies k-anonymity if and only if each set of values that can be used to identify the source of a data element appears at least k times in the set [65], i.e., the source of an element cannot be narrowed down to fewer than k candidates. We say that

each element of a data set satisfying k-anonymity is k-anonymous. This is the main idea used by Record-Mutate-Test to protect users' anonymity.

Castro and Costa [66] describe a technique to replace program inputs that cause a system to fail, which may identify users, with equivalent inputs, which trigger the same failure, but protect users' anonymity. Their technique relies on collecting path constraints that describe the decisions made by that system. Camouflage [67] builds on [66] and introduces two techniques to enlarge the set of bug-triggering inputs. We take a similar approach to ensure that a test cannot identify its source user.

Query restriction techniques pertaining to statistical databases [68] are another possibility to protect users' anonymity. We borrow their idea to limit the type of queries to ensure that one cannot deduce who is the source of a test based on the behavior a test contains.

Windows Error Reporting (WER) [69] is a an automated system for processing error reports. WER collects memory dumps when an application crashes and sends parts of them to Microsoft. It protects users anonymity by restricting access to error reports for debugging purposes only. WER is embedded into the operating system, making it easy for developers to take advantage of its bug reporting features.

2.7 Chapter Summary

This chapter described previous work on human error. We define user errors as human errors when interacting with a system. The chapter presented multiple definitions of human error, then described two classes of taxonomies of human error. Recall that there are two ways to look at human errors: from the point of view of their psychological causes, answering the question "Why do humans err?" and from the point of view of their manifestation, answering the question "How do humans err?" In this thesis, we are interested in a practical solution for testing software systems against human errors, so we define human errors as *actions*.

Next, the chapter described means to handle user error. There are three main approaches. First, prevent users from erring, by training them, by removing human-machine interaction through automation, by identifying error-prone aspects of an interface, or by allowing humans to correct their mistakes. Since user errors are pervasive, a second line of defense is represented by techniques that enable users to undo their errors.

Third, since a developer's reasoning is bounded, and human creativity is boundless, it is likely that the algorithms and implementations of the first two classes of techniques do not cover all possible user errors. Thus, one must test them, so that one can fix their shortcomings. The work described in this thesis falls in this category. Of particular interest are techniques that enable real users to test programs. This lends realism to the testing process. We extend such techniques by generating tests that focus on realistic user errors.

The chapter continues with criteria for assessing a test suite's effectiveness at discovering bugs. Statistical testing is of interest to this thesis, because it suggests to generate tests based on the expected usage of a program by its real users. We incorporate this observation in the requirements we wish our testing technique to satisfy.

This chapter closes with a review of techniques for protecting users' anonymity. As we will later see, we use the concept of k-anonymity to provide anonymity to users who contribute program interaction logs for testing.

This chapter described the ideas that underline our testing technique, Record-Mutate-Test: how to model user error manifestations, checkpoint and reexecution, automated test execution, leveraging users to derive tests and run tests, ways to achieve increased efficiency, derive tests based on real-user usage profiles, means to quantify the effectiveness of a test suite, and techniques to protect users' anonymity.

Chapter 3

Record-Mutate-Test: A Technique to Generate Tests that Simulate Realistic User Errors

3.1 Introduction

Testing is a common way to ensure that a software system respects certain properties. Ideally, one performs exhaustive testing, i.e., feed every possible program input and check that the system's output is correct. Alas, the number of inputs accepted by a system is too large to test every one of them, e.g., a program that adds two 32-bit integers has 2^{64} inputs. Even if we run a single test for each system execution path, the number of tests is still prohibitively large for large systems, because the number of paths is exponential in the number of branch statements in the source code of the system, which is proportional to the size of the system's source code. It seems Dijkstra was right in that "Program testing can be used to show the presence of bugs, but never to show their absence."

There are two solutions to being unable to test all execution paths: either use formal verification techniques to prove correctness or forgo complete correctness and focus on providing evidence that the system is correct with respect to some properties. Current formal verification techniques, such as theorem proving, model checking, and abstract interpretation, do not scale to systems comprising millions of lines of code [1].

A practical solution is to provide evidence that a system is correct with respect to certain properties. Properties that are important for any system to uphold are, e.g., availability (i.e., the ability of a system to deliver services when requested [1]), reliability (i.e., the ability of the system to deliver services as specified [1]), safety (i.e., the ability of the system to operate without catastrophic consequences [1]), security (i.e., the ability of the system to protect itself against accidental or deliberate intrusion [1]). Examples of functional properties include performance envelopes (i.e., the system responds to commands within a time limit, e.g., a web server always replies within 10 ms to all HTTP requests or a web application can be loaded in 100 ms using a 100 Mb/second connection) or compliance (i.e., the system adheres to established rules that govern its functionality, e.g., a web browser is HTML 5 compliant).

3.2 Resilience to User Errors

In this thesis, the property we wish to *automatically* test is that a software system is resilient to realistic user errors. In this chapter, we present Record-Mutate-Test, a technique for *automated* test generation.

3.2.1 Definition of User Error

We define user errors as the commands a user issues to a software system that cause the system to perform an action with unwanted consequences that run contrary to the goal the user is pursuing. We do not consider the errors committed by a system's developers.

Examples of user errors are tapping on the wrong button in the graphical user interface (GUI) of a smartphone application, mistyping one's password when logging in, or incorrectly setting access privileges to a part of a website.

We do not consider user errors the commands that users issue, but which trigger no reaction from a system, such as moving the mouse in a text editor over a piece of text. However, if the system reacts to mouse movements, such as in web pages that have HTML elements with the `onMouseOver` property specified [70], then these user commands are subject to user errors, and we target them with our technique.

3.2.2 User Errors Considered in This Thesis

The motto of this thesis, "Software for real, imperfect people," guides our selection of the category of software systems to which we want to apply Record-Mutate-Test. We want to test widely used applications against realistic user errors.

We observe that the Internet, mobile devices, and cloud computing are nowadays ubiquitous, i.e., more than 2 billion people now have mobile broadband Internet access [71], GMail, a cloud-based email application, has more than 425 million users [72], while Facebook touts over 1 billion users [73]. The appeal of cloud-based applications is that they make information easily available everywhere, anytime. We target these applications primarily.

We define cloud-based applications as software as a service (SaaS). SaaS allows a consumer to use applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings [74].

Thus, cloud-based applications have two kinds of users: end users and system administrators. The technique described in this thesis covers errors committed by both groups.

It is important to note that our technique is not particular to cloud-based applications. System developers can employ it to test any kind of system that requires user or system administration input, e.g., password managers, mathematical expression computation, or flashcard applications.

3.2.2.1 End User Errors

End users interact with the entire system through client-side code. Their actions influence the dependability of both the client-side code and of the back-end servers.

The interaction between a user and a system is dynamic: the user devises a plan, and then starts executing the steps corresponding to that plan [10]. In each step, he or she issues a command to the system, observes the outcome, and decides whether he or she is making progress toward completing his or her task [11].

Thus, the success of completing a task depends on each interaction step and on the order in which these steps are performed. An error in one interaction step impacts the success of subsequent interactions. This argues for testing each interaction step.

3.2.2.2 System Administrator Errors

The task of system administrators (sysadmins) is to ensure that the back-end servers are dependable, i.e., that they are reliable, available, safe, and secure. There are several tasks that require administrators to modify a running system, e.g., apply operating system, library, or server-code patches and updates, and configure the servers. System administrators can commit errors when performing any of these tasks.

When modifying the environment in which the server code is running, sysadmins act as end users of that environment. The previous section covers these errors.

In this thesis, we focus on what errors sysadmins commit when they configure a system. An example of misconfiguration is to set incorrect access rights to a website.

The interaction between sysadmins and a server is static, i.e., the administrator presents the server with an already-built configuration file. This argues for injecting errors into the configuration file, rather than in the sequence of actions the sysadmin undertook to generate it.

3.2.3 Definition of Resilience to User Errors

We say that a system is resilient to user errors if they cannot hinder the system's dependability [75]. Dependability is that property of a system which allows reliance to be justifiable placed on the service it delivers. Dependability has four properties: availability, reliability, safety, and security [1].

For example, a user of an e-banking application should be able to transfer money (availability), and tapping the wrong button in the application should not cause the application to crash (reliability), transfer all the user's funds away (safety), or make the user's account publicly accessible (security).

3.3 Verifying Resilience to User Errors

In this thesis we show that a system is resilient to user errors by testing it against realistic user errors. The question that needs be answered now is "How are tests generated?"

To generate and run tests, we use Record-Mutate-Test. Figure 3.1 shows a high-level overview of the technique.

The technique has three phases. First, it *records* the interaction between a user and a system into an interaction trace. An interaction trace is the sequence of events that determines a system's execution, and replaying it generates an execution that mimics the one that lead to the trace's creation.

Next, the technique *mutates* the interaction trace by injecting errors into it and generating new interaction traces. The injected errors correspond to the members of the user error taxonomy based on error manifestation described in Section 2.1.3. They affect either an event or the events' order in an interaction trace.

The newly generated interaction traces are effectively tests and correspond to users making mistakes. Thus, in its final phase, the technique *tests* the system against realistic user errors by replaying the new interaction traces.

It is the injection of realistic user errors into real users' interactions with a system that which makes Record-Mutate-Test practical. Recall that in Chapter 1, we argue that the number of user errors developers need to consider is exponential in the size of the system. Instead, by focusing on real users and their behavior, our technique reduces

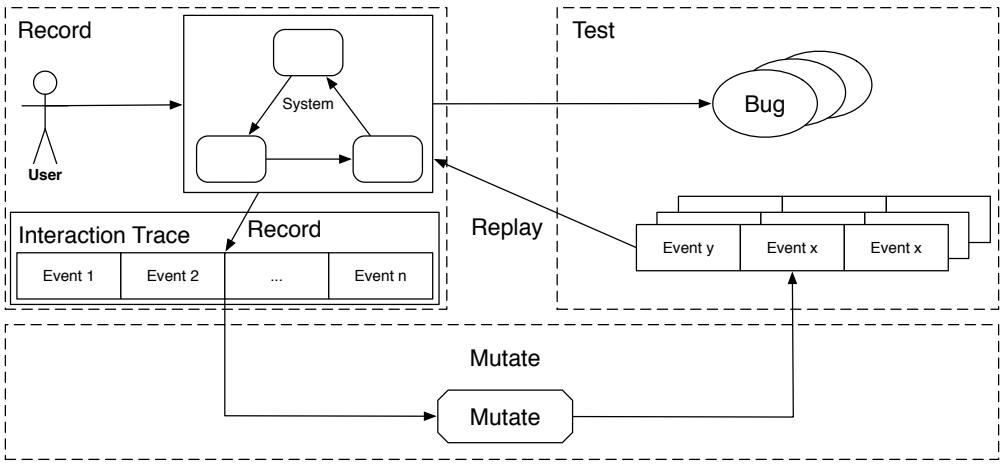


Figure 3.1: A high-level overview of the Record-Mutate-Test technique.

the number of program executions to consider to be proportional to the number of unique ways in which actual users interact with the program.

3.4 Interaction Traces

We define an interaction trace T as a sequence of events, $T =< e_1, \ldots, e_n >$ that describe a program's execution.

Executed program statements, i.e., the sequence of values that the program counter had, represent an obvious choice for events. However, a trace need not contain every executed program statement, because the information they provide is redundant. If the execution of one statement implies the execution of another one, then transmitting the latter one to developers is useless. This is the case for, e.g., basic blocks, where all statements in a basic block are either executed or not, so it suffices to record the first one in the block. The others can be deduced.

For Record-Mutate-Test, the interaction trace contains the sequence of events that determine a program's execution. Each event e_i records one of four sources of "non-determinism" that influence a program's execution:

- User interaction with the program's graphical user interface (GUI)
- Network communication
- Input read from the machine the program runs on
- Decisions made by the runtime system

Re-triggering the events in a trace T, which we call replaying the trace T, should consistently drive the program along the same execution path. Thus, we assume that the trace T is complete, i.e., there are no sources of non-determinism affecting a program's execution apart from those in T (e.g., if the microphone output is not recorded, then we assume the program does not rely on the user providing audio input), and given T, replaying the sequence of events $< e_1, \ldots, e_i >$ brings the program to a state that enables replaying event e_{i+1}.

Even though we are interested in injecting errors only in user-generated events, the interaction trace contains additional events. The reason for their existence is to enable Record-Mutate-Test to replay a test interaction trace up to the point where the injected error manifests. Otherwise, the replay, and thus the test, may deviate if the program

uses non-recorded nondeterministic values to guide its execution. For example, if the list of features available in an application depends on a parameter received from a server, then if during testing actual calls to the server return a different parameter, the tested feature may no longer be available, so the test cannot run to completion.

An event plays one of two roles during replay: *proactive events* cause a feature of the program to execute (e.g., click on the "Send SMS" button), while *reactive events* provide the feature with the input it needs (e.g., the phone number and SMS text). Table 3.1 shows events for each source of non-determinism for interactive Android smartphone applications and maps them to a role.

	User	**Network**	**Device**	**Runtime**
Proactive	Tap, Tilt, Drag, Press key, Scroll	Receive push notification	Trigger geofence	Fire timer
Reactive	Text box value, Selector value, Slider value	Server response	Date, GPS location Camera and mic output, Device settings, Shared data	Asynchronous tasks scheduling

Table 3.1: Trace events for Android applications, classified by covered non-determinism source and proactivity role.

3.5 Mutating Interaction Traces Using User Error Operators

We argue that is impractical to model the psychological conditions that cause users to make mistakes, because we do not have a working model of the human brain in which to inject errors (Section 2.1.3).

We take a pragmatical approach and model the *effects* of user errors, i.e., the way they manifest to the outside world, in a user's interaction with a system. We define four basic user error operators: *insert*, *delete*, *replace*, and *swap*. These operators work on interaction traces and affect either a single event or the sequence of events. One can view the interaction trace as a string and user error operators as edit operators. Applying a user error operator to an interaction trace causes the edit distance between the original interaction trace and the mutated one to become 1.

Each user event contains three kinds of information:

- *Action type* specifies how the user interacted with the system, for example by typing in a text.
- *Target* identifies with what did the user interact, for example a text box with the hint "Email address."
- *Parameters* completely determine the user's interaction, for example that the user typed in the text "silviu@pocketcampus.org."

This information enables Record-Mutate-Test to generate realistic user errors, because they incorporate particularities of the error taxonomies described by Norman [11] and Reason [12], e.g., perceptual confusions (Section 2.1.2).

Users can make mistakes that affect each of these components. For example, they long press on a widget, instead of a normal tap (action), or they tap the wrong button (target), or type in the wrong text (parameters).

3.5.1 The *Insert* Operator

The $insert$ user error operator simulates the user performing an extraneous action, which corresponds to introducing a new event in the interaction trace. In Record-Mutate-Test, the extraneous user action is derived from the correct action

by modifying one of its properties, e.g., the target.

Which new event is inserted into the interaction trace is defined by an *alternative* function that takes as input the event and returns a set of events.

Algorithm 1 describes how the *insert* user error operator works. In the algorithm, the size of the interaction trace IT, i.e., the number of events, is represented as $IT.length$. The index of the first event in the trace IT is 0, while the index of the last one is $IT.length - 1$. $IT[i]$ represents the $(i+1)^{th}$ event, while $IT[i:j]$ represents a contiguous sequence of events that starts with the i^{th} event and ends with the j^{th} one. If $i > j$ or $i > IT.length$, then $IT[i:j] = \emptyset$, i.e., the sequence is empty. If $i < 0$, then $IT[i:j] = IT[0:j]$. If $j > IT.length$, then $IT[i:j] = IT[i:IT.length-1]$, i.e., the sequence of events cannot go beyond the limits of the original interaction trace. The plus sign symbolizes concatenation, i.e., $IT[i:k-1] + IT[k] + IT[k+1:j] = IT[i:j]$. Each event has a property $isUserGenerated$ that specifies if the user generated that event and enables the algorithm to identify in which events to inject errors. $Tests$ is a set of interaction traces.

The *insert* operator traverses the interaction trace one event at a time, computes the alternative events to that event, using the *alternative* function, and creates two new interaction traces by adding each alternative before and after the currently processed event. Finally, the algorithm returns all generated traces.

Algorithm 1 The *insert* user error operator.

Require: Interaction trace IT, *alternative* function
  $Tests = \emptyset$
  for each $i \in \{0, 1, \cdots, IT.length - 1\}$ **do**
    $event := IT[i]$
    if $event.isUserGenerated$ **then**
      for each $event' \in alternative(event)$ **do**
        $IT_{before} := IT[0:i-1] + event' + IT[i:IT.length-1]$
        $Tests := Tests \cup \{IT_{before}\}$
        $IT_{after} := IT[0:i] + event' + IT[i+1:IT.length-1]$
        $Tests := Tests \cup \{IT_{after}\}$
      end for
    end if
  end for
  return $Tests$

The use of the *alternative* function makes Record-Mutate-Test extensible. Below we give some examples of *alternative* functions.

One common user error is that of interacting with the wrong UI widget. We define an *alternative* function, called *vicinity*, that simulates this error. We constraint the set of target alternatives to contain those that are in the correct widget's vicinity. Thus, *vicinity* affects the target of a user's action.

Consider the interaction trace $IT =< e_1 := Type\ textbox(id = x)$ "p", $e_2 := Type\ textbox(id = y)$ "h", $e_3 := Tap\ button(id = z) >$. This trace indicates that the user typed in the text box with $id\ x$ the letter "p" (e_1), then typed in the text box with $id\ y$ the letter "h" (e_2), and finally, the user pressed the button with $id\ z$ (e_3). Assume that the two text boxes are 10 pixels one below the other. Thus, $vicinity(e_1) = \{e_1, e_1^1 := Type\ textbox(y)$ "p"$\}$. Using the *insert* operator and the *vicinity* function on only the first event in the trace IT generates the following interaction traces:

- $< e_1, e_1, e_2, e_3 >$

- $< e_1^1, e_1, e_2, e_3 >$
- $< e_1, e_1^1, e_2, e_3 >$

Including the original event in its list of alternatives enables Record-Mutate-Test to simulate users forgetting they performed an action and redoing it.

Errors in action parameters are also common. One example are spelling mistakes. We define the $keyboard\_layout$ $alternative$ function that uses a keyboard layout to identify neighbors of a key and returns those as alternatives.

For our example, a partial definition of the $keyboard\_layout$ function is $keyboard\_layout(e_1) =$ { $Type\ textbox(id = x)$ "O", $Type\ textbox(id = x)$ "0", $Type\ textbox(id = x)$ "l", $Type\ textbox(id = x)$ ";", $Type\ textbox(id = x)$ "[", $Type\ textbox(id = x)$ "$-$"} for a US Mac keyboard. Figure 3.2 shows the set of interaction traces representing the outcome of applying the $insert$ operator and the $keyboard\_layout$ function. For simplicity, the figure shows only the typed characters. The root of the tree in Figure 3.2 is the initial text. Its children are the 6 alternatives for the letter "p". The leafs of the tree are the mutated texts.

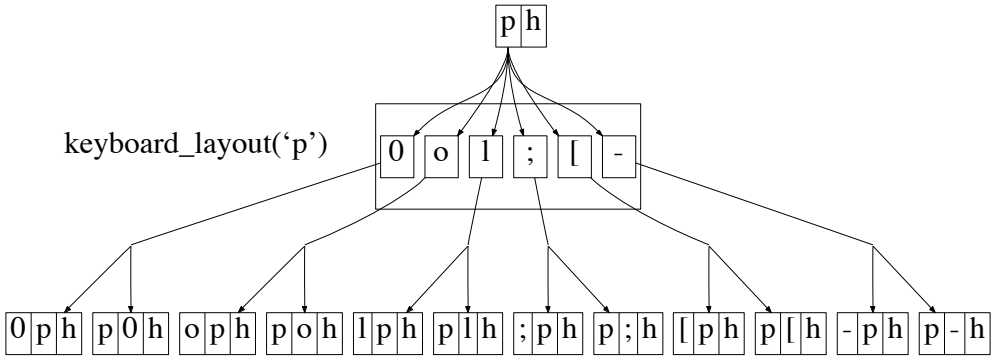


Figure 3.2: Using the $insert$ user error operator with the $keyboard\_layout$ function.

3.5.2 The *Delete* Operator

The $delete$ user error operator simulates a user forgetting to perform an action, which translates into removing an event from an interaction trace. Algorithm 2 describes the way the $delete$ user error operator works. The operator traverses the interaction trace one event at a time and generates a new interaction trace that does not contain the current event. Finally, the algorithm returns all generated traces.

Algorithm 2 The $delete$ user error operator.

Require: Interaction trace IT
  $Tests = \emptyset$
  for each $i \in \{0, 1, \cdots, IT.length - 1\}$ **do**
    if $IT[i].isUserGenerated$ **then**
      $IT_{delete} := IT[0 : i - 1] + IT[i + 1 : IT.length - 1]$
      $Tests := Tests \cup \{IT_{delete}\}$
    end if
  end for
  return $Tests$

Figure 3.3 shows the set of interaction traces representing the outcome of applying the *delete* operator to the example interaction trace.

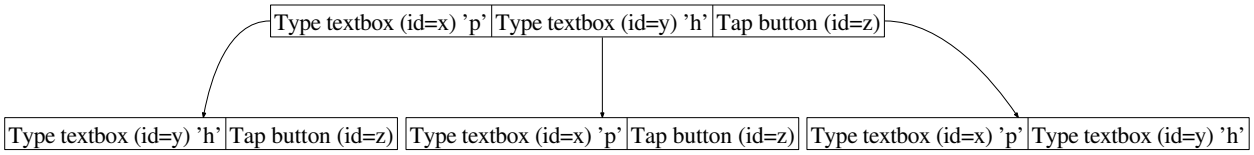


Figure 3.3: Using the *delete* user error operator on the example interaction trace.

3.5.3 The *Replace* Operator

The *replace* user error operator corresponds to the user executing a different event than the one from the interaction trace, i.e., the user substitutes an event for another one. From a modeling point of view, this error is the combination of two user error operators: *insert* followed by *delete*, or vice-versa.

Similar to the *insert* operator, the replacement event is derived from the replaced event. Therefore, Record-Mutate-Test can reuse all the *alternative* functions defined for the *insert* operator.

Algorithm 3 describes the way the *replace* user error operator works. The operator traverses the interaction trace one event at a time, replaces the event with each event alternative defined by the *alternative* function, and generates a new interaction trace. Finally, the algorithm returns all generated traces.

Algorithm 3 The *replace* user error operator.

Require: Interaction trace IT, *alternative* function
  $Tests = \emptyset$
  for each $i \in \{0, 1, \cdots, IT.length - 1\}$ **do**
    $event := IT[i]$
    if $event.isUserGenerated$ **then**
      for each $event' \in alternative(event)$ **do**
        $IT_{replace} := IT[0 : i - 1] + event' + IT[i + 1 : IT.length - 1]$
        $Tests := Tests \cup \{IT_{replace}\}$
      end for
    end if
  end for
  return $Tests$

Figure 3.4 shows the set of interaction traces that are the outcome of applying the *replace* operator using the *vicinity* function to the example interaction trace.

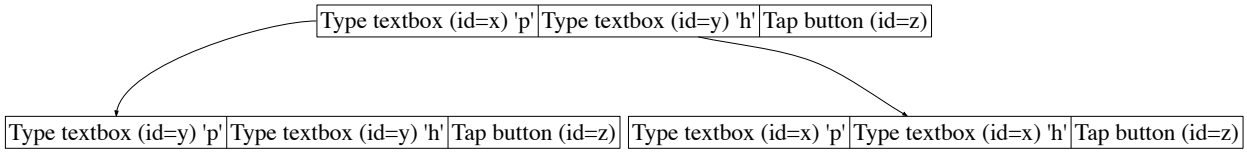


Figure 3.4: Using the *replace* user error operator with the *vicinity* function.

Figure 3.5 shows the set of interaction traces representing the outcome of applying the *replace* operator and the *keyboard_layout* function to the example interaction trace. For simplicity, the figure shows only how the text changes as a result of replacing letter "p" with its alternatives. The root of the tree is the initial text. Its children are the 6 alternatives for the letter "p". The leafs of the tree are the mutated texts.

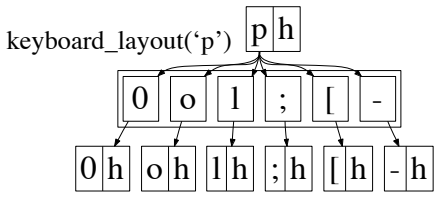


Figure 3.5: Using the *replace* user error operator with the *keyboard_layout* function.

We define two additional *alternative* functions that change the text input by a user. The first function, named *change_case*, identifies the state of the modifier keys (i.e., `Shift` and `Alt`) needed to generate the input character and alters their state. Applying this function to, e.g., the character p returns the set $\{P, \pi, \Pi\}$ for a US Mac keyboard, corresponding to pressing `Shift + p`, `Alt + p`, and `Alt + Shift + p`, respectively.

Figure 3.6 shows the set of interaction traces that is the outcome of applying the *replace* operator and the *change_case* function to the example interaction trace. For simplicity, the figure shows only the typed characters. The root of the tree is the initial text. Its children are the 3 alternatives for the letter "p". The leafs of the tree are the mutated texts.

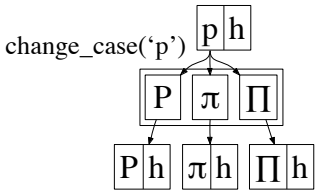


Figure 3.6: Using the *replace* user error operator with the *change_case* function.

The second *alternative* function is *phoneme* and simulates one's lack of spelling knowledge. We define this function using letters that can correspond to the same sound, e.g., the sound `/s/` can be written as the letter `s` (e.g., **s**ize), but also as `c` (e.g., exer**c**ise), so `s` and `c` are alternatives to each other, and the *phoneme* function replaces an instance of one with the other in the action parameters of a type event.

3.5.4 The *Swap* Operator

The *swap* user error operator reverses the order of two adjacent events in an interaction trace. The effect of applying this operator can be simulated by deleting an event and inserting it back after its successor. Algorithm 4 describes the way this operator works. The operator traverses the interaction trace one event at a time, swaps it with its successor, and generates a new interaction trace.

Figure 3.7 shows the set of interaction traces representing the outcome of applying the *swap* operator to the example interaction trace.

Algorithm 4 The *swap* user error operator.

Require: Interaction trace IT
  $Tests = \emptyset$
  for each $i \in \{0, 1, \cdots, IT.length - 1\}$ **do**
    $event := IT[i]$
    if $event.isUserGenerated$ **then**
      $event' := IT[i + 1]$
      $IT_{swap} := IT[0 : i - 1] + event' + event + IT[i + 2 : IT.length - 1]$
      $Tests := Tests \cup \{IT_{swap}\}$
    end if
  end for
  return $Tests$

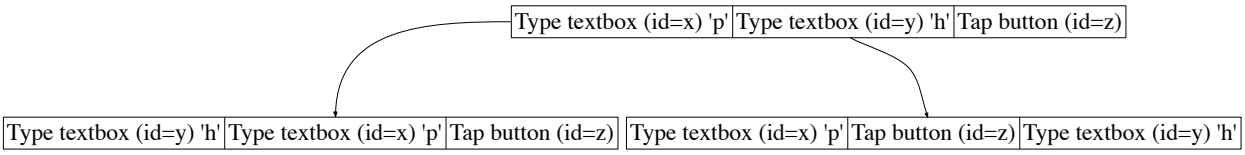


Figure 3.7: Using the *swap* user error operator.

3.6 Reproducing Realistic User Errors by Mutating the Interaction Tree

Up until now, each user error operator mutated either a single event or two consecutive ones, i.e., the *swap* operator. In this section, we describe an extension to Record-Mutate-Test that enables it to mutate sequences of events. The key idea is to abstract a sequence of events and treat it as a single entity. Figure 3.8 shows the added phases.

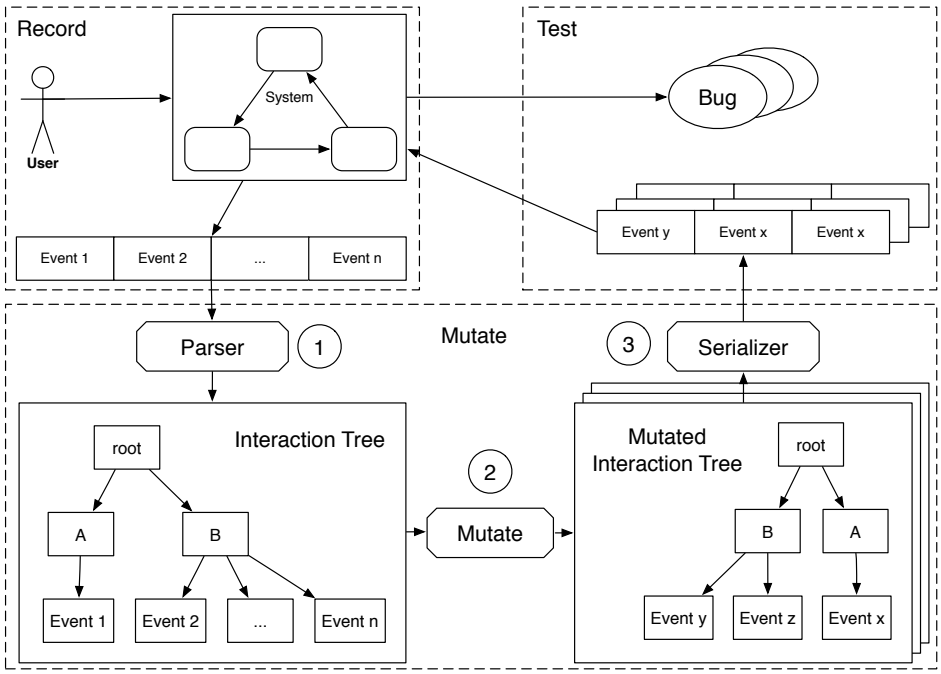


Figure 3.8: Extended version of the Record-Mutate-Test technique.

The extension implies organizing the events in an interaction trace into a hierarchy ①, which we call an interaction tree, applying the user error operators to the tree ②, and converting back the interaction tree to an interaction trace ③.

In the rest of this section, we define the interaction tree, show how to automatically build it using parsers, describe how user error operators work on the interaction tree, and how the tree is converted back to an interaction trace.

3.6.1 Representing User Interaction as an Interaction Tree

The interaction between a user and a system can be viewed at multiple levels of abstraction. Consider the task of editing a Google Sites web page. For people familiar with how Google Sites works, the name of the task is clear and they know what to do to complete the task.

However, if one were to explain this process to people familiar with using web applications, e.g., web-based email, but not with Google Sites, one would break down the task into steps, i.e., log in, edit the page, and save the changes.

One can describe each step in further details. For example, to sign in, one enters their username and password, then clicks on the button labeled "Sign in." To edit the website, one clicks on the button labeled "Edit," and so on.

Finally, for novices, one describes each step as a sequence of mouse clicks and keystrokes. For example, to enter one's username, one clicks on the text box next to the label "username," then presses the keys defining their username.

Refining the description of each task into a sequence of subtasks creates a hierarchy. Its root is the most abstract definition of what the actions described by its leaves accomplish. We call this hierarchy the *interaction tree*. Figure 3.9 shows part of this interaction tree for the task of editing a Google Sites web page.

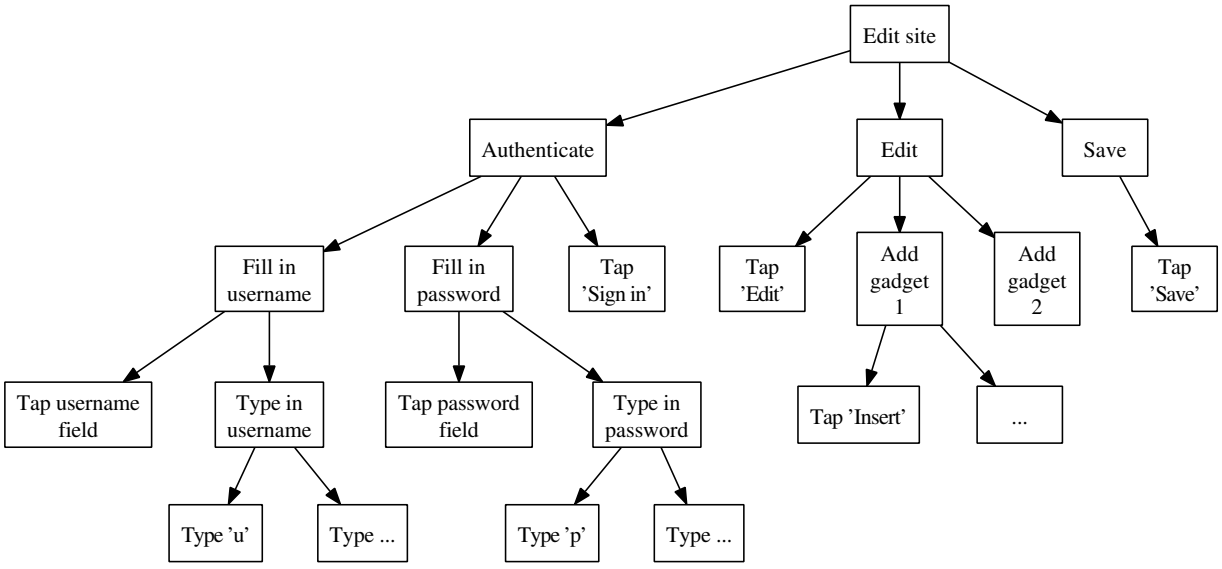


Figure 3.9: The interaction tree for the task of editing a Google Sites website.

Formally, we define an interaction tree $I = (V, E)$ to be a set of vertices V and edges E. We define the relation $parent(v_i) = v_j$ if $v_j \rightarrow v_i \in E$, which determines the parent of a vertex.

Note that user errors can occur at actions described at every level in the interaction tree. For example, a user might forget that they have to save their changes, add the two gadgets in the reverse order, click on the password field instead of the username field, or they can make a typo in their username.

Using an interaction tree for error injection enables Record-Mutate-Test to inject higher-level errors. For example, consider the situation in which a single error is injected. Applying the user error operators on the original interaction trace cannot simulate the error of logging in with a different, but valid, Google account. One may argue that the same effect can be obtained by typos in the username. But, this is not the case. In the first case, the user exists, but may not have the right to edit the website, which is treated by the system, arguably, differently than a non-existing user, as is the situation in the second case.

3.6.2 Manually Defining the Interaction Tree

Task analysis is a technique to understand how people complete tasks using computers and can identify possibilities for improving a user's performance, by reducing the time needed to complete a task and by limiting the impact incorrect user performance has on the dependability of a system [9].

One can use task analysis techniques to *manually* define the interaction tree, as Section 3.6.2.1 and Section 3.6.2.2 show, because they pursue a hierarchical breakdown of the activities and actions that users perform.

Task analysis techniques are interesting to review because they provide the ground truth for what an interaction tree should be. This is because they involve a specialist interviewing and observing actual users interacting with a system, and who can reason about the structure of a task by using more than just the sequence of actions the user performed, e.g., personal experience or knowledge about what the system does and how it is supposed to be used.

3.6.2.1 Hierarchical Task Analysis

Hierarchical Task Analysis (HTA) [9] is a systematic method of describing how a user's actions are organized in order to meet a goal, i.e., to complete a task. The analysis starts with a top level goal the user wishes to achieve. The analysis then refines the top goal into sub-goals. The refinement continues until the analysis meets a stopping criterion, e.g., the goals are atomic operations, such as clicking using the mouse or typing on the keyboard. These goals correspond to the user-generated events in the interaction trace. Stopping the analysis at this level enables one to construct an interaction tree. It is important to note that HTA is not an algorithm, but a set of guidelines on how to break a goal into the steps necessary to achieve it.

A plan defines the order in which to achieve each subgoal and execute each step. This plan is a manually-defined interaction tree. Figure 3.10 shows part of the tree generated using HTA for the task of ordering a book from Amazon [76]. The second-from-top goals show the steps involved in the purchase: locate book, add it to the shopping basket, fill in the payment details, complete address, and confirm the order.

HTA decomposes goals assuming expert users who are rational and do not commit errors. In this thesis, we leverage its principles to generate realistic erroneous user behavior tests.

3.6.2.2 Goals, Operators, Methods, and Selection Rules

Goals, Operators, Methods, and Selection Rules (GOMS) [9] is a task analysis technique to describe the procedures that a user must know in order to operate a system.

GOMS is similar to HTA in that it provides a hierarchical description of a user's tasks. However, in GOMS a task is broke down into goals, operators, methods, and selection rules. As in HTA, goals define what the user is trying to achieve. Operators are either what the user does to the system to achieve the goal or what the user has to remember or reason about. Example of operators are: press a button, remember a symbol, or look up a certain text on a page.

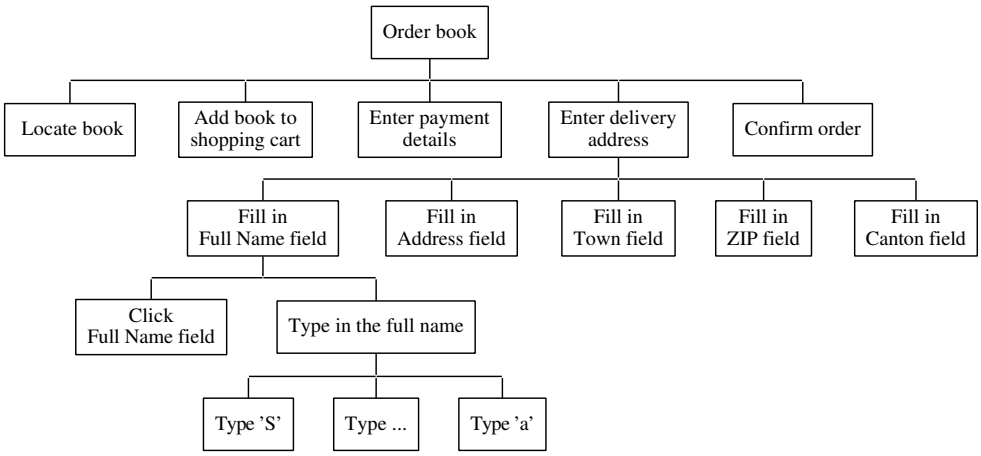


Figure 3.10: A Hierarchical Task Analysis breakdown of the task of ordering a book from Amazon.

Methods are a combination of operators that achieve a sub-goal. Methods in GOMS are akin to functions in programming languages. Selection rules determine which of multiple competing methods, which all lead to achieving the goal, to pursue.

A classic example is that of deleting a file from one's hard drive (Figure 3.11). This task involves two steps. First, one must select the file to delete, then one must issue the delete command. There are two rules to select the file, assuming one is using a graphical user interface, e.g., Finder in Mac OS: if the file is far from the beginning of the file list, then one must use the mouse to select the file, to save time. Otherwise, one can just navigate to the file using the arrow keys.

There are three rules to decide how to invoke the delete command: 1) if one's hands are on the keyboard, then one presses the corresponding key; 2) if the trash icon is visible, then one drags the file to the trash; and 3) if none of the previous two situations happen, then one has to use the right-click method of invoking the delete command.

Wood [2] shows that GOMS can be used to detect sources of errors in a design. The detection is based on analyzing the memory load imposed by a task on a user and on identifying error patterns in the GOMS model. Wood also proposed GOMS be extended to account for errors, by adding an exception handler that is triggered when the user realizes they made an error.

Both task analysis techniques generate a hierarchical description of the interaction between a user and a software system. Ideally, Record-Mutate-Test would inject errors into this description. However, applying these techniques requires human effort, which is contrary to our goal of *automatically* testing software systems against realistic user errors. Instead, we use the interaction tree as a proxy for the output of task analysis techniques.

3.6.3 Automatically Reconstructing the Interaction Tree

In this section, we describe algorithms to automatically construct an interaction tree from the events in an interaction trace. The purpose of the algorithms is to generate interaction trees that are similar to the hierarchical descriptions one would generate by applying the techniques described in the previous section. The algorithms try to automate applying the principles guiding task analysis techniques, but in reverse, because while task analysis techniques know what the analyzed task is, our algorithms reconstruct this task. We call these algorithms parsers and they correspond to Step ① in Figure 3.8.

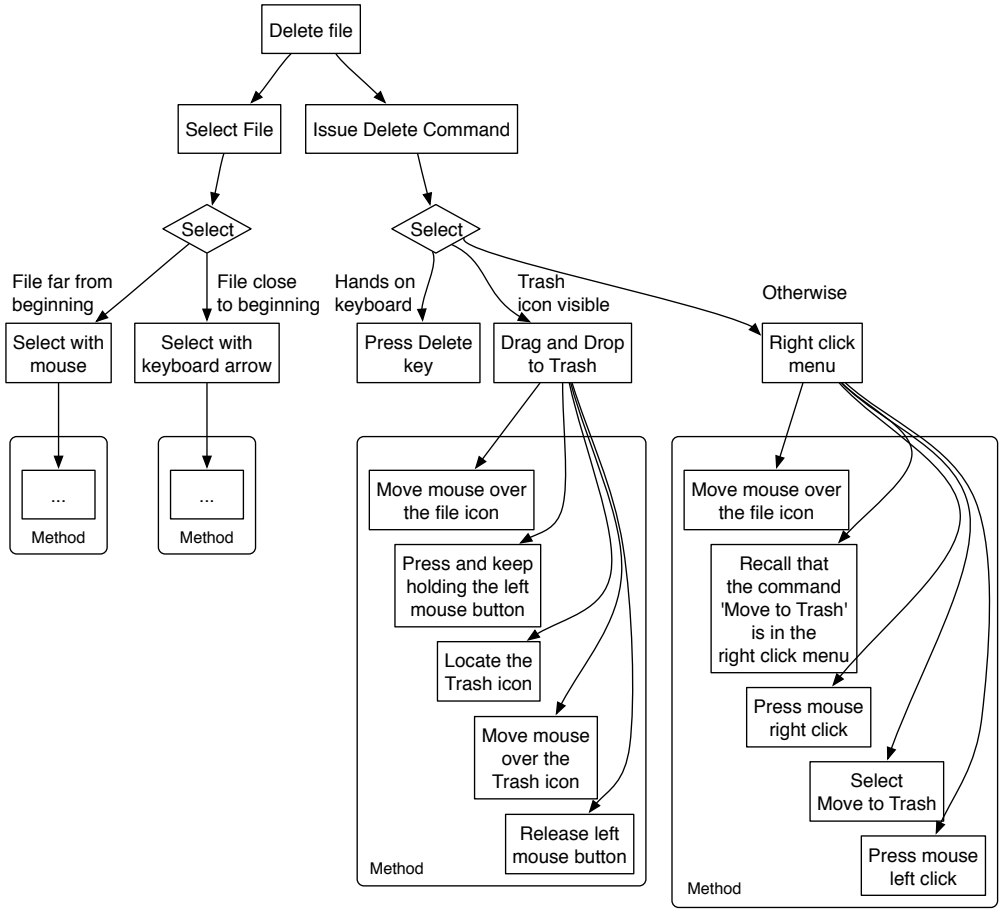


Figure 3.11: A GOMS decomposition of the task of deleting a file.

3.6.3.1 Reconstructing the Interaction Tree from End Users' Interactions

To reconstruct the interaction tree from an interaction trace, we leverage the observation that high performance and good UI and user experience design requirements determine developers to segregate the actions a user can perform into multiple screens [77]. For example, in an e-commerce website like Amazon, each step in the process of buying an item is a new page. Another example is the concept of storyboard adopted by iOS. A storyboard is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens [78]. Finally, a common pattern is for programs to provide menus that enable users to access different functionalities.

For Record-Mutate-Test, each screen is an inner node in the interaction tree, and the user events recorded in the interaction trace are children of these screens.

Figure 3.12 shows an example of a reconstructed interaction tree from an interaction trace. The interaction trace corresponds to writing an email using GMail.

3.6.3.2 Reconstructing the Interaction Tree from System Administrators' Interactions

While Record-Mutate-Test is general, we must customize it for testing a system against realistic system administrator configuration errors. In this case, the technique injects errors into the configuration file, rather than in the process of

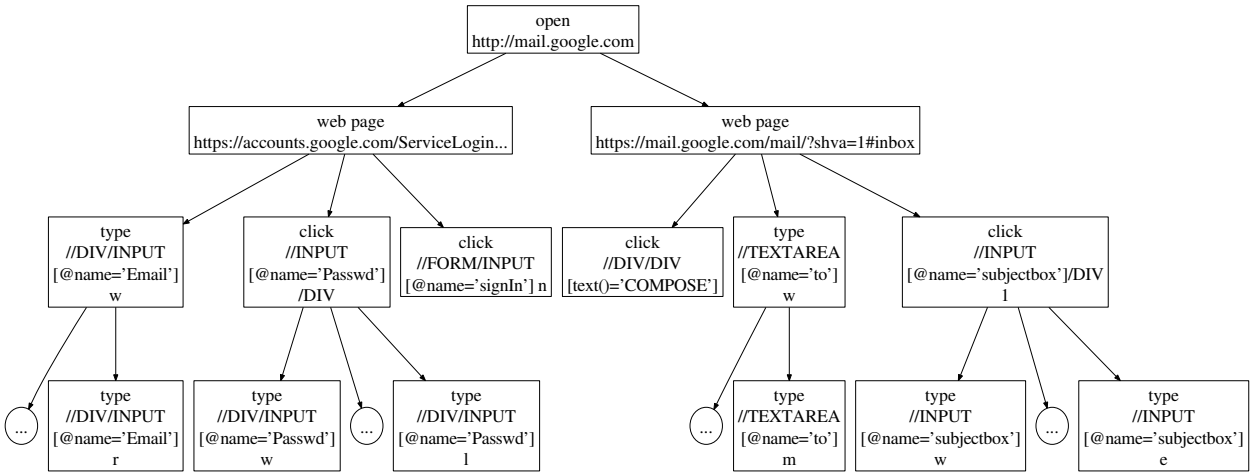


Figure 3.12: A reconstructed interaction tree.

constructing that configuration file.

One could argue that the technique can target the sequence of actions the system administrator undertook to build the configuration file. However, we argue that this is unnecessary, because it is not the sequence of actions that dictates the correctness of a configuration file, rather it is its content. For example, it is irrelevant, from the server's point of view, if when editing the configuration file, directive X was added before directive Y or not.

Instead, Record-Mutate-Test injects realistic system administrator errors in a configuration file. We define a configuration file as a list of configuration parameters, called directives, that can be grouped into sections. Thus, a configuration file is a hierarchy, so it is suitable to be an interaction tree. If the configuration file format does not support sections, then directives are children of the configuration file root node, but the hierarchical structure is preserved. A directive is a triple name, separator, value. This hierarchical configuration structure is common to multiple systems, e.g., database management systems like MySQL or PostgreSQL and the Apache and Lighttpd web servers.

While each system may have their unique syntax for their configuration file, after we remove the peculiarities of the system, such as configuration file format or how to refer to configuration variables, we are left with a hierarchical structure. We define the interaction tree to be this salient hierarchy.

Figure 3.13 shows a stylized configuration file, while Figure 3.14 presents the graphical representation of the corresponding interaction tree.

```
//file: c.conf
[server]
  mem = 500MB
  log = ON
```

Figure 3.13: An example of a configuration file.

Each node v of the interaction tree has a type $v.type$ and a value $v.value$. There are seven node types: *name*, *separator*, *value*, *directive*, *section*, *configuration file*, and *configuration*. Figure 3.14 represents a node's type as the shape of that node, i.e., hexagon for *configuration* nodes, pentagons for *configuration file* nodes, circle for *name* nodes, trapezoid for *section* nodes, rectangle for *directive* nodes, triangle for *separator* nodes, and diamond for *value* nodes.

The root of the interaction tree is a node of type *configuration*. A *configuration* node connects to one or multiple

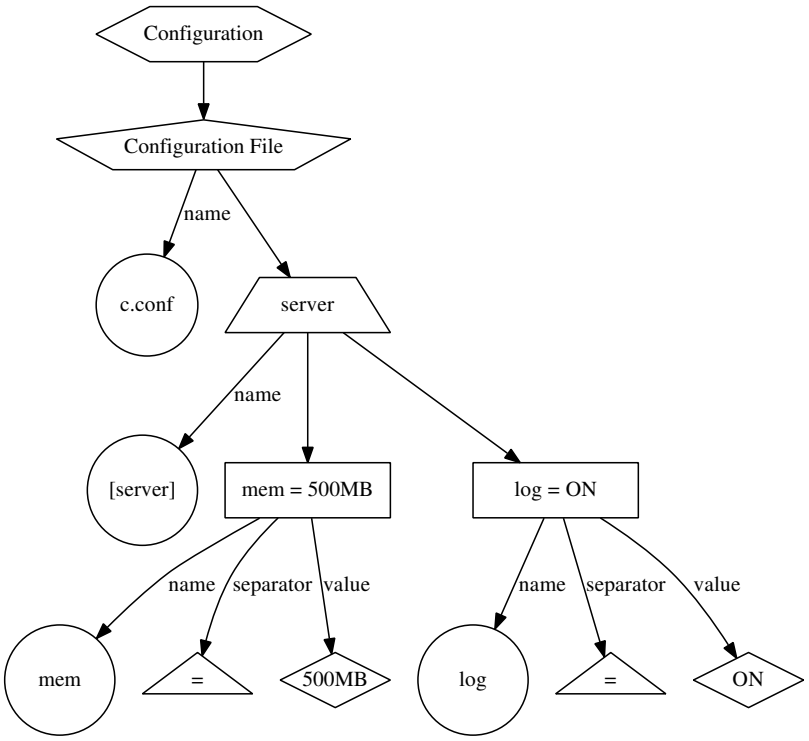


Figure 3.14: The interaction tree of the sample configuration from Figure 3.13.

configuration file nodes, because a system may need more than a single configuration file.

Next, a *configuration file* node connects to multiple *section* nodes and to a leaf node denoting that file's name. Each *section* node is connected to multiple *directive* nodes and to a leaf node denoting the section's name. Finally, each *directive* node is connected to nodes that represent that directive's *name*, *separator*, and *value* attributes.

Record-Mutate-Test uses configuration file parsers that read in a configuration file and output the interaction tree. Parsers can be either system-agnostic, as for line-oriented configurations such as MySQL's, or system specific, as is the case for Apache. By using parsers, Record-Test-Modify is portable across different systems.

Assigning types to various configuration components enables the development of specialized $alternative$ functions, such as one where separators from one system are used for another system, e.g., using Apache httpd's " " separator when configuring Lighttpd, which uses "=" as separator.

3.6.4 Converting an Interaction Tree back to an Interaction Trace

To convert an interaction tree I back into an interaction trace, which can then be replayed against a system, Record-Mutate-Test traverses the tree in pre-order, i.e., it inserts in the trace the current node of the tree, if the node is an event from an interaction trace, then traverses the left subtree, and then the right subtree.

3.6.5 Modeling User Errors as Mutations to the Interaction Tree

A fundamental assumption of this thesis is that the models that describe how users make mistakes are applicable to all vertices at all levels in the interaction tree. This section describes an extension to the Record-Mutate-Test technique

that enables it to apply the user error operators defined in Section 3.5 to an interaction tree.

For Record-Mutate-Test to reuse the user error operators on interaction trees, we extend the operators to work on lists of vertices and express the interaction tree as a set of lists, $L(I = (V, E))$, in which every list represents the children of a vertex. Figure 3.15 shows how the interaction tree in Figure 3.9 is split into a set of lists L.

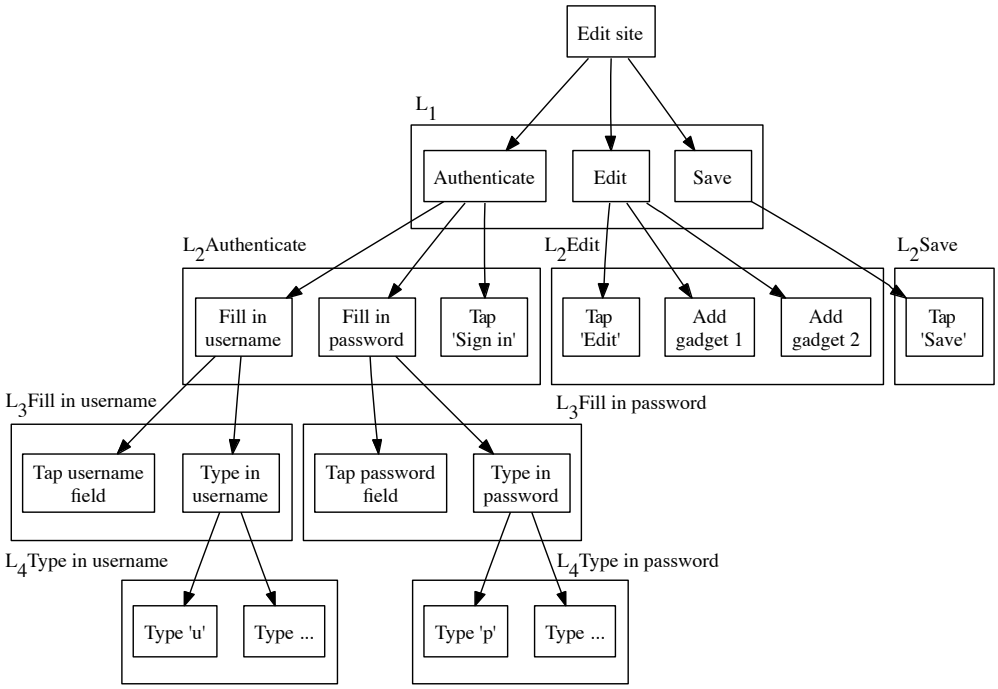


Figure 3.15: Splitting the interaction tree in Figure 3.9 into its lists.

Now, the error injection process can be formalized as shown in Algorithm 5. The error injection algorithm takes as input an interaction tree I and applies each user error operator o on each list l in the set of lists $L(I)$ to generate the list $l' = o(l)$, which Record-Mutate-Test uses to replace the original list l, to generate a new interaction tree I'.

Algorithm 5 Extension of the Mutate phase to inject errors into an interaction tree.

Require: $Interaction\ tree\ I, User\ error\ operators\ O$
  $Tests = \emptyset$
  for each $l \in L(I)$ **do**
    for each $o \in O$ **do**
      $l' = o(l)$
      $I' = I.replace(l, l')$
      $Tests = Tests \cup \{I'\}$
    end for
  end for
  return $Tests$

3.6.5.1 General *alternative* Functions

Expressing the interaction between a user and a system as a tree enables Record-Mutate-Test to easily model complex error manifestations described by Hollnagel [10]. These errors can be modeled by using general $alternative$ functions.

In the case of the side-tracking users error, the expected progression of a user's plan is replaced by another part of the same plan. We model this error by using the $side\text{-}tracking$ function, which is defined as $side\text{-}tracking(vertex) = \{alternative |\ alternative \neq vertex \wedge\ parent(vertex) = parent(alternative)\}$. That is, the alternatives of a vertex are its siblings. The $side\text{-}tracking$ function applies to every plan, sub-plan, and atomic operation.

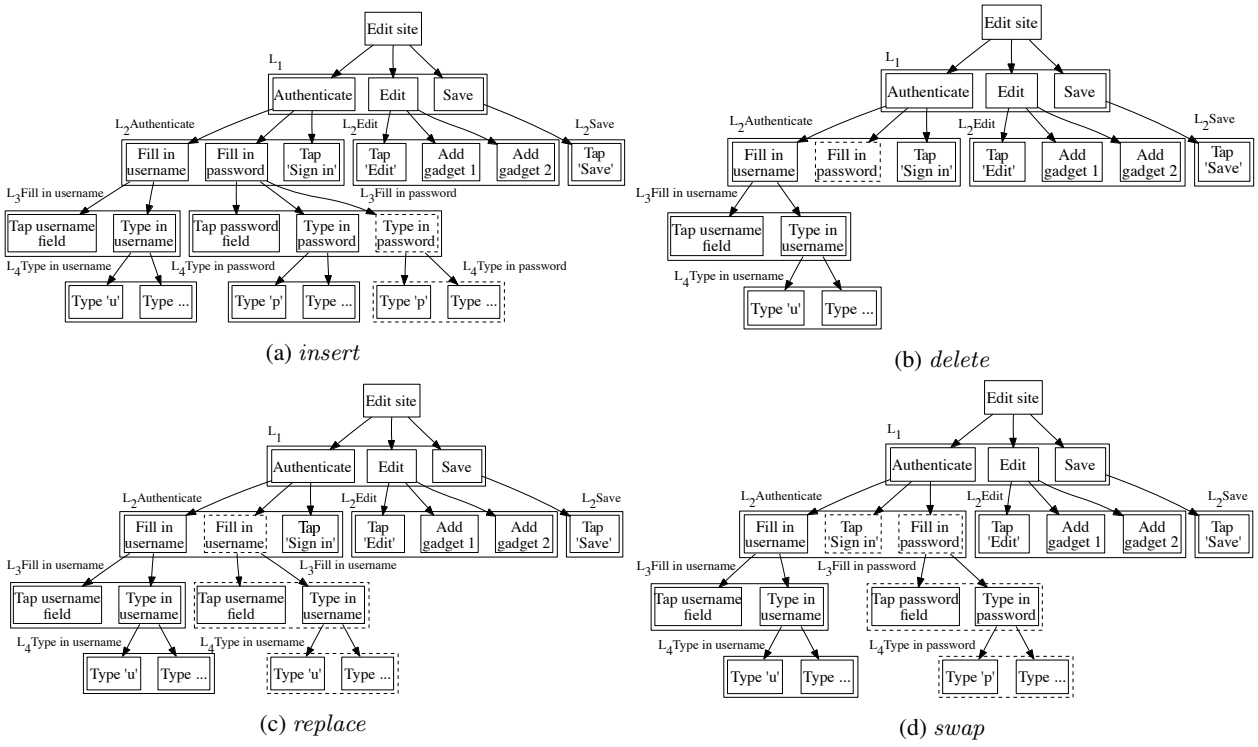
Another user error is capture, where the execution of one plan is replaced by that of another plan [10]. Hollnagel describes a special form of capture, called branching, in which there exist multiple plans that share a common prefix, and the user starts to follow the more frequently-used plan, even though it is inappropriate for the current situation.

We define two branching $alternative$ functions. First, the $local\text{-}branch$ function looks at the interaction tree and checks whether an inner node appears multiple times in the tree. In this case, $local\text{-}branch$ returns the other instances of the node as alternatives. For example, consider an interaction trace that corresponds to a user writing two emails from two different GMail accounts. In this case, filling in the "To" field in an email appears twice in the interaction trace, e.g., as vertices v_1 and v_2. The $local\text{-}branch$ function returns v_2 when v_1 is passed as parameter, and vice-versa. Using this function generates tests that swap the addresses to send emails to.

Second, the $general\text{-}branch\ alternative$ function applies the same principle to different interaction traces. For the previous example, $general\text{-}branch$ enables Record-Mutate-Test to generate the same tests even if there are two interaction traces, one for each email.

These $alternative$ functions, which work on vertices, can be combined with the previously defined $alternative$ functions that work only with events.

Figure 3.16 shows four examples of applying the four user error operators on the interaction tree.



(a) *insert*

(b) *delete*

(c) *replace*

(d) *swap*

Figure 3.16: Example of applying the user error operators to the interaction tree defined in Figure 3.15.

3.7 Leveraging End Users to Test Software Systems

Successful software systems are used billion times a day by hundreds of million of people. For example, the average number of daily Facebook users is about 700 million, and they generate about 4.5 billion Facebook likes [79]. The Google Chrome browser has 310 million active users [80].

We believe users are an untapped resource for testing a system. We envision users contributing in two ways to the testing process. First, their interaction with a system can used to derive tests for the system. Second, the users' devices can be used to run tests.

We propose a crowdsourced testing platform that leverages the multitude of a program's end users *and* incorporates Record-Mutate-Test. We call this platform ReMuTe.

Figure 3.17 shows how the platform works. ReMuTe collects interaction traces from real users, then uses Record-Mutate-Test to generate new traces, distributes these new traces to multiple users who replay them, and collects the results. Users can participate in any of these steps.

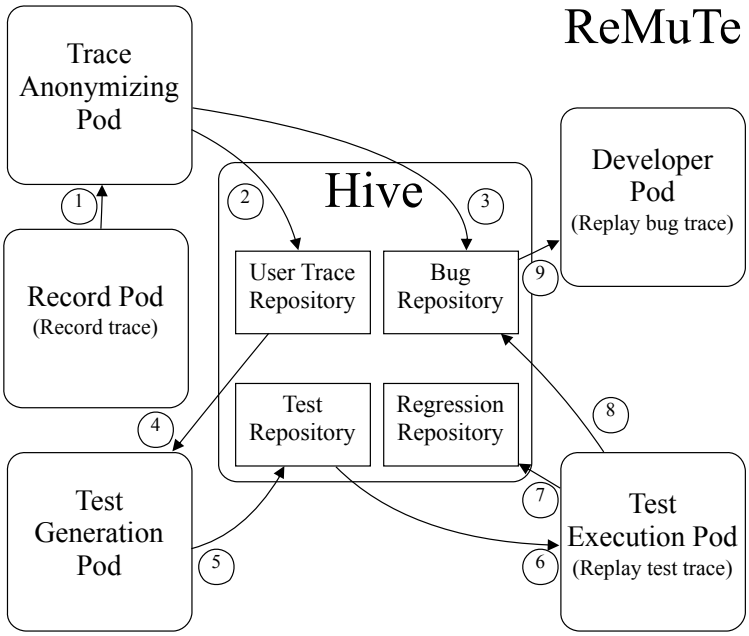


Figure 3.17: ReMuTe architecture.

ReMuTe is composed of a *hive* and five types of *pods*. Pods run inside programs and monitor or enforce a program's execution. The hive is hosted in the developers' computing infrastructure and acts as a mediator between pods and saves the information relayed to it to persistent storage. The hive stores interaction traces into 4 repositories:

- The *User Trace Repository* contains interaction traces that originate from users and will be used as seeds for test generation
- The *Bug Repository* contains traces that cause bugs to manifest
- The *Test Repository* contains generated tests
- The *Regression Repository* contains regression tests

Programs based on ReMuTe contain five types of pods, serving different purposes, that one can think of as plugins. They do not add new functionality to the program, rather enable users to choose the way they take part in the crowdsourced testing of that program. Alternatively, developers can run programs with some of the pods in virtual machines.

The basic pod is the *Record* pod that records *interaction traces*. Interaction traces are self consistent, so they can be replayed on any machine. We envision programs having a *Contribute trace* button that when pressed ships the current trace to the hive. Users can specify whether ReMuTe should use the trace as source for test generation, and the hive places it in the *User Trace Repository* ②, or if it triggers a bug, the hive places it in the *Bug Repository* ③.

A barrier to users contributing traces is their having to share private information contained in the trace with other users or the developers. To protect users' anonymity, traces need to be anonymized ①. The *Trace Anonymizing* pod enables users to review an interaction trace, ask ReMuTe to anonymize the trace, review the anonymized trace, and get information about how much personally-identifiable information the anonymized trace contains. Finally, if the user is satisfied with the anonymized trace, (s)he accepts ReMuTe's proposal of shipping the trace to the ReMuTe hive. Thus, in ReMuTe, no trace that uniquely identifies a user ever gets added to one of the hive repositories.

Once a non-bug-triggering interaction trace has been submitted, *Test Generation* pods use it as a seed for new tests that mimic the way users make mistakes ④. These pods correspond to the Mutate step in Record-Mutate-Test. The new traces are added to the *Tests Repository* ⑤.

Test Execution pods retrieve traces from the tests repository, replay them, check for bugs ⑥, and then submit them to the appropriate repository, i.e., the *Regression Repository* if the tests ran successfully and their output is correct ⑦ or the *Bug Repository* ⑧, in the opposite case. These pods use a replay mechanism to run tests that is able to control the communication between the program and the external world: up to the point where they diverge from their source trace, tests run in full replay mode with all their inputs controlled by the replay infrastructure. The program never interacts with the external world. After passing the divergence point, the program is allowed to communicate with the external world. Otherwise, recorded communication may be unsuitable for the new program execution and cause the program to wrongfully fail.

When developers wish to reproduce a bug, they use the *Developer* pod that displays the traces in the *Bug Repository*, allows developers to select one, and then replays the chosen trace ⑨. In this case, the replay infrastructure completely isolates the program from the external world.

3.8 Recording and Replaying Interaction Traces from Real-World Users

A key element of Record-Mutate-Test is its ability to record the interaction between users and a program and between that program and its environment and replay it in a way that consistently mimics the original program execution. In this section, we describe the requirements for a suitable record and replay solution.

We deem the following properties to be of interest:

- Ease of deployment. We wish for everyday users to use programs based on ReMuTe, so it is important that the record and replay solution is easy to deploy, in that it requires no changes to a user's environment.

- Interaction traces have test usability. This thesis concerns itself with generating tests that simulate realistic user errors. A trace has test usability if it can be used to generate tests, and the events it contains refer to actions performed by real-world users.

- Reduced interaction trace size. We wish to apply ReMuTe to various types of systems and programs. Of particular interest is testing smartphone applications, which, compared to traditional computers, have limitations in battery consumption. So the smaller the traces, the cheaper it is to send them to the hive, both in energy consumption and data traffic charges.

- Replay portability. This property refers to the possibility of recording an interaction trace on one device, then modify it, and replay it on a different device. Replay portability assesses how sensitive a recorded trace is to changes in the environment of the program whose execution it describes. This is an important property because the landscape of devices users possess is varied. So, if a trace is not portable, the tests generated from it cannot be universally replayed. This causes the technique to lose efficiency, because it will have to inject the same error, but in traces recorded on different devices. This generates redundant tests.

The ideal record and replay solution is one that is easy to deploy, its interaction traces can be easily used to generate tests, are of small size, and independent of the device they were generated on.

Table 3.2 presents five possible record and replay solutions. The table sums up how these solutions fair with respect to our evaluation criteria.

Property / Technique	**Virtual Machine**	**Operating System**	**Runtime System**	**Branches**	**Events**
Ease of deployment	Difficult	Difficult	Difficult	Easy	Easy
Interaction traces have test usability	Reduced	Reduced	Reduced	Reduced	High
Recorded trace size	8-138 MB/h	0.1-0.9 MB/h	$<$100 MB/h	43 GB/h	0.2 MB/h
Replay portability	Not needed	Reduced	Reduced	Medium	High

Table 3.2: Record and replay techniques compared with respect to evaluation criteria relevant to generating tests.

The first solution is to place an entire operating system and the programs running on top of it inside a virtual machine and record the interaction between the virtual machine and the virtual machine monitor, e.g., ReVirt [50].

The second record and replay technique is to modify the operating system to log the result of operating system calls, e.g., Scribe [81].

The third solution is to change the runtime system to provide record and replay functionality, as one can do for the Java Virtual Machine, as in ORDER [82]. ORDER proposes to track the order in which threads access an object, not an object's fields or a specific memory address.

The fourth solution is to record what branches the program took during its execution, and then reconstruct program inputs that generate the same program execution and enable replay.

The fifth solution is to record the interaction between a program and its environment, but from inside the program.

Techniques where the recording mechanism is placed in a virtual machine, the operating system, or the runtime system require changes to the user's environment. Therefore, they are inapplicable to collect traces from programs whose users do not know how or are not allowed to make the necessary changes to their system.

Second, for developing tests, Record-Mutate-Test needs to process events triggered by users. Alas, such techniques do not provide traces that can be easily mapped to users' actions, because they record the effect of a user's interaction. For example, virtual machine-based solutions record what CPU instructions were executed in response to a user touching the screen of a device.

Third, these techniques may lead to generating traces of considerable size.

Finally, they suffer from poor replay portability, as the execution traces they produce can only be replayed on a system with the same configuration as the one the trace was recorded on, e.g., same CPU and operating system. However, we envision to use ReMuTe to programs that run on a plethora of devices, with different CPUs and OS versions. We could develop multiple implementations of ReMuTe, one specific for each platform, but this segregates the user population into multiple sub-domains, each one having fewer members. This negatively impacts the anonymity ReMuTe can provide to its users, as we will see in Chapter 4.

We conclude that techniques that rely on virtual machines or on modifications of the operating system or the runtime system are inappropriate for ReMuTe's *Record* and *Replay* pods.

A second type of record and replay solutions is one in which one instruments a program to log certain events. One such record and replay technique is to record the outcome of the branch statements executed by a program, which can then be used to recover the program inputs that replay that execution. However, such a technique is not amenable to generating the kind of tests we are interested in, for the same reasons as the previous techniques.

The other possibility is to record the communication between a program and its environment. For user interaction, this communication is represented as the execution of user-triggered events. User-facing programs are event driven. They wait for the user to perform an interaction and then they react to it. A common way to handle user interaction is for programs to register callbacks with their runtime, and when an event occurs, the runtime invokes the callback. Such a paradigm is common to Android, iPhone, and web applications. Furthermore, when performing communications with the external world, programs make use of predefined APIs, on which ReMuTe can interpose.

This means that there is a clear boundary that one can interpose on in order to record and replay a program's execution. Furthermore, one can perform this recording from within the program, by using instrumentation techniques [83].

This solution ensures that programs that record interaction traces are easy to deploy, because they do not require changes to the environment they run in, the interaction traces contain events that directly map to how users interacted with the system, making it easy to use them as seeds for tests, they generate small traces, and are independent of the underlying runtime system and, to some degree, to even the internals of a program, i.e., it is possible that interaction traces recorded for one program version can be replayed on a later version of the same program.

We chose this last technique as basis for ReMuTe's *Record* and *Replay* pods.

3.9 Achieving Higher Testing Efficiency

We define efficiency as the ratio between the time it takes to run the tests Record-Mutate-Test generates and their effectiveness. Record-Mutate-Test should generate as few tests as possible without compromising its effectiveness.

Conceptually, users can commit an unbounded number of mistakes, whose simulation requires injecting an unbounded number of errors. Alas, doing so leads to an explosion in the number of generated tests, which translates into large amounts of time spent running the tests. Intuitively, if for each event in a trace one can generate n alternative events and there are k events in a trace, then there are $n \times k$ new traces, i.e., tests. Injecting a second error generates $n \times k$ traces for each of new traces, for a total of $(n \times k)^2$ traces. As we can observe, the number of tests is exponential in the number of injected errors.

To achieve high efficiency, Record-Mutate-Test needs to reduce the time spent on running tests and to prioritize test execution. We identify four means to reduce the time spent on running tests:

- Parallelize test execution. Continuous integration servers, like Jenkins [32], enable developers to run multiple tests in parallel

- Reduce the number of injected errors, while maintaining the technique's effectiveness
- Eliminate infeasible tests that cannot be replayed
- Eliminate redundant tests that explore the same program execution

In the rest of this section, we describe in more details how to reduce the number of injected errors (§3.9.1), how to eliminate infeasible (§3.9.2) or redundant tests (§3.9.3), and how to prioritize tests (§3.9.4).

3.9.1 Reduce Number of Injected Errors

We wish to reduce the number of injected errors without affecting the effectiveness of the generated tests. That is, we aim to inject the minimum number of errors that reproduce most of the errors that a system can practically experience in production.

To determine this minimum number of errors, we run an experiment in which we reproduce common misspellings of English words, as reported by Wikipedia [84]. This report contains a mapping between the incorrect and the correct spelling of a word. We use the user error operators on the correct version and compute how many of the misspellings they reproduce.

The experiment shows that injecting two errors enables Record-Mutate-Test to reproduce 97% of the reported misspellings. Reproducing the remaining ones requires more mutations and/or extending the *alternative* functions.

Thus, we conclude that injecting two errors is sufficient to cover practically all realistic errors.

3.9.2 Eliminate Infeasible Tests

Record-Mutate-Test keeps track of whether it was able to replay an interaction trace or not. If it detects a failure, it saves the sequence of events replayed so far and the event whose replay failed and uses it as an identifier for other tests that will also fail the replay. That is, if there is an interaction trace $IT =< e_0, e_1, \cdots, e_i, \cdots, e_k >$ and Record-Mutate-Test failed to replay event e_i, then the technique saves the prefix $P =< e_0, e_1, \cdots, e_i >$. Next, if it encounters a trace $IT' = P+ < e_j, \cdots, e_m >$, then it does not try to replay the trace IT', because replay will fail. This solution is useful when Record-Mutate-Test injects at least two errors and the first error causes the trace to become unreplayable.

Record-Mutate-Test enables developers to define dependencies between interaction trace events and ensures that the tests it generates do not violate them. A direct example of this are the implicit dependencies defined by the interaction tree that prevent generating a test in which a parent event is swapped with its first child. Such a test will likely be infeasible, because, generally, it is the parent event that makes it possible for the child event to occur. For example, consider the event of pressing the "Compose" button in GMail and the typing in the first letter of the email address to send an email to. Swapping the two events generates an infeasible trace, because one cannot type the email address if the "To" field does not exist, and this field appears as a consequence of tapping the "Compose" button.

System developers can specify other event ordering constraints. For example, we prevent Record-Mutate-Test from generating tests in which the start of an asynchronous request comes *after* the event denoting that request's end. Such traces are unreplayable, so it is useless to generate such tests.

A third means to eliminate redundant tests is to use dynamic program analysis, e.g., concolic execution [85], to detect the set of conditions that program inputs must satisfy to be accepted by the system. By using such techniques, Record-Mutate-Test can generate tests that reach the core functionality of a system and do not get rejected right away. For example, to test software systems against misconfiguration, Record-Mutate-Test learns what constitutes a valid

configuration file. Then, when generating mutated configuration files, the technique ensures that the configuration files it outputs are valid. This means that sysadmin errors get to impact the functionality of a system.

3.9.3 Eliminate Redundant Tests

The first line of defense against redundant tests is to simply use a set to keep track of tests. The user error operators we define can generate multiple identical tests. For example, given the interaction trace $IT =< e_1, e_2 >$, only two user error operators, $delete$ and $replace$, and up to two errors to inject, one obtains the trace $IT' =< e1 >$ both by deleting e_2 from the initial trace and by replacing e_1 with e_2 and deleting one instance of e_2.

Another means to prune redundant tests is to cluster tests in multiple partitions and select the medoid of each partition as the representative of that partition and run only that one test. A partition P_i is defined as a set of tests with the property that each test T_i in the partition P_i is more similar to each other test T_j in the same partition P_i than to any other test T_k in a partition P_k. The medoid of a partition P_i is the test T_m with the property that all other tests T_n in the same partition P_i are more similar to test T_m than to any test T_l in the same partition P_i. Thus, the medoid of a partition can be viewed as the representative of that partition.

We developed a clustering algorithm based on the complete link agglomerative clustering algorithm [86]. This algorithm can be used to generate K clusters, with $K \in [1, number\ of\ generated\ tests]$. We define the distance between two tests as their edit distance (i.e., the Damerau-Levenshtein distance [87]) that is the number of edit operations (i.e., insert, delete, replace, swap) needed to transform one string into another. The Damerau-Levenshtein algorithm uses a fixed cost for each operation, but we modified it to take into account different costs based on the type of each needed edit operation. The cost of each operator is given by its value in Table 3.3.

We imagine that Record-Mutate-Test uses the clustering algorithm in the following scenario. After the technique generates a set of tests, it runs the previous algorithm with $K = 1$, chooses the medoid test T_1 and runs that test. Next, it runs the algorithm with $K = 2$, generates two clusters, and selects their two medoid tests, T_2^1 and T_2^2, and runs them. At each step, Record-Mutate-Test runs only the new medoids. That is, if the test T_1 is the same as the test T_2^1, then in the second round, the technique runs only the test T_2^2. This process continues until developers using Record-Mutate-Test run out of time or there are no more tests to run.

The benefit of using this algorithm is that Record-Mutate-Test runs first tests that are dissimilar, which are likely to cover distinct lines of code and discover different bugs.

3.9.4 Prioritize Likely Errors over Unlikely Ones

Another means to increase Record-Mutate-Test's efficiency is to prioritize tests based on the likelihood of the errors they contain manifesting. This means that developers first test their system against the errors their system is most likely to encounter in practice.

We empirically determine the frequency with which errors manifest. During the experiment described in Section 3.9.1, we kept track of which user error operators were used to reproduce a spelling mistake. We associated a counter to each user error operator. Any time an operator was used, its associated counter was incremented. If two user error operators were used, each counter was incremented with 0.5. Table 3.3 show the values of the counters associated to each operator. Record-Mutate-Test uses these values to rank tests.

User error operator	Counter value
Delete	1856
Insert	1065
Replace	966
Swap	635

Table 3.3: Contribution of user error operators to reproducing most common spelling mistakes in Wikipedia articles.

3.10 Properties of the Proposed Technique

Recall that the properties Record-Mutate-Test must have are:

- *Accuracy*. Record-Mutate-Test should be effective (i.e., find a system's problems in handling user errors), efficient (i.e., require to run a reduced number of tests without hindering the technique's effectiveness), realistic (i.e., use erroneous user behavior that is representative of the behavior of actual users), and relevant (i.e., help developers make best use of their limited testing time, by focusing on important parts of the system).

- *Automation*. Employing Record-Mutate-Test should require minimal developer effort, i.e., developers should not have to specify tests, and they should be able to automatically run tests.

- *Scalability*. Record-Mutate-Test should be able to test software systems that contain millions of lines of code, and it should be able to express user errors independently from the system it is testing.

- *Protect users' anonymity*. Record-Mutate-Test should generate tests that do not identify the user who contributed the seed trace.

In Section 3.9 we described how to make Record-Mutate-Test efficient. We describe how to make Record-Mutate-Test relevant in Chapter 5, and how Record-Mutate-Test protects the users' anonymity in Chapter 4. In this section, we analyze whether Record-Mutate-Test is realistic, automated, and scalable.

Record-Mutate-Test models the manifestations of realistic user errors. It uses four user error operators inspired by the phenotypes of erroneous user actions defined by Hollnagel, meaning they are based on real-world observations. The error operators cover most of the erroneous actions defined by Hollnagel.

The $alternative$ functions, used by the $insert$ and $replace$ user error operators, impact the realism of injected errors. We define $alternative$ functions that simulate practically all typos. We define $alternative$ functions that simulate how users can alter the correct sequence of actions when executing a plan, and $alternative$ functions that are specific to end-users and system administrators.

One can easily develop new $alternative$ functions or new events for interaction traces to enable Record-Mutate-Test to improve its coverage of the injected errors. We will see examples of this in Chapter 6.

Record-Mutate-Test is an automated technique, but developers need to embed it into their applications and feed it interaction traces. The error injection process requires no manual assistance. However, one can conceivably tune the injection process to favor error injection into particular parts of a system or focus on a particular class of user errors.

Record-Modify-Test is scalable, because it is not particular to any system. The usage of user error operators enables the technique to scale in both number of systems it can test and in the size of the tested systems. The underlying observation is that since Record-Mutate-Test works at the interface of a system, it is independent of the system's size. As Chapter 7 shows, we used Record-Mutate-Test for testing smartphone and web applications, and web servers.

3.11 Chapter Summary

This chapter described Record-Mutate-Test, a technique to automatically test systems against realistic user errors. The technique has three steps.

In the first step, the interaction between a user and a system is recorded into an interaction trace. These traces contain events, and replaying these events causes the program to consistently follow the recorded program execution.

In the second step, the events in the interaction trace are organized in a hierarchy we call the interaction tree. This interaction tree is then modified to simulate realistic user errors. Next, the interaction tree is serialized, generating interaction traces, each one corresponding to one injected error. These new traces are effectively tests.

In the final step, the new interaction traces are replayed against the system, and the uncovered bugs are reported to the developers.

A particularity of our technique is that it leverages users as both source of tests and as test execution environments.

Record-Mutate-Test is practical, because it reduces the space of program executions and user errors that developers need to test their system against. The key to its practicality is its focus on realistic erroneous behavior. Otherwise, testing against user errors is equivalent to exercising all program execution paths, which is infeasible. Record-Mutate-Test derives realistic erroneous behavior by leveraging real-user interactions with the system and models of realistic user errors developed by psychologists.

The next chapter describes how ReMuTe maintains the anonymity of users who contribute interaction traces that are used to generate tests.

Chapter 4

A Technique to Preserve User Anonymity

4.1 Introduction

ReMuTe, the crowdsourced version of Record-Mutate-Test, is useful only if users contribute interaction traces. Recall that we envision programs based on ReMuTe to have a *Contribute trace* button that when pressed ships the current interaction trace to the ReMuTe hive. The hive then instructs the *Test generation* pods to read the trace, apply the user error operators defined in Section 3.5, and generate new interaction traces that simulate how users make mistakes.

Anonymity concerns over private information contained in shared interaction traces may prevent users from joining ReMuTe and contributing interaction traces. In this chapter, we describe how ReMuTe overcomes these concerns.

The idea of users sharing information with developers is not new. There exist automated systems that enable end users to report bugs. For example, Microsoft's Windows operating system ships with Windows Error Report, a system for collecting program crash information, that helped developers fix over 5,000 bugs [69].

Alas, existing solutions are not compatible with ReMuTe, because they sacrifice developer productivity to preserve user anonymity: they report some execution information (e.g., backtraces, some memory contents) but forgo the data the program was processing and the execution path it was following. Record-Mutate-Test needs this lost information. We believe that the current trade-off between user anonymity and developer productivity is unnecessary, so we seek to strike a better balance.

The work described in this chapter is closely related to the ReMuTe platform described in Section 3.7. Specifically, this chapter describes the *Trace Anonymizing Pod*.

Recall that in the ReMuTe platform, before users contribute a real-world usage scenario, they have the possibility to review the interaction trace, ask ReMuTe to anonymize the trace, review the anonymized trace, and get information about how much personally-identifiable information the anonymized trace contains. Finally, if the user accepts, ReMuTe ships the trace to the ReMuTe hive. Thus, in ReMuTe, no trace that uniquely identifies a user is ever added to one of hive's repositories.

The job of the *Trace Anonymizing Pod* is to perform trace anonymization and compute the amount of personally-identifiable information left over.

Intuitively, the more program execution information a trace contains, the more useful it is for developers, as developers need not manually compensate for parts missing from the trace, but the weaker the source user's anonymity, as this information can be used to identify them. We describe means to quantify user anonymity and trace utility, and devise algorithms to improve anonymity without harming utility.

First, we observe that it is impractical to provide perfect anonymity to users, yet enable developers to know even the tiniest detail about a program's behavior. Conceptually, we can think of a program's execution as a jigsaw puzzle, of end users as people who print their picture on the puzzle, and then they give it to developers. Initially, without laying any piece, the portrait could be of anyone. But, as more pieces are laid, the space of possible portraits decreases. It is the same with program execution information, where each piece of information is a piece of the puzzle.

We aim to provide users with k-anonymity, i.e., a guarantee that an interaction trace can at most identify the user as being part of a group of k indistinguishable users. Continuing the puzzle analogy, developers should receive enough pieces to deduce that the portrait is of a member of a family, but be unable to identify the member.

Second, we define a trace to have test-generation utility if it describes an execution generated by an actual user. Third, our system improves user anonymity by expunging personally identifiable information and ensuring the user behavior encoded in a trace is not unique to that user.

In the rest of this chapter, we define k-anonymity (Section 4.2), then define a new metric that quantifies the amount of personally identifiable information contained in a trace (Section 4.3), and describe a technique that combines dynamic program analysis and crowdsourcing to provide users with k-anonymity (Sections 4.4 and 4.5).

4.2 Anonymity of Interaction Traces

We say a trace contains personally identifiable information (PII) if it can be used to determine a user's identity, either alone or when combined with other information that is linkable to a specific user [88].

Recall that an interaction trace is a sequence of events that determine a program's execution. The interaction trace contains two types of events:

- *proactive events* cause a feature of the program to execute (e.g., click on the "Send SMS" button), and
- *reactive events* provide the feature with the input it needs (e.g., the phone number and SMS text).

Both types of events may contain information that identifies users.

4.2.1 k-Anonymity

A data set satisfies k-anonymity if and only if each set of values that can be used to identify the source of a data element appears at least k times in the set [65], i.e., the source of an element cannot be narrowed down to fewer than k candidates. We say that each element of a data set satisfying k-anonymity is k-anonymous.

In [65], the data set is represented as a table PT, and each row contains information about a single subject. Some table columns contain private information (e.g., received medication), others provide identification details about the subject (e.g., birth date and zip code), but none contain information that explicitly identifies the subject (e.g., the name of the patient). Thus, one may naïvely conclude that the information in the table PT is anonymous.

k-anonymity quantifies the possibility of linking entries from the PT table with external information to infer the identities of the sources of the data in the PT table.

For example, consider there exists a set QI of columns in PT, called a quasi-identifier, (e.g., $QI = \{birth\ date, zipcode, gender\}$, $PT = QI \bigcup \{medication\}$) that also appears in a publicly available table PAT. If the PAT table contains additional columns that explicitly identify its sources (e.g., $PAT = Voters\ list = \{name\} \bigcup QI$), then an attacker can use the quasi-identifier values to join the two tables and learn private information about a subject (e.g., the

medication a person receives). The attack is similar to executing an SQL join operation on the PT and PAT tables that uses the quasi-identifier as the join condition.

The identification attack relies on the value of the quasi-identifier being unique for each subject in the PT and PAT tables. To achieve k-anonymity, one must modify the PT table to break this assumption [89]. This is not necessary if, in the PT table, each quasi-identifier value already appears k times. If not, one can suppress the entries that prevent achieving k-anonymity, or repeatedly use generalization strategies to make the values of the quasi-identifier less precise (e.g., replace the birth date with the year of birth) until k-anonymity is reached, or add new entries to the table to make it satisfy k-anonymity (not covered in [89]).

4.2.2 k-Anonymity Specific for Interaction Traces

We seek to prevent ill-intentioned developers and users from the ReMuTe community from abusing interaction traces to learn the identities of users who share traces.

A trace identifies its source through reactive events, which may contain explicit PII (e.g., usernames), or through proactive events, which detail user behavior. We aim to provide users with k-anonymity, which in our case represents the guarantee that a trace identifies its source as the member of a set of k indistinguishable users.

We say an interaction trace is k-anonymous if it is k-proactive-anonymous and k-reactive-anonymous. A trace is k-reactive-anonymous if, for each reactive event in the trace, there exist at least k alternatives (Section 4.4). A trace is k-proactive-anonymous if at least k users observed it (Section 4.5). Thus, a k-anonymous trace contains behavior exhibited by k users, and there are k alternatives for each program input contained in a reactive event.

We now describe how our technique differs from the original k-anonymity one [65]:

- The entries of the table equivalent to table PAT are interaction traces and each column corresponds to an event.
- ReMuTe computes the maximal k it can achieve for a trace, it does not enforce a particular k.
- ReMuTe cannot detect a minimum, complete quasi-identifier, as is assumed in [65], because the structure of a trace is unconstrained and its length is unbounded. ReMuTe takes a conservative approach, chooses completeness over minimality, and defines the quasi-identifier to span all the events in a trace.
- The equivalent of the PT table is distributed among users. While the ReMuTe hive could store all the observed, non-anonymized traces, doing so poses the risk of an attacker subverting the hive, gaining access to the raw traces, and thus being able to identify users.
- The pods share a trace with the hive only once it has achieved k-anonymity. k-anonymity increases, for example, when a new interaction trace causes existing ones to become k-proactive-anonymous.

4.3 Amount of Disclosed Information

We define the k-disclosure metric to quantify the amount of PII in a trace T. There are two observations underlying its definition. First, its value should depend on the k-anonymity of a trace T. If $k = 1$, then T is unique, so it represents a distinctive attribute of the user who generated it. On the other hand, the higher the k, the less specific to a user T is, i.e., k-disclosure $\sim \frac{1}{k}$.

Second, we consider the amount of PII to be emergent, because concatenating traces reduces the probability of the result be encountered by other users. Thus, if $T = T_1 + \cdots + T_n$, then k-disclosure$(T) > \sum_{i=1}^{n} k$-disclosure(T_i).

We define the value of the k-disclosure metric for an observed trace T, k-disclosure(T), to be the sum of the inverses of the values quantifying how k-anonymous is each of T's subsequences, $k(trace)$. That is, k-disclosure$(T) = \sum_{1 \leq i \leq j \leq |T|} \frac{1}{k(T_{ij} = <e_i, \ldots, e_j>)}$.

By normalizing k-disclosure(T) to T's length, we obtain the average amount of disclosed personally identifiable information per trace event.

We expect k-disclosure(T) to decrease over time because, once a program observes a trace, it is permanently added to local storage; as more users encounter T or its subsequences, the trace's k-anonymity increases.

4.4 Identifying Users Based on Program Input

In this section we describe how ReMuTe computes how k-reactive-anonymous is a trace. Section 4.4.1 describes the anonymity threats we consider, Section 4.4.2 presents the algorithm, and Section 4.4.3 details the challenges associated to the algorithm.

4.4.1 Anonymity Threats

Reactive events contain program inputs that can directly identify users, such as usernames. A reactive event is useful for replaying a trace T if it causes the program to make the same decisions during replay as it did during recording [90]. If one can replace a reactive event e_R^{orig} with $k-1$ reactive events e_R^{sub} without affecting the trace's ability to replay the program execution, then we say the trace T is k-reactive-anonymous with respect to event e_R^{orig}. More simply, we say e_R^{orig} is k-anonymous.

Consider the example of an Android application for sending SMS messages. The user fills in the destination phone number (reactive event e_R) and the message body. When the user presses the "Send" button, the application converts the digits representing the phone number to a `long`. Say that the user entered a phone number that starts with a '+' character, which is not a digit, and causes the program to crash. Any phone number string that does not start with a digit reproduces this crash, so ReMuTe can replace e_R with k alternatives, where k is the number of such strings.

4.4.2 Computing k-anonymity for Reactive Events

To compute the number k of alternatives for a reactive event e_R, ReMuTe must know how the program makes decisions based on the program input associated with e_R. To gain this knowledge, ReMuTe uses concolic execution [85]. Concolic execution is a dynamic program analysis technique. Programs run in a concolic execution engine that extracts program execution information that defines equivalent program inputs.

In concolic execution, a program input variable has two states: a "symbolic" state that encodes constraints on the variable, and a concrete state that contains the original program input value. Initially, a symbolic variable is unconstrained: it can have any value permitted by its type. For example, initially, a 32-bit `unsigned integer` symbolic variable x can take any value in the range of $[0, 2^{32} - 1]$.

During program execution, the concolic execution engine uses concrete variables to determine which branch target a program takes. In turn, the condition associated to the taken branch target places constraints on the symbolic variables. When the program finishes, the constraints describe all the values that fed to the program will cause it replay the same execution.

Consider the example code in Figure 4.1. Say that we run the program in a concolic execution engine with the initial input $x = 6$. When executing the `if` statement in line 2, since $x < 10$, the program takes the `then` branch,

and the concolic execution engine records the constraint $x < 10$. Similarly, at line 3, because x is greater than 5, the program takes the `then` branch, and the concolic execution engine records the constraint $x > 5$. At the end of the execution, we get the set of constraints $x < 10 \wedge x > 5$. One can use a constraint solver to get the values that satisfy both constraints. Any of these values will lead the program along the same execution path. That is, to replay the program execution described by input $x = 6$, we need not necessarily use 6 as input. ReMuTe applies this technique to the program inputs contained in reactive events.

```
1  int example(int x) {
2    if (x < 10) {
3      if(x > 5) {
4        return 0;
5      }
6    }
7    return 1;
8  }
```

Figure 4.1: Example program.

ReMuTe uses Algorithm 6 to compute how k-reactive-anonymous is an interaction trace. It replays each event $event$ in a trace T (lines 1-3). When replaying a reactive event (line 4), the algorithm copies $event.input$ (the program input contained in $event$) to $event.concrete$, marks $event.input$ symbolic, and adds an entry for $event$ in the map tracking path constraints (*PC*) (lines 5-7). When during the replay the program branches on a condition depending on the symbolic variable $e.input$, and both branch targets may be followed (lines 8-13), the algorithm forces the program to take the target that $e.concrete$ satisfies (line 14). Next, the algorithm uses static analysis to decide whether to add the path constraint corresponding to the taken branch to the *PC* map and maintain $e.input$ symbolic (lines 15-17) or to replace it with $e.concrete$ (line 19). When replay finishes, the algorithm computes the number of solutions for each reactive event e_R (lines 27, 28).

ReMuTe iteratively computes the number of solutions for each set of path constraints PC corresponding to a reactive event. ReMuTe iteratively generates a solution, then adds its negation to PC, and asks the constraint solver for another solution. This process is time consuming, so ReMuTe bounds the number of solutions, establishing a lower bound for how k-reactive-anonymous is a trace w.r.t. that event.

ReMuTe modifies the trace to replace each program input contained in a reactive event with one of the alternatives, thus removing the PII from the trace.

The algorithm is similar to the one described in [66]. The difference is our use of static analysis (line 15) to make concolic execution more efficient by avoiding the concolic execution of runtime-system code. This code does not affect the execution of the program, so it needlessly slows down concolic execution. The static analysis examines the stack trace of a program when it branches on a symbolic variable, and checks if the branch is in the program's code or if its result is used by the program—only then is the associated path constraint added to the PC map.

4.4.3 Challenges

While all alternatives lead the program along the same path, they may lead the underlying runtime system to different states, compared to the recording time. However, the differences do not affect the state of the program or its execution.

Next, if a program processes only part of the input contained in a reactive event, for which there are $k_1 = 1$ alternatives, but the number of alternatives for the entire input in that event is $k_2 > 1$, then the trace is 1-anonymous

Algorithm 6 k-anonymity for proactive events

Require: Interaction trace T

while $\neg T.isEmpty()$ **do**
  $event \leftarrow T.pop()$
  $event.replay()$
  if $isReactiveEvent(event)$ **then**
    $event.concrete \leftarrow event.input$
    $markSymbolic(event.input)$
    $PC[event] \leftarrow \emptyset$
    while $\neg event.finishedReplaying$ **do**
      $instr \leftarrow CPU.curr\_instr$
      $input \leftarrow instr.input$
      $e \leftarrow \{x | x \in PC \wedge x.input = input\}$
      $isSymbolicBranch = isBranch(instr) \wedge e \neq \emptyset \wedge bothBranchesFeasible(instr)$
      if $isSymbolicBranch$ **then**
        $CPU.next \leftarrow target(instr, e.concrete)$
        if $staticAnalysis(CPU.stacktrace)$ **then**
          $C = constraint(instr, e.concrete)$
          $PC[e] \leftarrow PC[e] \bigcup \{C\}$
        else
          $instr.input \leftarrow e.concrete$
        end if
      else
        $CPU.next \leftarrow CPU.step()$
      end if
    end while
  end if
end while
for each $e_R \in PC.keySet()$ **do**
  $k[e_R] \leftarrow |\{s | PC[e_R](s) = true\}|$
end for
return PC, k

(i.e., not anonymous), not k_2-anonymous, so all alternatives may identify the user. To prevent this, ReMuTe computes k based only on the processed parts of each reactive event. That is, if there exists a reactive event e_R that contains n program inputs, i.e., $e_R =< e_{R_1}, \ldots, e_{R_n} >$, and a set of path constraints, $PC(e_R)$, that refers to those inputs, i.e., $PC(e_R) = \{PC_1(e_1), \ldots, PC_m(e_n)\}$, $e_i \subset e_R$, then the number k of alternatives for the event e_R is the minimum of the number of solutions for each path constraint PC_i.

As an example, consider an application targeted at a company's employees that receives from its server, in a single response, a time stamp and a configuration parameter, `userIsCEO`, that specifies if the user is the CEO of the company. Assume the application configures a set of features based on the value of `userIsCEO` and then saves the time stamp to local storage. Thus, the time stamp does not influence the application's execution, while `userIsCEO` does.

Now, assume the CEO uses the application, `userIsCEO = true`, and wishes to contribute an interaction trace. ReMuTe can generate $2^{num\_bits(time\ stamp)}$ alternatives, because the application made no decisions based on the time stamp, so there are no constraints over it. Alas, ReMuTe must maintain the same value for the `userIsCEO` parameter and, thus, each alternative identifies the user as the CEO. However, if ReMuTe considers only the processed part of the reactive event that is the server response, then it cannot generate any alternative, because any other value for

`userIsCEO` leads the program along a different execution path. Thus, ReMuTe reports the trace is 1-anonymous and warns the user that the trace uniquely identifies them.

Alas, computing k based only on the processed parts of a reactive event creates the possibility of wrongfully considering that the interaction trace contains personally identifiable information (PII). This situation arises from ReMuTe's inability, as a platform, to make sense of the semantics of the program input contained in a reactive event. In the example above, if the server response contains a server health status field instead of the configuration parameter, and the program checks it against a specific value, then this field is still 1-anonymous, yet contains no PII.

There are two solutions to alleviate this drawback. First, if ReMuTe is unable to generate alternatives for a reactive event e_R, the *Trace Anonymizing* pod from one program instance can interrogate other pods from different program instances whether they observed e_R. In so doing, ReMuTe switches the definition of a reactive event being k-anonymous to be the number k of users who observed it. To compute k, ReMuTe uses the crowdsourced algorithm described in Section 4.5. In the previous examples, users will have seen a different value for `userIsCEO`, but the same server health status code. This solution trades part of a user's anonymity in exchange for verifying whether an entire trace is k-anonymous.

A second solution is to enable developers to annotate the source of reactive events to specify they do not contain PII and instruct ReMuTe to skip them when computing alternatives. We see this solution viable for open-source programs, where interested users can inspect a program's code and decide whether an event should be annotated or not. For the previous example, developers annotate the server health status field, but not the `userIsCEO` one.

4.5 Identifying Users Based on Behavior

In this section we describe how ReMuTe computes how k-proactive-anonymous is a trace. Section 4.5.1 describes the anonymity threats we consider, Section 4.5.2 presents the algorithm, and Section 4.5.3 details the challenges associated to the algorithm.

4.5.1 Anonymity Threats

Proactive events reveal user behavior, which can uniquely identify the user. For example, consider an application developed for a company's employees that proposes multiple sets of features, each corresponding to a specific department of the company, e.g., features for executive management, for the financial department, etc. Suppose an employee uses a feature accessible only to executive management, and then a feature accessible only to the financial department employees. By analyzing the proactive events in the trace, one can infer that the user is the company's CFO.

One cannot use generalization to achieve k-anonymity for proactive events, as was possible for reactive events. The rationale is that either there is a single way to use a feature of a program, e.g., there is a single "Log in" button, or the alternative proactive events lead the program to a state that is equivalent to the original one, making generalization useless. This situation arises from the fact that one needs to anonymize the user behavior, not its particular manifestation as one proactive event or another.

One can build different traces that lead a program to the same final state, similar to [91], but in so doing harm the trace's utility for test generation, since synthetic user behavior is substituted for the real-world one. Thus, we consider that proactive events *cannot* be removed from a trace or altered.

4.5.2 Computing k-anonymity for Proactive Events

To check if a trace T is k-proactive-anonymous, ReMuTe can query every other program instance ran by a user from the crowd of users whether it observed the interaction trace T in the past, and tally up the results. Alas, providing trace T to other program instances compromises users' anonymity.

The challenge is to design an algorithm that counts how many program instances observed the trace T without explicitly revealing T to them. Our solution is to hide T among a set S of traces, ask program instances whether they observed any of the traces in S, and probabilistically compute the number k of instances that observed T.

The algorithm runs as follows: program instance A, run by user U who wishes to share the trace T, constructs the query set S. The set S contains the hashes of trace T and of other traces that act as noise. Next, instance A sends the set S to the ReMuTe hive, which forwards the set S to each program instance A_i run by users U_i. Program instances A_i reply positively if they observed any trace from the query set S.

After ReMuTe records an interaction trace, it saves the hashes of the trace and of its sub-traces to a history set H. Thus, each program instance A_i has a history set H_i.

When receiving a query set S, each instance A_i prepares a reply $R_i(S)$ that is positive if its history set H_i contains any of the hashes in the set S, i.e., $R_i(S) = true \Leftrightarrow H_i \bigcap S \neq \emptyset$.

Then, A_i checks if $R_i(S)$'s value enables an attacker who has access to all previous queries and their results to infer that A_i recorded at least one trace from a set of fewer than k candidates (described later). In this case, instance A_i does not reply.

The ReMuTe hive counts the number K of positive replies and sends it to instance A, which computes the probability that k of the K instances recorded T—this determines how k-proactive-anonymous trace T is.

This algorithm protects the anonymity of users U and U_i, because instances A_i cannot learn T, and instance A cannot learn the trace that caused A_i to reply positively.

ReMuTe runs the same algorithm for each of trace T's sub-traces and computes the amount of personally identifiable information contained in trace T, i.e., k-disclosure(T). Finally, instance A reports the k and k-disclosure(T) values to the user U. If the user agrees, instance A shares the trace T with the ReMuTe hive.

4.5.3 Challenges

There are four challenges associated with this algorithm: i) instance A may be tricked into identifying its user by a sequence of crafted queries; ii) instance A needs to generate feasible traces as noise for the set S; iii) to compute the number k of instances that recorded T, instance A must approximate the likelihood of instances A_i recording each trace from the set S; and iv) instance A may be tricked into revealing T by an attacker compromising the result of the voting process.

4.5.3.1 Preventing Query Attacks

A program instance A may be tricked by an attacker M into revealing that it recorded a trace T that identifies the user. In the attack, M generates all traces that identify a user, e.g., using symbolic execution [92], and then issues a set of queries to the ReMuTe hive RH, each containing the hash of a single trace. Each positive reply from instance A unambiguously denotes that it recorded that trace and identifies the user.

A similar attack is one in which the attacker M subverts the ReMuTe hive, and after receiving a query set $S = \{T_1, \cdots, T_n\}$ from program instance A, attacker M starts issuing the queries S_i each of which contains a single hash,

i.e., $S_i = \{T_i | T_i \in S\}$. Each positive reply from instance A reveals a trace T_i that may not have been shared.

To prevent this query attack, each program instance A maintains a set of positively-answered queries and one for negative replies. A program instance does not reply to a query from RH if the set difference between the positive and the negative set has fewer than k elements. To bootstrap the process, instance A pretends it recorded all the traces from the first n received queries.

The algorithm it uses to decide whether to answer or not is as follows: Instance A keeps track of the sets S received from the ReMuTe hive RH and the values of the replies, $R(S)$. Specifically, A maintains two sets:

- The set V contains the elements of the sets S_v for which program instance A replied positively, i.e., $V = \{e_i | e_i \in S_v, R_i(S_v) = true\}$, and
- The set N contains the elements of the queries S_n that generated negative replies, i.e., $N = \{e_i | e_i \in S_n, R_i(S_n) = false\}$.

Upon receiving a set S_c from RH, instance A computes the reply $R_i(S_c)$ and adds S_c to either the set V or to the set N. Then, instance A computes the set $D = V \setminus N$ that determines the traces that appear in positively-replied queries, but have not generated a negative reply. Therefore, at least one of the traces in D has been recorded by instance A. If the cardinality of the set D is less than k, then instance A does not reply to RH.

For example, consider that the history set H of instance A contains only the trace T and that instance A has replied positively to $S_1 = \{T, X, Y\}$, and $S_2 = \{T, X, Z\}$, so $V = \{T, X, Y, Z\}$ and $N = \emptyset$. Now, suppose instance A receives the set $S_3 = \{X, Y, Z\}$, for which it should reply with $false$, so $N = \{X, Y, Z\}$. Now the set D contains only the trace T, $D = \{T\}$, so if instance A replies, then one knows that instance A recorded the trace T.

4.5.3.2 Generating the Noise Traces

The second challenge is generating feasible traces as noise for the set S. While ReMuTe can automatically generate noise traces by generating random interactions with the program, using such traces in the query set S has the drawback that a positive reply divulges the trace T to program instance A_i, because it is unlikely that A_i's history set H_i contains any of the other traces in S.

Instead, we envision four ways to generate hashes of feasible traces:

- Reuse hashes from queries received from RH,
- Use the *Test Generation* pod to generate new traces,
- Ask the user to generate noise traces, and
- Use already shared traces.

4.5.3.3 Computing k

The third challenge is computing the number k of program instances that recorded the trace T. Remember that instance A does not know which trace caused each of the K instances to reply positively.

Instance A estimates the probability of the trace being T based on the relative frequency with which the traces $T_i \in S$ appear in its history set and in received queries. This requires the sets H, V, and N to become multisets.

The assumption is that program users are similar and that all program instances receive all queries, so the probability distribution defined over instance A's history is valid across other instances. Thus, the probability of an instance

A_i replying positively because of a trace T_i is $P_{T_i} = \frac{count(T_i, H \bigcup V \bigcup N)}{\sum_{T_j \in S} count(T_j, H \bigcup V \bigcup N)}$, where $count(x, MS)$ denotes the number of times element x appears in the multiset MS. Traces generated by the *Test Generation* pod inherit the frequency from their parent.

There are two ways to compute how many of the K positive replies are due to the trace T. First, instance A can use combinatorics to provide different values for k and a probability of the value being correct.

For this, it uses the formula $P(k|K) = C_K^k \times (P_T)^k \times (1 - P_T)^{K-k}$. The rationale for the formula is as follows. Each instance A_i that replied positively because it recorded T did so with probability P_T. The probability that instance A_i replied because of another trace is $1 - P_T$. The probability that k specific instances all replied positively because of T, while the other $K - k$ did not, is $(P_T)^k \times (1 - P_T)^{K-k}$. Since in a population of size K there are C_K^k ways to select k members, the probability of any k instances having recorded T is $P(k|K) = C_K^k \times (P_T)^k \times (1 - P_T)^{K-k}$.

Next, instance A finds the maximum value of k for which $P(k|K) \geq threshold$, e.g., 50%, and reports k to the user, notifying the user that ReMuTe is confident that the trace T has been observed by at least k other users.

An alternative to compute k is to model the positive reply of a program instance A_i as a random variable X and assign to it 1 if the reply was caused by instance A_i having recorded trace T and 0 otherwise, i.e., X is 1 with probability P_T. Then, the K positive replies act as trials of the random variable X, yielding a value for $k = K \times P_T$.

4.5.3.4 A Cryptographic Escrow Scheme

The last challenge is to prevent an attacker from tricking instance A into sharing a trace T by artificially boosting the result of the crowdsourced voting process, e.g., by compromising the hive or by flooding the hive with positive replies.

As prevention measure, instance A sets up an escrow scheme that requires at least k users share T before developers can access it. Instance A encrypts the trace T with the trace's hash and places the encrypted trace in a repository. To be able to decrypt the trace T, one requires the hash of the trace T.

Next, instance A shares with the ReMuTe hive a part of the hash, called a share. We need a way to split the hash into shares so that it is impossible to reconstruct the hash with fewer than k shares. We use Shamir's secret sharing scheme [93] that generates shares using a random $(k-1)$-degree polynomial, i.e., $P(x) = \sum_{i=1}^{k-1} a_i \times x^{k-1} + SK$, where SK is the secret. A share of the key is a pair of values $(x, P(x)), x \neq 0$. One needs k shares to reconstruct $P(x)$'s coefficients and get SK.

Shamir's scheme uses a central authority to compute the shares. The ReMuTe hive could play this role, but as outlined in Section 4.2.2, this is a design we wish to avoid. Instead, we designed a version of the scheme in which each program instance computes independently the shares. The challenge is ensuring that instances that recorded the same trace T generate the same polynomial P. Otherwise, T cannot be recovered.

We propose to generate $P(x)$'s coefficients by successively applying a hash function to a trace T, i.e., $P(x) = \sum_{i=1}^{k-1} hash^{i+1}(T) \times x^i + hash(T)$, where $hash^i(T)$ denotes i applications of the $hash$ function. We conjecture it is computationally hard to use the equations $a_i = hash(a_{i-1})$ to find out $hash(T)$ and decrypt T.

The drawback of our scheme is that it requires k be fixed, to ensure all T's encryptions can be used to recover it. While there exist variants of Shamir's scheme that support changing the value of k, they require coordination among participants, to update their shares of the secret, mandating program instances maintain a list of peers who also shared T, thus breaking k-anonymity.

4.6 Chapter Summary

The success of Record-Mutate-Test and of the crowdsourced testing platform that embodies it hinges on its ability to protect the anonymity of users who contribute traces.

In this chapter, we presented a technique that provides users with k-anonymity, i.e., a guarantee that an interaction trace they share can at most identify them as being members of a group of k indistinguishable users.

The technique makes a better trade-off between user anonymity and developer productivity than existing automated bug reporting solutions, because it provides developers with means to reproduce a program execution.

The technique has two main components. First, it uses dynamic program analysis to expunge explicit personally identifiable information, e.g., usernames, and then ensures that the user behavior encoded in the trace is common to multiple users.

Having seen how ReMuTe protects users' anonymity, in the next chapter we describe how to assess the quality of the tests ReMuTe generates and when to stop testing.

Chapter 5

PathScore–Relevance: A Better Metric for Test Quality

5.1 Introduction

A software system's reliability is considered to be roughly proportional to the volume and quality of testing done on that system. Software vendors, therefore, use extensive test suites to reduce the risk of shipping buggy software. Record-Mutate-Test is an automated technique to generate tests that discover bugs in how software systems handle user errors, so that developers fix them and improve the reliability of their system. Therefore, the quality of the tests it generates influences the reliability of the system. In this chapter, we describe how developers measure the quality of testing using code coverage metrics and present a new code coverage metric. Code coverage metrics serve as means to quantify the effectiveness of the tests Record-Mutate-Test generates.

Testing is subject to constraints: it is resource-intensive and under serious time-to-market pressure, so there is rarely enough time to be thorough. This problem is compounded by the sheer size of today's systems. Such constraints call for difficult trade-offs: development managers aim to maximize software quality within the available human and time resources. Making these trade-offs requires weighing the benefits of more tests against the cost of writing them, and this is more of an art than a science.

To reduce reliance on art, software engineers invented quantitative ways of assessing the quality of a test suite. Metrics tend to lead to better-informed trade-offs. A common technique is to measure code coverage, i.e., how much of a system's code is exercised during testing. For example, line code coverage computes the percentage of all source code lines that were executed at least once during testing [64]. Thus, one can use the code coverage value obtained by a test suite to judge whether a test generation technique, like Record-Mutate-Test, is effective.

The higher the code coverage value, the better the test suite and the higher the expected quality of the tested product. It is therefore common for development organizations to choose a target coverage level between 75%-90% [64] and, once this is met, to declare the software ready to ship. The coverage target serves to prioritize testing of those components that fall short of the goal.

Yet, despite testing taking up more than half a typical development cycle [94], bugs still exist. Have the limits of testing been reached? We argue that there is still plenty of room for improving the *efficiency* of testing, i.e., we can achieve better testing in the same amount of time. In particular, widely-used code coverage metrics are poor

approximations of test quality, and they do not take into account the fact that not all code requires the same depth of testing. A poor coverage metric leads to suboptimal trade-offs and fails to steer the testing process toward efficiently improving software quality within the budget of available resources.

In this chapter, we introduce $PathScore\text{-}Relevance$ $\mathcal{PR}$, a new code coverage metric that induces developers to focus on the testing that improves the most the quality of the software they ship. In a nutshell, $PathScore\text{-}Relevance$ combines two measures:

- *relevance* $\mathcal{R}$ of a code component to the overall functioning of the system, as determined both explicitly by developers and implicitly by end users, and
- *path score* $\mathcal{P}$ that weighs execution paths inversely to their length, encouraging testers to exercise shorter paths first. Since longer paths consist of partially overlapping shorter paths, $\mathcal{P}$ discourages test redundancy.

While $\mathcal{R}$ defines the order in which to test components, $\mathcal{P}$ nudges the testing process toward lower-priority components whenever testing the higher priority ones has reached a plateau of diminishing returns.

In the rest of this section, we describe the limitations of current coverage metrics (Section 5.2), define $PathScore\text{-}Relevance$ (Section 5.3), and discuss its advantages and limitations (Section 5.4).

5.2 Limitations of Current Coverage Metrics

Ideally, developers use code coverage metrics as guides that point them toward the parts of the code that are in most need of additional tests. This need can be due to the corresponding code being highly critical and/or due to existing tests not exercising enough of the possible paths through that code. Time-to-market pressures require fast, smart choices in allocating quality assurance resources, so a metric that is closer to "the truth" than basic coverage is imperative. Thus, a good metric for assessing the quality of a test suite complements efforts aimed at improving software reliability, such as better testing tools and programming practices.

The two widely-known and used code coverage metrics are line coverage and branch coverage [95, 96]. A third, well-known and more powerful metric is path coverage. We reviewed them in Section 2.5.1. To briefly recap, line code coverage measures the percentage of source code lines in a program that were executed at least once during testing. Statement and basic block coverage are similar, the difference being the unit of measure, i.e., a programming language statement or a basic block, respectively. Branch coverage computes the fraction of branch outcomes taken during testing. Finally, path coverage reports the percentage of program execution paths explored during testing.

We believe that these coverage metrics have significant drawbacks that impede developers to use them as testing guides. Their main weakness is that they treat each line of code equally, assumption we find to be false. Code is different along three axis.

First, the behavior of a system is defined not only by its statements, but also by the sequence in which they execute. Alas, line and branch coverage metrics are oblivious to such order; for them, a program is a set of statements. Path coverage, on the other hand, does not suffer from this problem, but takes the other extreme and considers every possible sequence of statements. Yet, doing so renders the metric useless, because programs may have an infinite number of paths. Think about a web server: it runs in a never-ending loop, each time serving a possibly different web page. This argues for introducing a middle ground between no statement sequence information and complete sequencing.

Second, two identical programming errors that manifest in different parts of a system most likely have consequences of different severities on the system's behavior. For example, an off-by-one bug in the query engine of a

database management system (DBMS) has a substantially worse effect on quality than, say, an off-by-one bug in the DBMS's output formatting code. This argues for being able to differentiate the relative importance of testing depending on the target code.

Third, the three code coverage metrics described in the introduction of this section miss the fact that some code is more important than other. Consider, for example, error recovery paths: they typically represent small portions of the code, so testing them has minimal impact on the level of coverage. However, recovery code is critical: it runs seldom, but must run perfectly whenever it runs, because it is rescuing the system from the abyss of failure. Yet testers avoid writing recovery tests: they are more difficult than functionality tests, and, if one must improve code coverage by 5%, writing recovery tests is the least productive way to do so.

The last observation is relevant to this thesis, because user errors can lead to systems executing recovery code. Thus, it is important this code be thoroughly tested.

5.3 The PathScore–Relevance Coverage Metric

We propose to measure test quality with a combination of two sub-metrics: *component relevance* $\mathcal{R}$, which captures the importance of a component from both the developers' and from the end users' perspective, and *path score* $\mathcal{P}$, which prioritizes testing shorter program paths over longer ones. As we discuss later, $\mathcal{R}$ indicates which components deserve most attention, while $\mathcal{P}$ helps the testing process move on, when a point of diminishing returns is reached. We generically refer to an arbitrary unit of code as "component;" this could be a module, class, method, or block of code.

5.3.1 Component Relevance

As suggested in the earlier example concerning recovery paths, even a well-intentioned developer will, when under time pressure, aim to "check off" as many program issues as possible, regardless of how important they are; this is confirmed both by intuition and systematic studies [97]. By directly weighing the test quality result by the importance of the tested component, we can motivate solving problems in critical code instead of merely reaping low-hanging fruit in less critical code.

The goal of $\mathcal{R}$ is to rank components according to their importance, to an arbitrarily fine granularity. Since software testing is a perpetual compromise, greedily focusing on the most important components can improve the chances of being left with good software when testing resources run out.

There exist two primary stake-holders involved in determining the relevance of a component: developers and end users. In the absence of a widely accepted method for classifying components, we aim for a low-overhead way of taking the opinion of both stake-holders into account, while minimizing subjectivity as much as possible.

5.3.1.1 Component Relevance from Developers' Perspective

The first way to determine relevance of a component is through *annotations*: next to the name of each function, class, or subsystem, developers can include an annotation that informs the testing tool suite about the criticality of that code. Simple scales, such as high / medium / low, seem to work best in practice, and are routinely used in the case of log messages, to indicate their criticality. Annotation-based relevance information can be augmented with simple *static analysis* to identify, for example, exception handlers; depending on programming language, more general error recovery code can also be identified statically. Finally, developers may wish to complement annotations

and static analysis with an automatically-derived complexity measure of the respective components, like cyclomatic complexity [98] or Halstead's metric [99].

A second source for determining relevance is *code and bug history*. First, recently added or modified code ought to be emphasized in tests, because it is more likely to contain new bugs; studies have shown a direct correlation between the age of code and the number of bugs it contains [100]. Second, after the system has been delivered to the customer, bug reports filed by end users can be used as indicators of fragile components that require more attention. A recent survey found that there exists a strong correlation between bugs found by users and bugs detected using a static analysis tool [97], which suggests that the number of problems found through static analysis can also be used as a low-cost indicator for relevance, if collecting such statistics from bug databases is not possible. The bottom line is to test where bugs are most likely to be found.

Each of the methods above can provide a relevance measure for each component or block of code. If more than one measure is used, such as age of code together with whether it is or not recovery code, we must unify the measures into a single metric. The easiest approach is to normalize the individual metrics and ensure their semantics are uniform (i.e., value 1 should mean the same thing for all metrics).

In this chapter, we do not prescribe a specific way to combine the metrics, as each development team can decide on its own relevance metrics and aggregation method. Nevertheless, we do provide an illustrative example.

Say for each line of code l we represent its instability as a value $instability_l$ that ranges from 0, for lines that did not change since the last release, to 1, for lines that underwent repeated changes. We define the instability of a line of code l depending on the number of times line l was changed and by how many developers, i.e., $instability_l = 1 - \frac{1}{2^{C+D}}$, where C is the number of times line l was changed and D is the number of developers who changed that line. The $\frac{1}{2}$ factor can be thought of as the chance of a change to introduce a bug.

Second, we define a recovery flag *recovery* that indicates 1 for "yes" (code is meant to do recovery) or 0 for "no" (otherwise); this can be inferred based on annotations or static analysis.

For each component c in program P, we can compute a sum of the flags $\mathit{FlagSum}(c) = \sum_{l \in \text{code lines}}(instability_l + recovery_l)$ and express relevance as:

$$\mathcal{R}(c) = \frac{\mathit{FlagSum}(c)}{\max_{c_i \in P} \mathit{FlagSum}(c_i)}$$

$\mathcal{R}$ captures the fact that a component with lots of new or modified code or lots of recovery code should be tested more than components with less code of this kind. Code that is both new and performs recovery must be tested even more.

5.3.1.2 Component Relevance from End Users' Perspective

The second way to determine the relevance of a system's component is based on which parts of the software are most used by end users. Regardless of how the system is constructed, whatever code the user most depends on should be viewed as the most important and thus relevant to testing.

We rely on usage profiles to determine which components are most frequently used in the field. One might argue that a widely-used component is implicitly tested extensively by the users themselves, but that is true only of the particular exercised paths, not true of the component as a whole. For example, users may make frequent use of a network discovery component, but not exercise the recovery paths within that component. Usage profiles would indicate that component as being highly relevant, making it a high priority test target, including its recovery paths.

Record-Mutate-Test, the technique described in this thesis, lends itself to deriving the usage profiles. In Section 3.7, we describe how developers can leverage end users to collect interaction traces that can be transformed into usage profiles. These profiles indicate, for each component c, how many times it has been exercised.

If we think of a target program P as a collection of components, then we can use the information from the profile to define the relevance of each component c :

$$\mathcal{R}(c) = \frac{NumTimesExecuted(c)}{max_{c_i \in P}\ NumTimesExecuted(c_i)}$$

$\mathcal{R}$ captures the frequency with which c was executed by end users, relative to that component which was executed the most.

Regardless of whether we employ the developers' or the end users' perspective, relevance always varies from 0 (completely irrelevant) to 1.0 (most relevant). It is therefore trivial to combine the two perspectives either in a multiplicative or a weighted additive fashion, depending on what developers feel is most accurate for their needs.

5.3.2 Assigning Path Scores to Tests

In contrast to path coverage, which treats all paths equally, $\mathcal{P}$ aims to reward tests that exercise shorter paths over those that exercise longer ones. The rationale is that longer paths are composed of shorter paths that overlap, so testing done on the shorter paths can be leveraged in testing the longer paths.

We define the length of a path to be the number of basic blocks contained in that path. It is possible to use the number of statements as well, but the overhead of computing the metric in practice would be higher, in exchange for no increase in precision.

We denote by $NumPathsExercised_L(c)$ the number of paths of length L in a component c that were exercised by the test suite, and by $TotalNumPaths_L(c)$ the number of paths of length L in that component. We define a test suite's path score for the component c as the length-weighted amount of path exploration done by that test suite:

$$\mathcal{P}(c) = \sum_{L=1}^{max} \frac{1}{2^L} \times \frac{NumPathsExercised_L(c)}{TotalNumPaths_L(c)}$$

where max is the upper bound on the path length. This formula computes the fraction of fixed-size paths in c covered by the test suite. The value of $\mathcal{P}$ (c) ranges from 0 to 1.0, with 1.0 indicating that all paths of all lengths smaller than max were explored. The $1/2^L$ ratio serves to weigh longer paths exponentially less, according to their length; 2^L is the number of leaves one would expect in a typical execution tree of depth L.

In theory, the sum starts with paths of length 1 (as if measuring line coverage) and ends with paths of length max (as in bounded-length path coverage). However, the $1/2^L$ ratio limits the impact that long paths have on the score, so we expect that in practice L will be limited to a small number. E.g., $1/2^7 < 1\%$, so it is unlikely for paths of length $L > 7$ to improve $\mathcal{P}$ in any significant way. In that sense, path score $\mathcal{P}$ represents a practical compromise between line coverage and path coverage.

5.3.3 Aggregate PathScore–Relevance

The $PathScore\text{-}Relevance$ metric $\mathcal{PR}$ combines relevance $\mathcal{R}$ and path score $\mathcal{P}$ such that, the higher the relevance of a component c, the higher the path score must be, in order to achieve the same value of $\mathcal{PR}$ as less relevant components:

$$\mathcal{PR}(c) = min(1, \frac{\mathcal{P}(c)}{\mathcal{R}(c)})$$

Aiming to maximize overall $\mathcal{PR} = \sum_{c_i \in P} \mathcal{PR}(c_i)$ in a greedy fashion will guide the testing efforts to first improve $\mathcal{P}$ for components with high $\mathcal{R}$, as this is the quickest way to increase overall $PathScore\text{-}Relevance$ $\mathcal{PR}$.

$PathScore\text{-}Relevance$ prevents developers from writing tests for components with small $\mathcal{R}$ values by capping its maximum value.

5.4 Discussion

In this section, we present the advantages and disadvantages of $PathScore\text{-}Relevance$.

5.4.1 Advantages

The $PathScore\text{-}Relevance$ metric ensures that both the way customers use the system and the way developers understand the system help decide what level of test quality is sufficient. Our approach does not require end users to write specifications, rather they implicitly define what is important by using the system. $\mathcal{PR}$ prioritizes testing critical code (e.g., error recovery) or potentially unstable code (e.g., a recent patch), because such code will have a high $\mathcal{R}$ value from the developers' perspective.

At the same time, $\mathcal{PR}$ combines the tractability of computing line or basic block coverage with the accuracy of path coverage. $\mathcal{PR}$ treats components as sets of sequences of statements and is thus deeper semantically than basic block coverage, which only considers paths of length 1, but shallower than full path coverage.

5.4.2 Disadvantages

Like anything resulting from an engineering trade-off, $\mathcal{PR}$ also has shortcomings. First, $\mathcal{P}$ emphasizes short paths, but some bugs may lie on long paths within a component. Whether it is better to spend resources searching for that bug or finding other more shallow ones in other components depends entirely on how relevant that component is. If it is critical, $\mathcal{R}$ will keep the testing within that component, encouraging deeper testing, otherwise $\mathcal{P}$ will nudge testing toward less critical components, where the return on time invested may be higher. If, however, the overall value of $\mathcal{PR}$ plateaus before QA resources run out—indicating that a point of diminishing returns was reached while resources were still available—one can replace the aggressive exponential decay $1/2^L$ with a more slowly (but still monotonically) decreasing function f, as long as $\sum_l f(l) = 1$. This way, the relative influence of path length decreases in favor of $\mathcal{R}$.

Second, $\mathcal{P}$ treats a branch condition as a single true-or-false predicate. However, most predicates have multiple boolean sub-expressions, each of which may influence the subsequent path structure. The metric could be enhanced to support modified condition/decision coverage [64], which requires a test suite to set every sub-expression that can affect the result of the decision to both true and false.

Third, even a $\mathcal{P}$ value of 100%, meaning that all paths have been exercised, does not mean the component has been sufficiently tested. Consider, for instance, the following definition of the mathematical absolute function:

```
#define abs(x)  (x<0) ? (-x) : (x)
```

There are two paths, one for $x < 0$ and another for $x \geq 0$, so test cases `abs(-5)` and `abs(5)` will achieve complete coverage. However, `abs(INT32_MIN)` will wrongly return a negative number on most platforms.

Finally, unlike a simple line coverage metric, the connection between writing a test and the expected improvement in $\mathcal{PR}$ is not immediately obvious to a developer. E.g., writing a test that executes 10 new statements clearly increases the number of covered statements by 10, but its effect on $\mathcal{PR}$ is less clear. We expect, however, that developers' intuition can develop quickly, since $\mathcal{PR}$ is a relatively simple metric.

5.5 Chapter Summary

This chapter described $PathScore\text{-}Relevance$, a code coverage metric that guides the testing effort toward the components of a system that are most relevant to users and developers, which increases the chances of shipping software of higher perceived quality.

The new metric combines a program-path scoring metric that prioritizes shorter program paths over longer ones, thus creating a middle ground between line code coverage and path coverage, with a metric that quantifies the relevance of each a system's components, as defined by end users and developers.

In the next chapter, we describe the tools we built based on Record-Mutate-Test.

Chapter 6

Tools Built Using Record-Mutate-Test

6.1 Introduction

The first step to empirically demonstrate that Record-Mutate-Test is useful to developers is to build tools that embody it and apply them to real systems. Our tools target widely-used applications, such as web and smartphone applications, for which it is important to be resilient to user errors, because they target a large mass of users. Our tools also target web servers, because they power an important fraction of the applications we use today.

From a research point of view, the tools enable us to verify if the principles behind Record-Mutate-Test can be generalized to a wide variety of systems. They also enable us to try out various means to record and replay the interaction between users and systems. Finally, solving the particularities of each category of programs and systems we target enables us to enhance Record-Mutate-Test.

This section describes four tools we built based on Record-Mutate-Test. The first two tools target end-user errors, focusing on the interaction between users and modern web applications (Section 6.2) and on the interaction between users and smartphone applications (Section 6.3). The last two tools focus on system administrator configuration errors (Section 6.4 and Section 6.5).

6.2 WebErr: Testing Web Applications Against Realistic End User Errors

Web applications are becoming ubiquitous. Users rely increasingly more on applications that are accessed through a web interface rather than on shrink-wrapped software. One of the most widely used web applications is web-based email, such as GMail [101] that has over 400 million users [72]. With Google Drive [102], a productivity office suite, even applications that traditionally run completely on a user's computer are moved to the cloud and accessed through web browsers.

Modern web applications are distributed across back-end servers and front-end clients, with clients providing some of their functionality. Testing and debugging modern web applications requires a holistic approach that includes both client-side code and server-side code.

Testing a web application with realistic usage scenarios requires high-fidelity record and replay of the interactions between users and the application, because it is users, through commands, who trigger the features of an application to execute. High-fidelity recording means capturing *all* interactions, while high-fidelity replaying simulates *all*

interactions between a user and a web application.

High-fidelity record and replay is challenging because, nowadays, the HTML pages of a web application are mere containers whose contents dynamically change in reaction to user commands. Moreover, record and replay tools should always be recording, so that users can contribute complete interaction traces.

In this section, we present a tool, called WebErr, that is an implementation of the Record-Mutate-Test technique for testing web applications. We focus on application logic errors, and disregard bugs triggered by browser differences or network errors.

6.2.1 Design

WebErr relies on a high-fidelity record and replay system that is "always-on." Record and replay tools that target web applications, such as Selenium IDE [103], have low fidelity. Selenium IDE yields incomplete user interaction traces (deterring its record fidelity), fails to trigger event handlers associated to a user action (deterring its replay fidelity), and must be explicitly installed by users.

WebErr is an implementation of the Record-Mutate-Test technique applied to web applications. Figure 6.1 depicts how WebErr works: it records the interaction between a user and a web application as an interaction trace ①, then injects realistic user errors into this trace (② and ③), and then uses the WebErr Replayer to test that web application against the modified interaction traces (④).

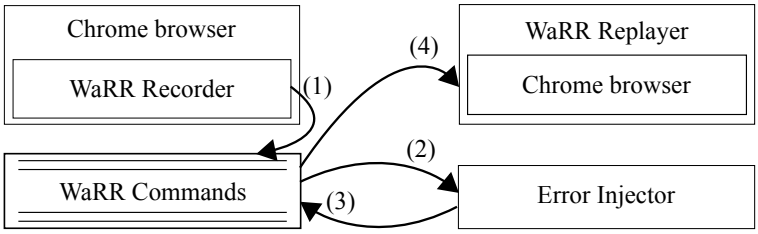


Figure 6.1: Testing web applications against user errors with WebErr.

WebErr has a novel architecture in which the record and replay functionality is an integral part of a web browser. This design decision brings five advantages:

- The WebErr Recorder has access to a user's every click and keystroke, thus providing high-fidelity recording.
- The WebErr Recorder requires no modification to web applications, being easy to employ by developers.
- The WebErr Recorder has access to the actual HTML code that is rendered, after code has been dynamically loaded.
- The WebErr Recorder can easily be extended to record various sources of nondeterminism (e.g., timers)
- Since the WebErr Recorder is based on the web page rendering engine WebKit [104], which is used by a plethora of browsers, it can record user interactions across a large number of platforms.

While WebErr is targeted at testing web applications, users can employ it to submit bugs as interaction traces, and developers can replay them to reproduce the bugs.

6.2.2 Definition of Interaction Traces

An interaction trace is a sequence of WebErr commands that describe the interaction between a user and the client-side code of a web application. A WebErr command contains:

- The type of user interaction, which can be `click`, `doubleclick`, `drag`, `type`, and `open`. The `click` and `doubleclick` WebErr commands simulate users clicking on HTML elements. The `drag` WebErr command simulates users dragging an HTML element from screen coordinates (x, y) to coordinates (x', y'). The `type` command simulates keystrokes. Finally, the `open` commands specifies which URL the user typed in the browser's address field.
- An identifier of the HTML with which the user interacted
- Action parameters
- The time elapsed since the previous WebErr command

Figure 6.2 shows a sequence of WebErr Commands slightly edited for readability.

```
click //div/span[@id="start"] 82,44  1
type  //td/div[@id="content"] [H,72] 3
type  //td/div[@id="content"] [e,69] 4
type  //td/div[@id="content"] [l,76] 7
type  //td/div[@id="content"] [l,76] 9
type  //td/div[@id="content"] [o,79] 11
type  //td/div[@id="content"] [ ,32] 12
type  //td/div[@id="content"] [w,87] 15
type  //td/div[@id="content"] [o,79] 17
type  //td/div[@id="content"] [r,82] 19
type  //td/div[@id="content"] [l,76] 23
type  //td/div[@id="content"] [d,68] 29
type  //td/div[@id="content"] [!,49] 31
click //td/div[text()="Save"] 74,51  37
```

Figure 6.2: Fragment of the sequence of WebErr commands recorded while editing a Google Sites web page.

A WebErr command identifies the HTML element with which to interact by using XPath [105] expressions. XPath is a language for locating an element in an XML/HTML document, by specifying a set of properties of that element and/or by specifying how to reach it from one of its ancestors. For example, `//td/div[@id="content"]` denotes an element of type `div` that has the property `id` set to `content` and is a child of an element of type `td`. For a single HTML element, there can be multiple XPath expressions, and various HTML elements may correspond to the same XPath expression.

`Click` and `doubleclick` WebErr commands indicate the click's coordinates in the web browser window, as backup element identification information. The `drag` command indicates the difference in the dragged element's position. The `type` command provides a string representation of a typed key and its ASCII code.

When typing letters using the `Shift` key, the browser registers two keystrokes: one for the `Shift` key and one for the printable key, but WebErr discards logging the `Shift` key since it is unnecessary. Other control keys, such as `Control`, do not always lead to new characters being typed, so WebErr logs their ASCII codes.

6.2.2.1 Reconstructing End Users' Interaction Trees from Web Application Interaction Traces

As described in Section 3.6.3, Record-Mutate-Test follows the process of how humans solve tasks: an initial task is split into subtasks, and these subtasks are then performed one by one [106]. Subtasks are recursively split into other subtasks until they can be performed directly in a web application (e.g., click on a particular link, type a key). Doing so yields an interaction tree. Figure 6.3 depicts such a tree for the task of editing a Google Sites website. Google Sites [107] is a structured wiki- and web page-creation tool.

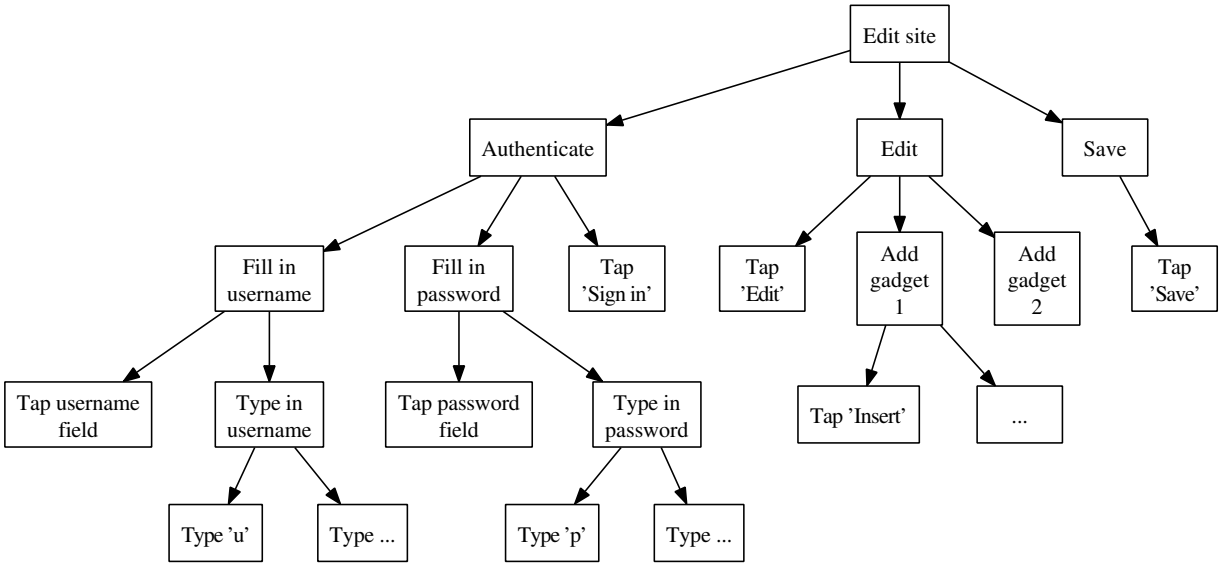


Figure 6.3: The interaction tree for editing a Google Sites website.

The basic idea of the interaction tree is to group together user actions, i.e., in this case WebErr commands, that belong to a task or subtask. The insight WebErr uses to automatically reconstruct the interaction tree is that good user experience requirements instruct developers to segment the tasks that users need to complete across different web pages. Thus, the WebErr commands that take place on the same page belong to the same (sub)task.

The challenge now is to determine when a web application user moved from one web page to another, so from one task to another. Naïvely, one can consider that as long as the browser address field shows the same URL, the user is on the same web page, so on the same (sub)task. Alas, this assumption is not true nowadays. The advent of JavaScript asynchronous requests enables developers to maintain a URL constant, but change the page content dynamically, which tricks the above strategy into merging multiple subtasks together.

Hence, WebErr needs to determine when web pages change by not relying only on the URL displayed in the browser's address field. We propose to use the structure of web pages to enhance the decision of whether the user moved from one task to the other.

We define the *dissimilarity* metric to quantify how different the *structures* of two web pages are. A web page is a tree of HTML elements. The value of the metric is the position of the first HTML element whose children differ in the two trees. WebErr traverses trees left to right and top to bottom. Elements are compared based on their type, e.g., `div`, and `id` property.

To use the *dissimilarity* metric, WebErr needs to know the structure of an application's web page after replaying a WebErr command. Alas, WebErr commands do not contain the changes that replaying them produces on the state of

Algorithm 7 Transforming an interaction trace into an interaction tree.

Require: Interaction trace IT

1: $interaction\_tree := \emptyset$
2: $root := nil$
3: $stack := \emptyset$
4: $last\_url := nil$
5: $last\_html\_element := nil$
6: **for each** $cmd \in IT$ **do**
7: $current\_url := cmd.current\_url$
8: $cmd.replay()$
9: $cmd.web\_page := browser.current\_web\_page$
10: $current\_html\_element := cmd.html\_element$
11: **if** $stack.isEmpty()$ **then**
12: $stack.push(cmd)$
13: $root := cmd$
14: $last\_url := current\_url$
15: $last\_html\_element := current\_html\_element$
16: **continue**
17: **end if**
18: **if** $last\_html\_element \neq current\_html\_element \wedge last\_url \neq current\_url$ **then**
19: $stack.clear()$
20: $stack.push(root)$
21: $stack.push(cmd)$
22: $interaction\_tree[root].append(cmd)$
23: **continue**
24: **end if**
25: **if** $last\_html\_element \neq current\_html\_element$ **then**
26: $parents := \{p|p \in stack \wedge dissimilarity(cmd.web\_page, p.web\_page) = min_{e \in stack} dissimilarity(cmd.web\_page, e.web\_page)\}$
27: $parent := p \in parents|p.level = min_{e \in parents} e.level$
28: $stack := stack[root : parent]$ // pop the $stack$ until $parent$ is on top
29: $interaction\_tree[parent].append(cmd)$
30: $stack.push(cmd)$
31: **continue**
32: **end if**
33: $parents := \{p|p \in stack \wedge dissimilarity(cmd.web\_page, p.web\_page) = min_{e \in stack} dissimilarity(cmd.web\_page, e.web\_page)\}$
34: $parent := p \in parents|p.level = max_{e \in parents} e.level$
35: $stack := stack[root : parent]$ // pop the $stack$ until $parent$ is on top
36: $interaction\_tree[parent].append(cmd)$
37: **end for**
38: **return** $interaction\_tree$

the web application, including changes to the URL displayed in the browser's address field. Thus, WebErr must replay an interaction trace and track web page changes in order to reconstruct an interaction tree from it.

Algorithm 7 shows how WebErr reconstructs an interaction tree. The output of the algorithm is a tree of WebErr commands. The basic idea of the algorithm is that after WebErr replays a command, the command is added as a child of the WebErr command that generated the web page for which the value of the *dissimilarity* metric is lowest.

The algorithm represents an interaction tree as a map between a WebErr command and its list of children (line 1).

It keeps track of the $root$ of the tree (line 2), uses a $stack$ to keep track of the path to the root of the tree (line 3), of the URL address shown in the browser's address field ($last\_url$) (line 4), and of the last HTML element with which the user interacted ($last\_html\_element$) (line 5).

Next, the algorithm replays each WebErr command (lines 6-37). When replaying the first WebErr command, the algorithm sets it to be the $root$ of the tree, pushes it on the $stack$, and updates $last\_url$ and $last\_html\_element$ to $cmd.current\_url$ and $cmd.html\_element$, respectively (lines 11-16).

Whenever WebErr replays a WebErr command, it performs two checks. First, if the command acts on an HTML element different from the one the previous command interacted with *and* changes the URL displayed by the browser (line 18), then the algorithm concludes that the user moved to a new task and makes the current WebErr command a child of the $root$ command (lines 19-23). An example of this is when the previous command was pressing the "Sign in" button while authenticating to Google and the current command is pressing the "Compose" button in GMail to start writing a new email.

The algorithm checks for both address field and target changes to prevent the situation in which the address field changes, which signals the beginning of a new (sub)task, but the HTML element remains the same, which signals that the (sub)task is the same. This is the case, for example, when composing an email in GMail. Every change in the fields of an email causes the browser to display a different URL. However, it is wrong to consider that each WebErr command that triggers such a change is a new task and should become a child of the root WebErr command.

The second check is for when the user switched from interacting with one HTML element to another, but the web page's URL did not change (line 25). This condition identifies the situation when the task the user is currently solving can be split into multiple subtasks, and the user moved from one subtask to another. In this case, WebErr adds a branch to the tree to represent the new subtask. The algorithm computes the value of the $dissimilarity$ metric between all web pages generated by the WebErr commands on the stack and the current web page. It then chooses the WebErr commands with the lowest $dissimilarity$ metric value (line 26) as possible $parents$ for the current command. If there are multiple possible parents, the algorithm chooses the one highest up in the tree as the $parent$, pops the $stack$ until $parent$ is on top, and pushes the current command on the stack (lines 27-31). The rationale behind choosing the parent WebErr command is to add a branch to the tree at the highest point in the tree where this new subtask can be performed.

An example of this is the log in page for GMail, where the user first typed in their username, and now they are typing in their password. Filling in the username does not change the structure of the web page. So, if WebErr does not choose the highest vertex in the tree, then the task of filling in the password becomes a child of the task of filling in the username, which prevents generating tests in which the order of filling in the two fields is swapped.

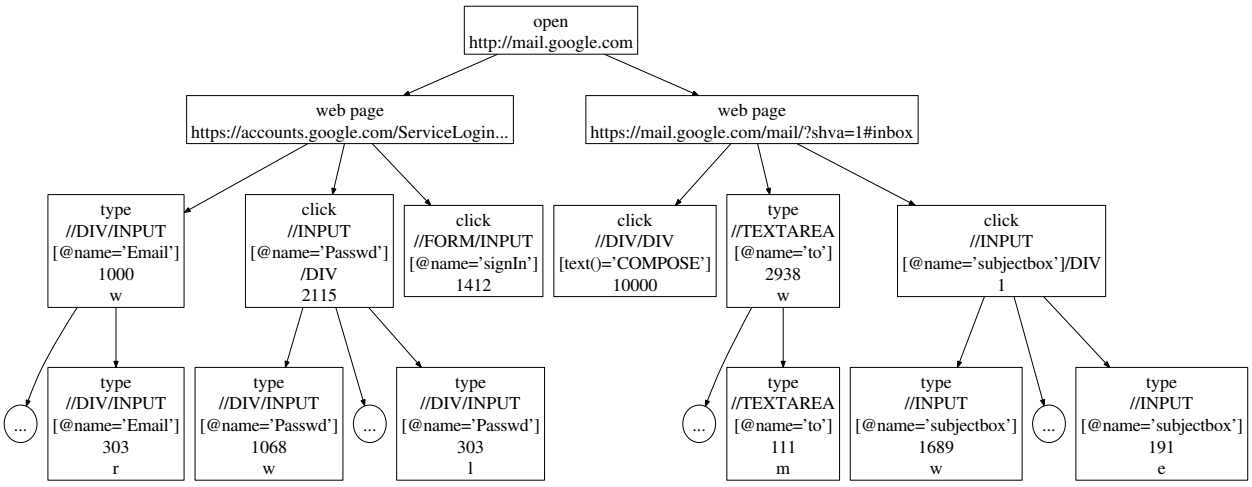
Finally, when the user continues interacting with the same GUI widget, the algorithm sets the command's parent to be the WebErr command that generated the most similar web page to the current one and is lowest in the interaction tree. This situation corresponds to users completing a low-level task, for example, filling in their username. The algorithm pops the $stack$ until $parent$ is on top (lines 33-36).

After the algorithm finishes reconstructing the interaction trace, WebErr adds a layer of nodes beneath the root. These nodes group WebErr commands based on the server's domain name and path of the web page they take place on. Their goal is to prevent WebErr from generating infeasible tests in which a command is performed on a page where its target HTML element does not exist. For example, pressing the "Compose" button on the Google login page.

Figure 6.4 shows how the algorithm works. Figure 6.4a shows a part of the interaction trace, while Figure 6.4b shows the reconstructed interaction tree.

```
open http://mail.google.com
type //DIV/INPUT [@type='text' and @name='Email' and @id='Email'] 1000 'w'
...
type //DIV/INPUT[@name='Email'] 303 'r'
click //INPUT [@id='Passwd' and @name='Passwd'] /DIV 2115
type //DIV/INPUT[@type='password' and @name='Passwd' and @id='Passwd'] 1068 'w'
...
type //DIV/INPUT[@type='password' and @name='Passwd' and @id='Passwd'] 303 'l'
click //FORM/INPUT[@name='signIn' and @value='Sign in'] 1412
click //DIV/DIV[@role='button' and text()='COMPOSE'] 10000
type //TD/TEXTAREA [@name='to' and @id=':mz' and @dir='ltr'] 2938 'w'
...
type //TEXTAREA [@name='to'] 111 'm'
click //INPUT [@id=':lw' and @name='subjectbox'] /DIV 1
...
type //INPUT [@name='subjectbox'] 191 'e'
```

(a) The example interaction trace.



(b) The interaction tree reconstructed from the interaction trace in Figure 6.4a.

Figure 6.4: WebErr interaction tree reconstruction example.

6.2.2.2 Interaction Tree Operators Specific to Web Applications

We define a new *alternative* function, named *timing_error* and used by the *insert* and *replace* user error operators, that simulates timing errors. Users cause timing errors when they interact with web applications while the latter are not yet ready to handle user interaction.

The advent of Asynchronous JavaScript And XML (AJAX) [108], which enables asynchronous browser-server communication, made web applications more vulnerable to timing errors, because users can interact with an application while not all of its code has been downloaded.

These errors occur because, although applications display wait messages, users disregard them. Hence, we consider them to be user errors, because the actions violate the interaction protocol.

To simulate timing errors, WebErr modifies the delay between replaying consecutive WebErr commands. WebErr stress tests web applications by replaying commands with no wait time. We formally define $timing\_error(event) =$

$\{event' | event' = event|_{delay=0}\}$.

We define a second *alternative* function, called *neighbors*, that simulates users interacting with a nearby HTML element instead of the recorded one. The *neighbors* function returns all HTML elements that are within x pixels from the bounds of the original HTML element.

Figure 6.5 shows an example of using the *neighbors* function. The button in the middle, sporting the "Button" label, appears in the interaction trace. The buttons that have parts that are within the dashed rectangle are the "Button"'s neighbors, i.e., "Button B," "Button D," "Button E," and "Button G."

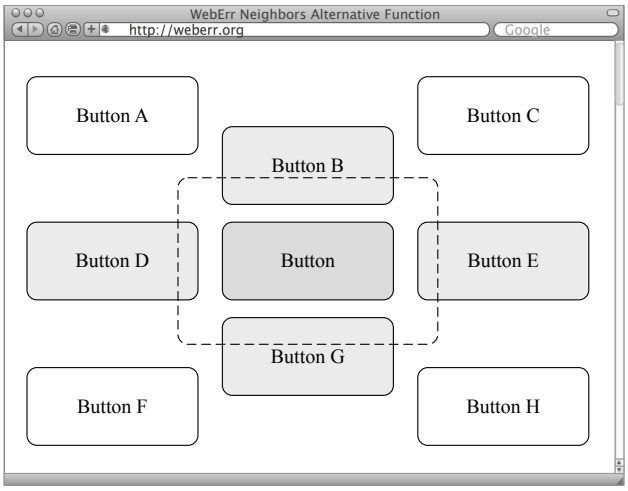


Figure 6.5: An example of using the *neighbors* function.

One challenge with defining this function is that an HTML page is composed of multiple layers and there always exists a neighbor for each HTML element. For example, every `input` HTML element is the child of the `body` HTML element, which is always its neighbor. To avoid such situations, WebErr defines the *alternative* function to take into account only HTML elements that respond to user interaction, e.g., they have the `onClick` Document Object Model (DOM) property set.

6.2.3 Recording Interaction Traces

The WebErr Recorder outputs a sequence of WebErr commands. The recorder offers high fidelity, is lightweight, always-on, and requires no setup, because the WebErr Recorder is embedded deep inside a web browser.

There are two main alternative designs. First, one could log the HTTP traffic between the web browser and the sever. But, increased HTTPS deployment makes logging HTTP traffic obsolete. Logging HTTPS traffic produces unintelligible traces, therefore the utility of such traces is much lower than in the case of HTTP. Moreover, there even exist proposals to replace parts of HTTP and augment it with SPDY [109]. Second, well-known web application testing tools like Selenium or Watir [110] rely on browser extensions to capture user interaction. Their drawback is that they are limited to the extension APIs provided by the browser. WebErr suffers from neither of these drawbacks.

The WebErr Recorder is a customized version of the WebKit HTML rendering engine and is embedded into the Chrome web browser. However, the Recorder is not specific to Chrome, rather it is specific to WebKit. The Recorder can be ported to other browsers, such as Apple's Safari.

Figure 6.6 shows Chrome's architecture for displaying web pages. As described in [111], *WebKit* is the rendering engine, *Renderer* proxies messages across process boundaries, *Tab* represents a web page, and *Browser window* contains all the opened Tabs.

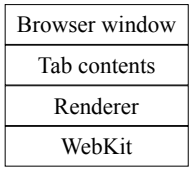


Figure 6.6: Simplified Chrome architecture.

Chrome's WebKit layer provides the ideal opportunity to record user actions: when a mouse button is clicked or a key is pressed, this event arrives at the WebKit layer to be dispatched to the appropriate HTML element. Figure 6.7 shows parts of Chrome's stack trace when handling such events.

```
WebCore::EventHandler::handleMousePressEvent
WebKit::WebViewImpl::handleInputEvent
RenderView::OnMessageReceived
IPC::ChannelProxy::Context::OnDispatchMessage
DispatchToMethod
MessageLoop::Run
ChromeMain
main
```

Figure 6.7: Fragment of Chrome's stack trace when performing a mouse click.

We implement the WebErr Recorder by adding calls to the recorder's logging functions in three methods of the `WebCore::EventHandler` class: `handleMousePressEvent`, `handleDrag`, and `keyEvent`. The changes amount to less than 200 lines of C++ code.

A benefit of our choice of implementation layer is that, if necessary, other events of interest can easily be monitored, requiring only slight modifications to the WebErr Recorder. For example, we initially did not record drag events, but adding support for them took less than one person day.

Even though WebErr's design requires browser changes, it brings an important advantage: Being based on WebKit, the WebErr Recorder can capture user interactions on more platforms than any other web application record and replay tool, because WebKit is used for desktop browsers, e.g., Chrome and Safari, and mobile devices browsers, e.g., in iOS, Android, and WebOS. Thus, WebErr enables developers to test web applications with usage scenarios originating from varied usage contexts.

6.2.4 Replaying Interaction Traces

The WebErr Replayer is the counterpart to the WebErr Recorder and simulates a user interacting with a web application as specified by WebErr commands. Replaying a WebErr command involves identifying the target HTML element and then asking Chrome to perform an action on it.

The WebErr Replayer has two main components: a browser interaction driver, based on WebDriver [112] and ChromeDriver [113], and the Chrome browser. WebDriver is a browser interaction automation tool that controls various browsers through a common API. It provides functionality to click, drag, and enter text. ChromeDriver is the WebDriver implementation for Chrome.

High-fidelity replay is hard to achieve in browsers based on WebKit, because they make certain properties of JavaScript events read-only. This prevents event handlers associated to such events from running with correct parameters, thus damaging replay fidelity. Since the WebErr Replayer targets developers, its browser need not obey such restrictions. We modify Chrome to enable setting properties of the `KeyboardEvent` JavaScript event, making such events practically indistinguishable from those generated by users.

Chrome is controlled through a ChromeDriver plug-in composed of a master and multiple ChromeDriver clients, one for each `iframe` in an HTML document. The master acts as a proxy between the ChromeDriver clients and the rest of ChromeDriver/WebDriver. Clients receive commands and execute them on the `iframe` they are responsible for. At any point in time, only one client executes commands.

6.2.4.1 Replay Challenges

The main challenge faced by the WebErr Replayer is identifying HTML elements whose properties differ between record time and replay time, which causes the recorded XPath expression to become invalid. For example, whenever GMail loads, it generates new `id` properties for HTML elements.

To mitigate this problem, whenever the WebErr Replayer cannot find the target HTML element, it employs an automatic, application-independent, and progressive relaxation of that element's XPath expression, until it finds a matching target. This automatic relaxation is guided by heuristics that remove XPath attributes. These heuristics are:

- Keep only the `name` property. For example, transform `//DIV/INPUT[@type="text" and @spellcheck="false" and @name="Email" and @id="Email" and @value=""]` into `//DIV/INPUT[@name="Email"]`. This heuristic is useful when logging in to Google.
- Keep only the `id` property.
- Remove the `id` property, but keep the rest of properties.
- Remove the `text()` property, but keep the rest of properties.
- Remove the `href` property, but keep the rest of properties.
- Remove the `name` property, but keep the rest of properties.
- Keep the `name, id, title,` and `text()` properties.
- Remove a component of the XPath expression. For example, transform `//TD/TEXTAREA[@name="to"]` into `//TEXTAREA[@name="to"]`.
- Search for an HTML element across all the frames embedded in a web page.

WebErr applies the above heuristics recursively. For example, when sending an email using GMail, WebErr recorded a user's interaction with an HTML element identified by the `//TD/TEXTAREA[@name="to" and @id=":mz" and @spellcheck="false" and @tabindex="1" and @dir="ltr" and @aria-haspopup="true"]` XPath expression which corresponds to filling in the "To" field of an email. Alas, due to multiple GMail layout changes over the years, WebErr can no longer find the matching HTML element. By applying the heuristics outlined above multiple times, WebErr derives the `//TEXTAREA[@name="to"]` XPath expression that identifies the correct HTML element.

With each round of applying the heuristics, the precision of the element identification process, and thus of the replay, decreases, because by removing constraints from the XPath expression, the number of possible HTML elements that match it increases.

By using these heuristics, WebErr can tolerate extensive changes to a web application's DOM between record time and replay time. To replay a user action on an HTML element, the WebErr Recorder requires the application preserve only some of the DOM properties of that element.

The next major challenge we faced was ChromeDriver's incomplete functionality. First, ChromeDriver lacks support for double clicks. It is important for WebErr to be able to replay double clicks, because web applications that use them, such as Google Drive, are becoming increasingly popular. We add double clicking support by using JavaScript to create and trigger the necessary events.

Second, ChromeDriver does not handle text input properly. When simulating keystrokes into an HTML element, ChromeDriver sets that element's `value` property. This property exists for `input` and `textarea` HTML elements, but not for other elements (e.g., `div`, a container element). We fix this issue by setting the correct property (e.g., `textContent` for `div` elements) as the target of received keystrokes and triggering the required events.

The third replay challenge we encountered was improper support for `iframe`s, and it involved Chrome and ChromeDriver. An `iframe` allows an HTML document to embed another one. First, one cannot execute commands on `iframe`s that lack the `src` property, because Chrome does not load ChromeDriver clients for them. We solve this issue by having the ChromeDriver client of the parent HTML document execute commands on such `iframe`s. Second, ChromeDriver provides no means to switch back to an `iframe`. We use a custom `iframe` name to signal a change to the default `iframe` and implement the necessary logic.

The last challenge was ChromeDriver becoming unresponsive when a user changed web pages. Chrome unloads the ChromeDriver clients corresponding to the `iframe`s of the old page and loads new clients for the new page's `iframe`s. The ChromeDriver master keeps track of the active client, the one executing commands, and when it is unloaded, a new active client is selected. The selection process assumes an order of web page loads and unloads, but Chrome does not ensure this order, so a new active client may not be chosen. Therefore, new commands will not be executed, and the replay halts. We fix this issue by ensuring that unloads do not prevent selecting a new active client.

6.2.5 Limitations

Currently, WebErr cannot handle pop-ups, because user interaction events that happen on such widgets are not routed through to WebKit. A possible solution is to insert logging functionality in the browser code that handles pop-ups, but this would make the WebErr Recorder specific to Chrome. The WebErr Recorder is specific to WebKit, not the Chrome browser.

WebErr interaction traces capture a single user's perspective on interacting with a system, so it cannot test collaborative web applications.

WebErr cannot control the environment it runs in and, therefore, cannot ensure that event handlers triggered by user actions finish in the same amount of time during replay as they did during recording, possibly hurting replay accuracy.

WebErr targets web applications, which are widely used, e.g., GMail has over 400 million users [72]. Another type of applications, whose popularity is rapidly increasing are smartphone applications. They are the target of the next tool we built, called ReMuTeDroid, and which we describe next.

6.3 ReMuTeDroid: Testing Android Applications Against Realistic User Errors

Having built a testing tool for web applications, we built another tool, called ReMuTeDroid, that embeds the crowd-sourced testing platform ReMuTe and targets Android smartphone applications. Building this tool enabled us to try a different means to record and replay the interaction between a user and a program, i.e., from inside the program, and to develop new user error models.

We chose smartphone applications because the adoption rate of smartphones and tablets is soaring, making such devices ubiquitous. Google has reported that it activates more than one and a half million Android devices *daily* [114]. Hundreds of millions of users have downloaded tens of billions of smartphone applications [115]. Yet, smartphone applications are buggy.

Developing smartphone applications presents developers with new challenges, compared to developing desktop applications. These challenges beget new types of bugs. For instance, to conserve battery power, smartphone operating systems (OSes) enforce life-cycle state transitions onto applications, e.g., they move applications from the running state to stopped as soon as they are no longer visible. Mishandling such transitions proves to be a major cause of bugs [116].

An increasing number of tasks can be solved using smartphone applications, and, therefore, an increasing number of users rely on them. Yet, due to the lack of thorough application store admission control, users are left open to bugs that compromise their security, privacy, or prevent them from completing their tasks. On the other hand, users treat smartphones as appliances. When applications can affect one's sensitive information (e.g., financial applications such as PayPal), smartphones become dangerous devices when used carelessly.

In this section, we describe ReMuTeDroid, the embodiment of Record-Mutate-Test for smartphone applications.

6.3.1 An Overview of Android Applications

Android is the most popular mobile device operating system, as of 2013, holding a market share of over 80% [117]. Google, its developer, reports there are more than 1 billion Android devices currently in use [118]. Therefore, Android applications are an interesting target for Record-Mutate-Test due to Android's vast number of users, which translate into a vast source of tests and a large test execution platform.

Android applications are generally written using the Java programming language and run inside the Android Dalvik virtual machine. Android applications have four types of components:

- `Activity`. Activities represent single windows that contain the UI widgets with which users interact.
- `Broadcast Receiver` enable applications to react to device-wide notifications. When the state of the device changes, the Android OS may send a broadcast notification to all running applications, e.g., that the battery is low. Broadcast receivers enable an application to react to such notifications.
- `Content Provider`. A part of an application can provide input for another application. For example, a photo taking application may provide other applications with the list of photos it took. This sharing of data is managed through content providers.
- `Service`. A service is a background thread that can be used to perform long-running operations and does not provide a user interface. For example, playing music from one's library can be done using a service. Activities can connect to a service and interact with it, for example to change the song.

In this thesis we focus on testing smartphone applications against user errors. Since such errors manifest during users' interaction with an Android `Activity`, we focus on this Android component.

6.3.2 Design

ReMuTeDroid's design closely follows ReMuTe's design, which was presented in Chapter 3, Section 3.7. Recall that ReMuTe is composed of modules called pods that communicate with the ReMuTe hive, which resides in the cloud.

ReMuTeDroid is built as a development platform that enhances the Android SDK. Most of the time, developers build applications using features provided by the Android SDK, but in certain cases, e.g., when performing asynchronous tasks, they need to use the ReMuTeDroid SDK.

To use ReMuTeDroid, developers need to change their build system. This allows ReMuTeDroid to transparently inject the *Record* and *Replay* pods into a target application.

6.3.3 Definition of Interaction Traces

ReMuTeDroid records the interaction between users and a program and between that program and its environment, including network communication, in interaction traces. The interaction traces contain the following events:

- *Click* and *Long click* events correspond to the user tapping on a UI widget. In Android, the GUI is a tree of UI widgets. A widget is identified using the path from the root of the UI widget hierarchy to that widget, i.e., by its XPath expression.
- *Orientation* events denote users tilting their device. In Android, changing screen orientation leads to restarting the current Activity.
- *Type* events contain the text entered by a user in a GUI widget.
- *Back press* events denote that users pressed the `Back` button on their Android device, which destroys the current Activity.
- *Key press* events correspond to users pressing a physical button on their device.
- *Menu button pressed* events denote that a user invoked an application's menu.
- *Menu item selected* events record what menu item the user selected.
- *Motion* events store a user gesture.
- *Scroll* events denote users scrolling through a list.
- *Search* events correspond to users invoking the universal Android search facility.
- *Slider* events retain the value a user chose using a `Slider` UI widget.
- *Date* events record the date information an application reads.
- *Location* events save information about the geographical location of the user.

- *Request start* events correspond to the beginning of an asynchronous task, i.e., a piece of code that runs on a thread different from the main, UI thread. For example, one uses asynchronous tasks to perform network requests.
- *Request end* events denote an asynchronous task finished.

We can observe that ReMuTeDroid logs user interaction events at different levels, e.g., clicks and gesture events. This is because Android OS dispatches interaction events to applications hierarchically along two dimensions. First, when a user taps the screen of a device, Android OS tries to interpret the tap as a click and checks whether the application registered a click event callback. If it does, then Android invokes that callback function. If not, Android checks whether the application registered a callback for a motion event, i.e., a gesture. The second dispatch dimension is from child to parent. If Android cannot dispatch the event to a UI widget, it will try to dispatch it to that widget's parent.

In addition to these events, ReMuTeDroid records other events that help it to generate tests. Recall that a challenge for WebErr was determining what web page a user interaction took place on, and WebErr had to analyze web pages to determine if a page changed enough from the previous one to warrant considering it as a new one. In ReMuTeDroid, we bypass this challenge by *recording* the currently running Android Activity, which is similar to logging a browser's address field. ReMuTeDroid adds three new types of events to the interaction trace:

- *Activity started* events denote that a new Android `Activity` came to the foreground.
- *Dialog started* events record that a dialog window appeared on the device's screen.
- *Dialog closed* events record the fact that the dialog window disappeared.

ReMuTeDroid's events are extensible. Application developers can easily add events that encode a specific sequence of actions or that change replay settings. For example, we developed an event that simulates users pressing the home button and then returning to the application and another event that causes the replay infrastructure to relax replay enforcement, e.g., network requests are allowed to go through and reach their intended server.

Figure 6.8 shows an example of a simplified interaction trace recorded by ReMuTeDroid. Each event contains a boolean parameter that specifies whether a user generated that event or not.

It is possible that in response to a user event, the application triggers another UI event, which is also recorded. ReMuTeDroid logs this causal relationship between the two events by using a $nesting$ level.

6.3.3.1 Reconstructing End Users' Interaction Trees from Android Application Interaction Traces

ReMuTeDroid follows the same steps in generating tests as Record-Mutate-Test. Before the *Test Generation* pod can use the user error operators defined in Section 3.5, it must reconstruct the interaction tree. Recall that the interaction tree is a hierarchical description of the interaction between a user and an application, and it groups together events that belong to the same (sub)task.

Algorithm 8 shows how ReMuTeDroid reconstructs the interaction tree. The general idea behind the algorithm is to use `StartActivity`, `StartDialog`, and `CloseDialog` to determine when a user moves from one Activity or dialog window to another one. The algorithm groups together all events that take place on the same Activity or dialog window. That is, a (sub)task is defined by an Activity or dialog window.

Compared to WebErr's Algorithm 7, this algorithm is simpler. The most important distinction is that ReMuTeDroid does not need to replay an interaction trace to reconstruct its interaction tree. We believe that for web applications,

```
StartActivity false DashboardView
NewSession true
OnClick true LinearLayout[2]
StartActivity false FoodMainView
RequestStart false
RequestReturned false
Date false
Scroll false ListViewElement[0]
MenuButonPressed true
MenuItemSelected true Suggestions
StartActivity false FoodSuggestionsView
Scroll false ListViewElement[0]
OnClick true CheckBox[0]
OnItemClick false CheckBox[0]
OnClick true CheckBox[0]
OnItemClick false CheckBox[0]
OnClick true ButtonElement[0]
Scroll false ListViewElement[0]
OnClick true ImageButton[2]
Scroll false ListViewElement[0]
BackButton true
```

Figure 6.8: A simplified interaction trace recorded by ReMuTeDroid when interacting with PocketCampus [119].

Algorithm 8 Transforming an interaction trace into an interaction tree.

Require: Interaction trace IT

1: $interaction\_tree := \emptyset$
2: $root := IT[0]$
3: $parents := root$
4: **for each** $curr\_event \in IT \setminus \{root\}$ **do**
5: $\quad parent = parents.top()$
6: $\quad prev\_event = IT.before(curr\_event)$
7: $\quad$ **if** $prev\_event.nesting < curr\_event.nesting$ **then**
8: $\quad\quad parent = prev\_event$
9: $\quad$ **end if**
10: $\quad$ **if** $isStartActivity(curr\_event) \vee isStartDialog(curr\_event)$ **then**
11: $\quad\quad$ **if** $parents.contains(curr\_event)$ **then**
12: $\quad\quad\quad parents.pop()$
13: $\quad\quad$ **else**
14: $\quad\quad\quad parent = prev\_event$
15: $\quad\quad\quad parents.push(curr\_event)$
16: $\quad\quad\quad interaction\_tree[parent].add(curr\_event)$
17: $\quad\quad$ **end if**
18: $\quad\quad$ **continue**
19: $\quad$ **end if**
20: $\quad$ **if** $isCloseDialog(curr\_event)$ **then**
21: $\quad\quad parents.pop()$
22: $\quad$ **end if**
23: $\quad interaction\_tree[parent].add(curr\_event)$
24: **end for**
25: **return** $interaction\_tree$

developers wish to reduce the page loading time, and this is why they are reluctant to create a new web page for each user interaction (sub)task. Instead, web applications just load the part of a web page that changes, saving the cost of downloading, for example, the application's header and footer. On the other hand, smartphone users get the entire code of an application when they download it. Thus, there is no benefit in reusing the same Activity for multiple (sub)tasks. This observation enables ReMuTeDroid to use Activities to cluster interaction trace events.

Algorithm 8 represents an interaction tree as a map between an event and the list of its children (line 1). The tree is rooted at the first event from the interaction trace (line 2). The algorithm uses a stack data structure, called $parents$, to keep track of the stack of running Activities. The top of the stack contains the currently visible Activity or dialog window (line 3). The algorithm processes ReMuTeDroid events iteratively (lines 4-25). It begins by proposing the parent of the current event, $curr\_event$, to be the event at the top of the $parents$ stack (line 5). However, if the previous event triggered the current one, i.e., the previous event's $nesting$ level is smaller than the current event's one, then the previous event becomes the parent of the current event (line 6). This ensures that ReMuTeDroid does not generate a test in which the order of the two events is swapped, which is likely to be an unreplayable test.

If the current event denotes the start of an Activity (line 10), then there are two cases. First, it may be that the user is returning to a previous Activity, i.e., the user finished interacting with an Activity that was started from this Activity (line 11). In this case, the $parents$ stack is popped (line 12). Second, if this Activity does not appear on the $parents$ stack, then it means the user is starting a new step in their interaction with the application (line 13). In this case, the previous event becomes the parent of the current event (line 14), the current event is pushed on the $parents$ stack (line 15), and the current event is added to its parent's children (line 16). If the event represents a dialog window closing (line 20), then the $parents$ stack is popped (line 21). Finally, the current event is added to the children list of its parent (line 23).

Figure 6.9 shows how the algorithm reconstructs the interaction tree based on the interaction trace from Figure 6.8.

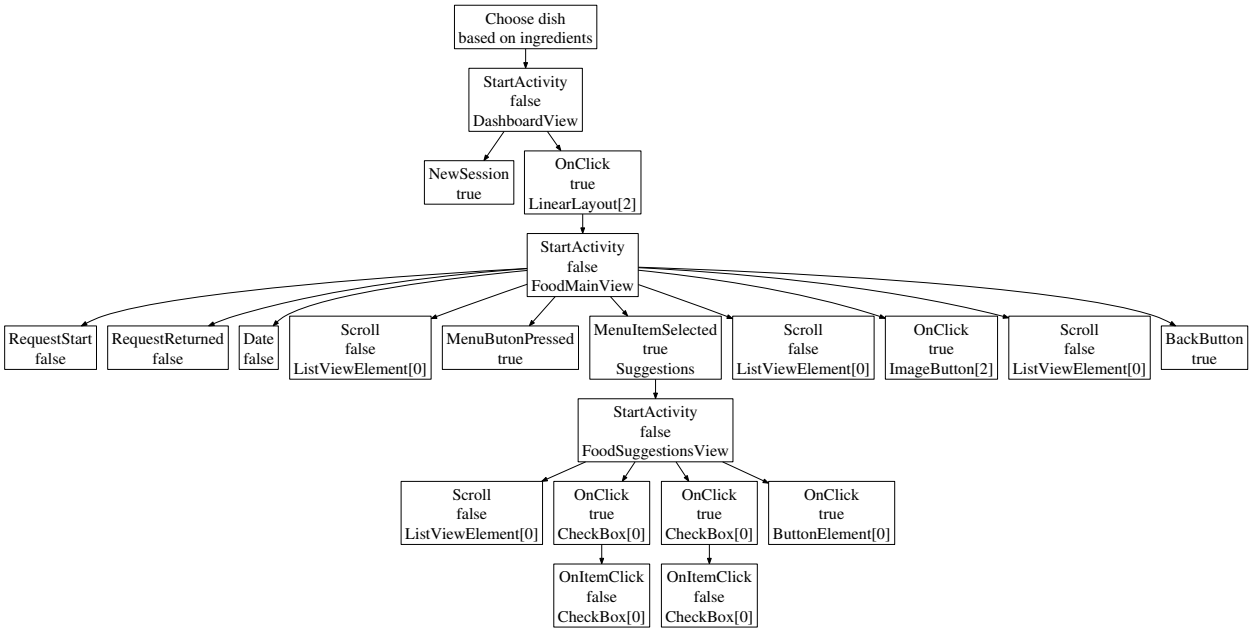


Figure 6.9: The interaction tree reconstructed by ReMuTeDroid.

6.3.3.2 Interaction Tree Operators Specific to Android Applications

Android is a new operating system specifically designed for mobile devices. This requires Android be stingy with its resources, e.g., main memory. Android applications have reduced memory at their disposal compared to desktop Java programs. Therefore, it is more likely that they will run out of memory and get killed by the OS if they do not manage their memory correctly.

One particular instance where this problem is prone to manifest is scrolling through lists. Consider an example in which each list item contains an image that is downloaded from the Internet when the item becomes visible. If the application does not keep track of each image it downloaded and re-downloads it every time an item becomes visible, it may be the case that if the user scrolls multiple times through the list, the memory consumed by the images grows beyond the maximum limit, and the application is killed.

We model this error using a new event that simulates users scrolling 20 times from the first element in the list to the last and back up again. In terms of user error operators, this error can be modeled as a new $alternative$ function that returns this new scrolling event and is used by the $insert$ user error operator.

The Android OS restarts the foreground Activity when the orientation of the device changes, e.g., from portrait to landscape. Since this behavior is specific to mobile, Android devices, programmers coming from other development platforms may not correctly handle such situations, even though they are common behavior to smartphone users. We model this error using a new event that triggers this orientation change. Again, we define a new $alternative$ function that returns this event and is used by the $insert$ user error operator.

Another common usage pattern encountered among smartphone users is to press the `home` button, start another application, and then come back to the current application. This again triggers a sequence of state changes for the application that developers may not correctly handle. Researchers show that this type of behavior can lead to program crashes [116]. We model this error using a new event that simulates the calls to the state-changing methods associated to users pressing the `home` button and use it to define a third $alternative$ function.

Users are prone to mishandle mobile devices. For example, an unlocked device put in one's pocket may register spurious taps and text input. Or one may knock over one's smartphone and while trying to catch it, one inadvertently generates a random sequence of user events. We model this error by developing a new event that simulates random user interactions, and use it to define a fourth $alternative$ function used by the $insert$ user error operator.

Compared to normal random testing, using this new event has the advantage that it injects random events after the user interacted with the application, not when the application just started. This increases the random events' bug-triggering potential. The obvious example for this advantage is applications that require users to sign in: it is unlikely for a random sequence of events to generate a suitable pair of username and password.

Mobile devices have small screens, so one frequent error is tapping a button different than the intended one. We simulate this error by logging the identities of UI widgets for which the application registered a call back and which are within half-a-thumb away from a UI widget with which the user interacted. We use these UI widgets to define an $alternative$ function that is used by both the $insert$ and $replace$ user error operators. A similar error manifests when a user selects an item from an application's menu. We define a similar $alternative$ function that returns other possible menu items.

The Android runtime requires applications to perform network communication asynchronously, on a thread different from the main, UI thread. The purpose of this requirement is to maintain the application responsive. Alas, this has the drawback that users can make the application use uninitialized state that depends on the outcome of the network request, which may cause the application to crash. As an example, consider an application that downloads a list of

movies from the Internet and allows users to sort them by rating. A negligently built application may allow users to press the sorting button before any movies are downloaded, and the application may use uninitialized state and crash.

We model this error as a new user error operator that moves user-generated events before the events signaling completion of network requests. One can view this new operator as an extension to the *swap* operator.

An example of this error is, in Figure 6.9, to move the `MenuButtonPressed` and `MenuItemSelected` events before the first `RequestReturned` event.

6.3.4 Recording Interaction Traces

This section describes ReMuTeDroid's *Record* pod. We decided to embed the *Record* and *Replay* pods inside programs, so *every* Android user can install programs based on ReMuTeDroid. Thus, each such program ships with its own record and replay infrastructure, so all users can participate in the crowdsourced testing of the program.

A common paradigm to handle user interaction is for programs to provide callback functions that the runtime system invokes whenever a user interaction occurs. The *Record* pod interposes on the execution of these functions. Figure 6.10 shows the architecture of the *Record* pod.

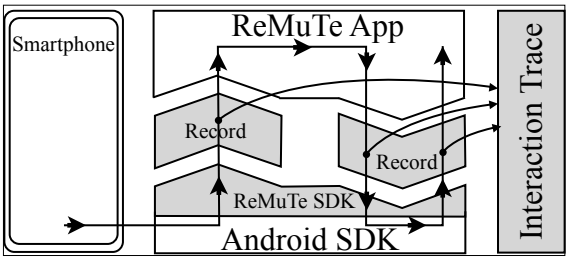


Figure 6.10: The *Record* pod for Android applications.

For example, a program registers a callback function to handle clicks on a button. When the user taps the button, the Android OS invokes the callback function, and ReMuTeDroid records a `Click` event and the identity of the button. In response to the tap, the program may perform an asynchronous task, and ReMuTeDroid logs its result.

6.3.5 Replaying Interaction Traces

Replaying a trace involves the cooperation of the *Record* and *Replay* pods. The *Replay* pod is split in two components: one inside the program that handles replaying the program's requests to the environment; and an external one that replays interaction events originating from the environment, as depicted in Figure 6.11.

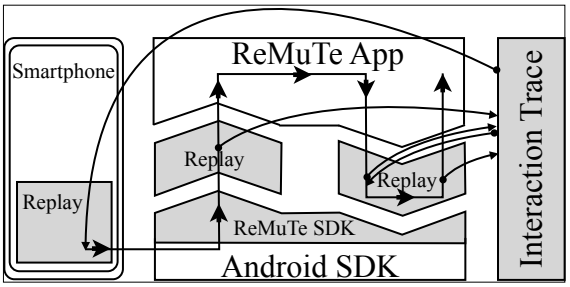


Figure 6.11: The *Replay* pod for Android applications.

The *Record* pod records the order in which events are replayed, and the *Replay* pod checks this order against the order specified in the trace and flags any replay divergence, in which case it stops the replay, or allows replay to continue to the next event.

The *Record* pod represents the foundation for the *Test Execution* pod.

6.3.6 Implementation

This section describes the implementation of the ReMuTeDroid Record, Replay, Test Generation, Test Execution, and Trace Anonymizing pods.

6.3.6.1 The ReMuTeDroid Record Pod

The ReMuTeDroid Record pod uses AspectJ [83] to instrument an Android application to inject into it calls that record the interaction between a user and an application and partially the interaction between the application and its environment, e.g., accesses to the device's geographical location.

AspectJ takes as input a set of pointcuts, which are constructs that specify a control-flow condition in the execution of a program and what code to execute when that condition evaluates to true. An example of a control flow condition is the execution of the method `void android.view.View.OnClickListener.onClick(View view)` that is invoked by the Android runtime when a user clicks on `view`. The Record pod logs the execution of this method as a `Click` event.

Using the ReMuTeDroid Record pod to log most user interactions requires no changes to the source code of an application. The application is instrumented before its classes are translated into the format expected by Android's Dalvik virtual machine. Developers need only to change their build system, to replace the standard Java compiler with the AspectJ compiler.

While the current implementation requires access to source code, one can use tools like `dex2jar` to convert Android programs from Android Dalvik VM to the standard Java classes, use `abc` [120] to instrument the Java classes, and update the Android application with the new code.

Android applications are object-oriented programs, and this poses some challenges to the ReMuTeDroid Record pod. The primary challenge is that each Activity in an Android application is a subclass of `android.app.Activity`. This superclass defines some methods that the Record pod must instrument, e.g., `onResume()`, which denotes that an Activity is starting. Alas, `android.app.Activity` is part of the SDK, so it is impossible to change it. Therefore, developers need to redefine the methods whose execution is logged by the Record pod in each of their Activity classes. We think it is easy to build an automated script that relieves developers of this task.

To record the interaction between the application and the external world, the application needs to use the ReMuTeDroid API, which extends the Android API. Again, this requirement arises because it is impossible to instrument the Android OS's source code.

Finally, whenever an application starts, the ReMuTeDroid Record takes a snapshot of its persistent state. This includes local files, caches, and databases. This snapshot enables a faster replay than replaying all interaction traces since the user installed the application.

6.3.6.2 The ReMuTeDroid Replay Pod

We developed the ReMuTeDroid Replay pod as a JUnit test. To replay an interaction trace, developers need to specify an URL that specifies where the ReMuTeDroid hive saved it.

The Replay pod combines two replay mechanisms. One corresponds to events generated by the user, e.g., click on a UI widget. To replay such events, the pod uses Robotium [28] to trigger the event, and then waits for the ReMuTeDroid Record pod to log that event. Only then does the Replay pod advance to the next event.

The second type of events are those generated by the application, and usually correspond to it interacting with its environment, e.g., reading the device's date. In this case, the replay is reactive, in that it waits for the Record pod to record that event. At that point, control passes to the Replay pod, which returns the logged value.

The Test Generation and the Test Execution pods extend the Replay pod. The Test Execution pod can relax the replay restrictions imposed by the Replay pod, i.e., it can enable the application to communicate with the external world.

6.3.6.3 The ReMuTeDroid Hive, Test Generation, and Test Execution Pods

We embed the Test Generation and Test Execution pods into cloud services. These pods are implemented as JUnit tests that are executed by a Jenkins [32] job. The pods run inside an Android emulator, a virtual machine that simulates an Android device.

The ReMuTeDroid hive is implemented using Bugzilla [121]. Whenever a user trace is sent to the hive, the hive starts the Jenkins job for the Test Generation pod. When the pod generates a new test, the pod sends it to the hive. In turn, the hive starts a different Jenkins job for the Test Execution pod, which runs the test and logs its outcome.

This architecture enables ReMuTeDroid to generate and execute tests in parallel. Furthermore, we can configure Jenkins to run the same test on multiple emulators, each having a different configuration, e.g., a different Android operating system version. We can also run Test Generation pods on Android emulators with different screen sizes and aspect ratios, which may drive the pods to log different neighbors for the same UI widget and, thus, generate different tests.

6.3.6.4 The ReMuTeDroid Trace Anonymizing Pod

The *Trace Anonymizing* pod's job is to ensure that once a user shares an interaction with the ReMuTeDroid hive, that trace cannot identify the user.

Recall that this pod protects users' anonymity against two identification attacks: one that uses explicit information contained in the trace, i.e., in reactive events, and one that relies on user behavior encoded in the trace, i.e., the proactive events.

To protect users' anonymity against identification based on reactive events, we use concolic execution to generate alternative values for the information contained in each reactive event. Our concolic execution engine is based on the S2E selective symbolic execution engine [92] that was modified to run Android applications [122]. S2E is a combination of the QEMU virtual machine [123] and the Klee [124] symbolic execution engine. The Android emulator, too, is based on the QEMU virtual machine.

S2E works in the following way. Certain regions of memory are marked as symbolic. When the program branches on a condition that references a memory location marked as symbolic, S2E makes two copies of the state of the program: one state corresponds to the `true` outcome of the branch condition, while the other corresponds to the `false` branch. In both cases, S2E adds a new constraint on the memory location marked as symbolic that constraints its values to only those that satisfy the branch outcome, i.e., either `true` or `false`. The transformation of this symbolic execution engine into a concolic execution engine requires keeping only the state for which the original, concrete value of the memory location is a solution.

The main challenge in implementing Algorithm 6 is to decide what ReMuTeDroid marks as symbolic. That is, at what granularity to perform concolic execution. There are multiple levels in the software layers of an Android application where one can mark memory locations as symbolic. We are going to explain the design decisions we made using the example of a network communication between the application and a server.

First, one can mark as symbolic the network packets that come from the network, i.e., marking is performed at the OS level. However, as the OS processes these packages, it generates constraints that refer to them. The downside of these constraints is that they reference the OS implementation, which is irrelevant from the point of view of the application, so they unnecessarily reduce the number of alternative events the Trace Anonymizing pod can generate. Thus, we move up the software stack.

The second option is to mark as symbolic the bytes returned by the Java runtime when reading from a network socket, but doing so generates unnecessary constraints. Serialization libraries like Protocol Buffers [125] or Thrift [126] make it easy for developers to ship data between a server and a client without having to care for the serialization and deserialization of data, which is handled by the library. Thus, marking as symbolic the Java bytes received from a server means that the concolic execution engine will generate constraints that reflect the way the library deserializes data, leading to disadvantages similar to the ones of marking OS-level network packets as symbolic.

The general description of the problem that pesters marking low-level bytes as symbolic is that it is at the wrong abstraction level. The application is oblivious to how it received an object from the server, and its decisions are not based on the bytes it received, but rather on the state of the object it received.

This is the reason why the ReMuTeDroid Trace Anonymizing pod marks as symbolic the fields of objects reconstructed from network communication. Doing so enables the pod to collect the minimum, yet necessary, set of constraints that describe the way the application processes server responses.

Our prototype currently supports Thrift. Thrift is a network communication framework that allows developers to define data types in a simple definition file. Taking that file as input, the Thrift compiler generates code used to build remote procedure call (RPC) clients and servers. For Android clients, the result of an RPC call is a Java object. The Trace Anonymizing pod marks as symbolic the fields of this Java object.

6.3.7 Limitations

The main limitation of ReMuTeDroid is that it requires developers to build their applications using the ReMuTeDroid SDK. This is a disadvantage compared to WebErr. However, we believe that this is a better trade-off than that of the alternative of changing the Android operating system and embedding the record and replay functionality inside it. The latter solution dramatically reduces the applicability of ReMuTeDroid. As the Evaluation chapter will show, recording interaction traces does not incur a runtime overhead that significantly impacts user experience (Section 7.7.2).

WebErr and ReMuTeDroid test applications against realistic end user errors. Recall that we target both categories of users of cloud-based applications, i.e., we apply Record-Mutate-Test also to system administrator errors. Next, we describe two tools that focus on these errors.

6.4 Arugula: A Programming Language for Injecting System Administrator Errors in Configuration Files

Misconfiguration is a frequent administrator mistake. Studies have shown that more than 50% of operator errors observed in Internet services are configuration errors [25, 127]. A field study found that 24% of Microsoft Windows

downtime is caused by system configuration and maintenance errors [128]. Not only are they frequent, but misconfigurations also have wide-reaching effects (e.g., a DNS misconfiguration made Microsoft's MSN hosting services unavailable for 24 hours [129]).

Protecting administrators from mis-configuring systems, by checking configuration files against more than just syntactic errors, is difficult, because it requires system developers to reason about what are the errors system operators can make and what are their consequences (which is hard).

We describe Arugula, an error injection language that provides system developers with control over what errors to inject into a configuration file. System developers write Arugula programs that describe what errors to inject, where in the configuration file, and when. These programs run during the Mutate step of Record-Mutate-Test.

6.4.1 Interaction Trees

While each system may have their unique grammar for their configuration file, the hierarchical structure of the configuration file is still there. By removing the peculiarities of a system, such as configuration file format or how to refer to configuration variables, we obtain a hierarchical structure that is common to multiple systems. We define this hierarchy as an interaction tree and inject errors into it.

An interaction tree has the following form. The system's configuration is the root of the tree. Configuration files, sections, and directives are the inner nodes of the tree, while sections' names and the directives' names, separators, and values are its leaves.

Before Arugula can inject errors into a configuration file, that configuration file must be converted to an interaction tree. This job is performed by a *parser* that can be either system-specific or a system-independent. For example, the Apache httpd web server requires a system-specific parser, while for MySQL configuration files, Arugula can use a generic, line-oriented configuration file parser, in which the value of each parameter is specified on one line.

After Arugula mutates the interaction tree, it uses *serializers* to convert it back to a concrete configuration file.

6.4.2 The Arugula Error Injection Description Language

Arugula is an error injection description language. The Arugula runtime runs Arugula programs that specify what errors to inject, which are the error-injection targets, and when to inject the error, e.g., only when the configuration file contains another directive with a certain value.

An Arugula program is a pipeline of Arugula operators, wherein the output of an Arugula operator is the input of its successor. Arugula programs are reproducible (i.e., given the same configuration files, an Arugula program generates the same erroneous configuration files). We describe the Arugula operators next.

6.4.2.1 Arugula Operators

Arugula provides four types of operators that correspond to each step of the error injection process:

- Arugula operators of type I (from input) generate an interaction tree from that system's configuration, i.e., they are parsers.
- Arugula operators of type S (from select) specify where in an interaction tree to inject errors.
- Arugula operators of type M (from mutate) transform the interaction tree by injecting realistic system administrator errors.

- Arugula operators of type O (from output) are the reverse of I-type operators, because they transform a mutated interaction tree into concrete configuration file, i.e., they are serializers.

Table 6.1 presents Arugula's operators. Each operator receives as input a set of elements and outputs another set of elements. Arugula operators of type I read in concrete configurations and output the root node of the interaction tree. Operators of type S and M read and output interaction tree nodes. Finally, O-type operators take in interaction trees and generate system configurations. There are Arugula operators of type R that define subroutines.

Name	**Type**	**Additional parameters**	**Description**
Parser	*I*		Generate an interaction tree
Serializer	*O*		Generate a system's configuration
XPathSelect	*S*	cond: XPath expression	Return the input nodes that match the *cond* XPath expression
Random	*S*	count: integer	Return maximum *count* input elements
Parent	*S*		For each input node, return its parent, if one exists
Delete	*M*		Delete input nodes
Copy	*M*	where: XPath expression	Copy input nodes to XPathSelect(*where*)
Insert	*M*	what: abstract representation	Insert *what* into each input node
Move	*M*	where: XPath	Move input nodes to XPathSelect(*where*)
ModifyText	*M*	func: string mutation function	Apply *func* to input nodes' values
ReplaceBasedOnDictionary	*M*	dictionary: set of pairs of words	Replace input nodes' values with matching words from *dictionary*
BreakSemanticRelation	*M*	expression: a boolean formula	Inject errors into input nodes until *expression* evaluates to false
BreakGrammar	*M*	grammar: a grammar defining correct values	Inject errors into input nodes until their values no longer obey *grammar*
ForEach	*R*	transf: *pipeline*	For each input node, run the *transf* pipeline
DoUntilUniqueConfigurations	*R*	count, tries, transf: *pipeline*	Run *transf* until *count* different configurations or *tries* attempts made
Branch	*R*	transf: $set < pipeline >$	Run all pipelines $p \in transf$ with a copy of the input

Table 6.1: The Arugula operators.

An Arugula program begins with an I-type operator, followed by a combination of S-, M-, and R-type operators and finishes with an O-type operator. While Table 6.1 only contains one I-type and O-type operator, there are multiple such operators.

To select where to inject errors, one can use three S-type operators: `XPathSelect`, `Random`, and `Parent`. Table 6.1 contains their description. These operators return a set of tree nodes.

The `Delete`, `Insert`, `ModifyText`, and `ReplaceBasedOnDictionary` operators correspond to the user error operators defined in Section 3.5.

The `BreakSemanticRelation` operator takes in an expression that describes a relation between various configuration values and injects errors into them until the expression evaluates to false. One can use this operator, for example,

```
1   <ArugulaProgram configuration="c.config">
2     <LineParser/>
3     <XPathSelect cond="//directive"/>
4     <ForEach>
5       <DoUntilUniqueConfigurations n="1">
6         <ModifyText func="SwapLetters"/>
7       </DoUntilUniqueConfigurations>
8     </ForEach>
9     <LineSerializer/>
10  </ArugulaProgram>
```

Figure 6.12: An Arugula program that generates configurations in which each directive has a typo injected into it.

to break the PostgreSQL constraint $max\_fsm\_pages \geq 16 \times max\_fsm\_relations$ [130]. This operator will inject errors into the two directive's values until the constraint no longer holds.

Arugula operators of type R act as subroutines. They take as parameter Arugula pipelines, run them, and then gather new, modified interaction trees. The `ForEach` operator runs its parameter pipeline once for every input node.

The `Branch` operator enables developers to inject multiple errors in an interaction tree, by running the passed-in pipelines with copies of the input. For example, after selecting a directive, one can delete that directive, inject a typo in that directive's value, or copy that directive into another section.

Arugula can easily be extended to support new operators. This is important, as we do not claim we model every administrator error.

6.4.2.2 Arugula Example

Figure 6.12 presents an example of an Arugula program. Line 1 declares that the Arugula program works on the "c.config" configuration file. In line 2, that configuration is parsed into the interaction tree by a line-oriented parser. Next, in line 3, the configuration's directives are selected. Next, for each directive, the program injects the single typo of swapping two adjacent letters (lines 4-8). Finally, line 9 transforms the modified interaction trees into concrete configurations.

6.4.3 The Arugula Runtime

Arugula is an interpreted language: the Arugula runtime reads an operator, executes it, saves its output, and then proceeds to the next operator. Arugula programs are reusable: they have a "link" parameter that specifies where on the Internet to find their source code, i.e., the sequence of Arugula operators and the values of their parameters. This opens up the possibility of having a central repository of injection scenarios to which developers can contribute their own Arugula programs and reuse others' programs.

Arugula, similar to ConfErr, the tool that inspired it, ends up generating many configurations that are syntactically invalid and are rejected by the configuration parser, hurting the tools' efficiency. In the next section we describe Sherpa, which one can think of as a smarter `BreakGrammar` Arugula operator that automatically learns the grammar of configuration files, so that it generates valid configurations.

6.5 Sherpa: A Tool for Efficiently Testing Software Systems Against Realistic System Administrator Errors

In this section, we improve upon Arugula to generate configurations that pass a system's configuration parser barrier, a part of the system that checks if a configuration file is syntactically correct.

The new technique, called Sherpa, learns what constitutes a valid configuration by observing how the parser processes one. It then uses this information to generate new configurations that follow the same path through the parser, but contain system administrator-realistic errors that affect the system's logic.

Our technique uses dynamic program analysis to collect the constraints a parser places on the values of configuration directives, then expresses the effects of errors also as constraints, and uses a constraint solver to generate configurations that satisfy both sets of constraints, and thus, are valid configurations.

Sherpa does not require access to source code, which is why it can test proprietary systems, like Microsoft IIS. We prototyped our technique in the Sherpa web service.

6.5.1 Sherpa Design

We provide an overview of our technique to automatically generate mutated configurations that are accepted by a system's configuration parser in Figure 6.13.

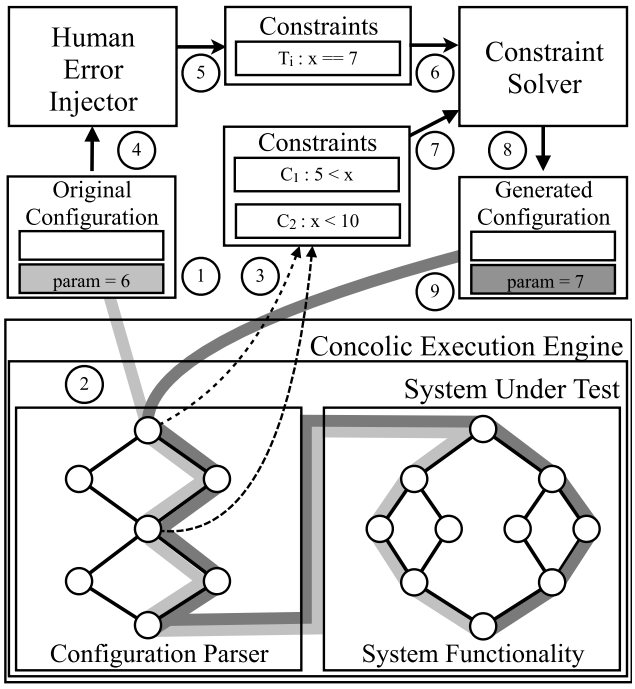


Figure 6.13: Overview of Sherpa. Light gray lines show the execution path through the system's control flow graph with an original configuration. Dark gray lines show the execution path with a generated configuration.

We partition a system into configuration parser and system functionality (bottom of Figure 6.13). The configuration parser is the code executed before the system is ready to operate, e.g., before a web server starts handling requests.

The parser encodes a grammar that defines acceptable configurations. We can use existing valid configurations

as guides ("sherpas") through the parser code to automatically infer part of this grammar. The grammar is useful to generate configurations that follow the same code path through the parser as the initial configuration, but then cause the system to exercise different behavior.

Sherpa infers the grammar of a single configuration parameter, i.e., the value of a configuration directive, at a time. It selects the target parameter P ①, then runs the system inside a concolic execution [85] engine ②. When the configuration parser code branches on the value of parameter P, the concolic execution engine computes an associated path constraint ③. These constraints determine a set of valid values for that token, i.e., they encode part of the grammar accepted by the parser.

Next, Sherpa feeds the value of parameter P to the *Human Error Injector* ④ that applies the user error operators (Section 3.5) to P to obtain a set M of parameters P_i that are mutants of P. Sherpa expresses the tokens P_i as constraints ⑤. We focus on spelling mistakes in configuration parameters. They manifest as insertions (an extra letter appears in a token), omissions (a letter disappears), substitutions (a character is replaced with another character), and transpositions (two adjacent letters are swapped).

Then, Sherpa uses a constraint solver to select from the set M of mutant parameters ⑥ only those that satisfy the constraints C collected by the concolic execution engine ⑦, and generates new configurations by substituting the mutants for the original parameter ⑧, which may cause the system to execute different code ⑨.

Finally, developers test their system using the generated configurations and identify opportunities to strengthen the configuration checks performed by their system. Sherpa sorts the generated configurations based on the frequency of the errors they contain (Section 3.9.4), which helps developers prioritize the more likely misconfigurations.

6.5.2 Implementation

We prototyped Sherpa as a web service. To generate valid misconfigurations, developers choose a web server, say Apache httpd [131], Lighttpd [132], or Microsoft IIS 7 [133], a configuration directive, and specify its correct value.

The major benefit of providing Sherpa as a web service is that its understanding of what is a valid configuration parameter can grow with each new use, because of different correct configuration parameter values it receives as example. This in turn, helps it reduce the number of false positives, i.e., valid configurations that Sherpa deemed invalid. Another advantage is that Sherpa can reuse the constraints it learned for a configuration parameter across requests from multiple developers, which reduces the time to generate configurations.

Sherpa uses a concolic execution engine based on S2E [92] that works on binary programs. Modifying S2E to run concolically requires discarding the program states corresponding to branch targets not taken. The modifications required of S2E are similar to the ones described in Section 6.3.6, which describes the implementation of ReMuTe-Droid's *Trace Anonymizing* pod.

Sherpa expresses mutant tokens as constraints using Hampi [134], an SMT solver that implements the theory of strings. We modified Hampi to return all tokens that satisfy the constraints collected during concolic execution, to enable developers to test their system against all configurations that generate the same execution path within the configuration parser, but generate different paths within the rest of the system.

6.5.3 Limitations

Arugula is able to inject errors in the structure of a configuration file or semantical errors that are consequences of wrong reasoning, while Sherpa currently does not. For structural errors, Sherpa could collect path constraints over the configuration file's hierarchy, but we suspect its effectiveness will be low, because systems place few constraints on

the hierarchy's structure. For semantic errors, Sherpa could use concolic execution to understand how configuration parameters relate to each other.

Since Sherpa learns only part of the grammar accepted by a configuration parser, it will reject valid, mutated configurations not covered by the learned part. One can use rejected tokens as probes, and if they are accepted by the system, use them to extend the collected path constraints. Another means to alleviate this drawback is to deploy Sherpa as a web service. This enables it to learn new parts of the grammar accepted by the configuration parser, which may reduce the number of valid configurations it rejects.

Finally, a limitation of our Sherpa prototype is that it stops concolic execution when the system under test interacts with its environment, meaning that it will miss the checks a system does.

6.6 Chapter Summary

This chapter presented four tools that implement the Record-Mutate-Technique:

- WebErr is a tool for testing modern web applications. The key idea behind WebErr is to embed the recording functionality inside a web browser. The advantage of doing so is that developers need not change their application.
- ReMuTeDroid is a crowdsourced testing infrastructure that enables smartphone application developers to harvest realistic user interaction traces from more than a billion users, but also to use these users' devices to run the tests.
- Arugula is a user error description language that enables developers to test how resilient their system is to realistic system administrator configuration errors.
- Sherpa improves upon Arugula by generating configuration files that pass initial configuration checks performed by a system's configuration parser.

All these tools are automated in that test generation does not require developer effort. The tools are also scalable both in the number of systems they can be applied to and in the size of the systems they can test. Finally, the tools are accurate in that they inject errors that are realistic and representative of the errors real-world users make.

Having seen what are the tools we built using Record-Mutate-Test and how they operate, we now turn our attention to the evaluation of Record-Mutate-Test.

Chapter 7

Evaluation

7.1 Evaluation Criteria

In this chapter, we report on empirical and simulated evaluations of the four properties we believe a testing solution must have:

- *Accuracy*. An accurate technique is realistic, relevant, effective, and efficient.
  - *Realism*. It is impractical to test a system against every input. Thus, to be practical, the solution needs to focus on a subset of important inputs. In this thesis, we focus on testing a system's resilience to user errors, i.e., against those inputs that have been affected by user errors. Alas, such testing is equivalent to testing all a system's inputs, because one can argue that any input is the consequence of a user error. Instead, we limit testing against those user errors that are representative of the erroneous behavior of real-world users.
  - *Relevance*. The technique should ensure that the tests it generates focus on system components users most rely on.
  - *Effective*. The technique should be able to discover bugs triggered by user errors.
  - *Efficiency*. The technique should use the minimum number of tests to achieve maximum effectiveness.
- *Automation*. Employing the technique and its associated tools should require minimal developer effort. There exist two dimensions to this:
  - *Automated test generation*. Developers should not have to specify tests.
  - *Automated test (re)execution*. Developers should be able to automatically run tests. For this to work reliably, the technique must generate reproducible tests.
- *Scalability*. The technique should scale along two dimensions:
  - *System size*. The technique should be able to test software systems that are comprised of millions of lines of code.
  - *Portability*. A portable testing technique is able to express user errors independently from the system it is testing, making the technique applicable to a large set of systems.

- *Protect users' anonymity*. The technique leverages real users' interactions with a system to generate tests and guide the testing effort, so one should not be able to determine the identity of a user based on an interaction trace shared by the user or on tests generated from that trace.

As discussed in Chapters 3 and 6, the tools implementing Record-Mutate-Test provide automated test generation and execution. We previously covered the technique's realism (Section 3.10).

In the rest of this chapter, we evaluate Record-Mutate-Test's scalability (Section 7.2), effectiveness (Section 7.3), efficiency (Section 7.4), relevance (Section 7.5), and its protection of user anonymity (Section 7.6). Finally, we evaluate practical aspects of the tools we built using Record-Mutate-Test (Section 7.7).

7.2 Scalability

Recall that we defined scalability as the property of a technique that enables it to test a wide variety of systems independently of their size.

To show that Record-Mutate-Test possesses this property, we describe the systems to which we applied it. Table 7.1 shows the name of each system, a short description, what programming language was used to develop them, and their size, in terms of lines of source code.

Name	**Description**	**Programming language**	**Size (KLOC)**
PocketCampus	Android mobile device application that improves life on EPFL's campus	Java	30
Secrets [135]	Android mobile device application to securely store and manage passwords and secrets	Java	6
Addi [136]	Android mobile device computing environment like Matlab and Octave	Java	24
AnkiDroid [137]	Android mobile device flashcard application	Java	32
GMail	Web-based email application	Javascript	443 [138]
Google Drive	Web-based productivity suite	Javacript	N/A
Google Sites	Web-based wiki- and web page-creation application	Javascript	N/A
Apache httpd 2.0.53	Web server	C	192 [139]
Lighttpd 1.4.11	Web server	C	34
Microsoft IIS 7	Web server	N/A	N/A

Table 7.1: The systems to which we applied tools embodying the Record-Mutate-Test technique.

The programs in Table 7.1 show that Record-Mutate-Test scales in the size of the systems it can test and can be applied to a wide variety of programs that differ in their functionality and in the primary programming language used to develop them.

7.3 Effectiveness

In this section we evaluate the effectiveness of Record-Mutate-Test.

The best way to measure how effective is Record-Mutate-Test is to present the deficiencies in handling user errors the technique discovered. A proxy way of measuring effectiveness is to show how the tests it generates achieve higher

code coverage compared to the interaction trace from which they were generated. An improvement in code coverage shows that user errors affect the execution of a program and can exercise new code, which can contain bugs.

7.3.1 Testing Web Applications Against End User Errors

We use WebErr to test the resilience of web search applications to typos in search queries, a frequent user error. We want to test how well three web search engines, Google, Bing, and Yahoo!, handle such typos. We choose 186 frequent queries, from New York Times's top search keywords [140] and Google Trends's list of top searches [141]. Next, we inject a typo into each search query, perform the searches, and measure the number of errors detected by each search application. Table 7.2 presents the results.

Search engine	**Google**	**Bing**	**Yahoo!**
Percentage	100%	59.1%	84.4%

Table 7.2: The percentage of query typos detected and fixed by the Google, Bing, and Yahoo! web search engines.

A low resilience to typos exposes users to security threats. Reports of web analytics and personal experience show that users type complete URLs into search engines instead of the browser's address bar [142]. This is effective, because the web search applications will provide a link that users can click to get to the intended website. Alas, if the user makes a typo in the URL, and the web search application does not correct it, then users are exposed to typosquatting. Typosquatting refers to the practice of registering domain names that are typo variations of popular websites [143]. These domains can be used for phishing or to install malware [143].

7.3.1.1 Bug Found

We tested Google Sites against timing errors and found a bug. When editing a Google Sites website, one has to wait for the editing functionality to load. In our experiment, we simulated impatient users who do not wait long enough and perform their changes right away. In doing so, we caused Google Sites to use an uninitialized JavaScript variable, an obvious bug.

7.3.2 Testing Android Applications Against User Errors

We modified four Android applications, i.e., PocketCampus, Secrets, Addi, and AnkiDroid, to use ReMuTeDroid. We recorded interaction traces using the four applications. We then fed these interaction traces to ReMuTeDroid as sources for test generation. In this section, we report the bugs found and the code coverage improvement obtained by running the tests ReMuTeDroid generated.

Table 7.3 describes the source interaction traces, while Table 7.4 describes the errors ReMuTeDroid injects.

7.3.2.1 Bugs Found

The tests generated by the *Test Generation* pod uncovered two bugs in PocketCampus. Both bugs cause PocketCampus to crash. These were previously unknown bugs, and they were acknowledged by the PocketCampus developers, of which the author is one.

The first bug results in a `NullPointerException` exception that crashes the application. PocketCampus enables users to browse the dishes offered by campus cafeterias. Users have the option to filter dishes based on the campus

Trace id	Interaction trace description
1	PocketCampus - Check student card balance
2	PocketCampus - Edit public transport destination
3	PocketCampus - Browse dishes based on their ingredients
4	PocketCampus - Select displayed restaurants
5	PocketCampus - Browse the day's menus and vote for a dish
6	PocketCampus - Search for a person
7	PocketCampus - Set up initial public transport destination
8	PocketCampus - Read news
9	PocketCampus - Change News settings
10	Secrets - Store password for website
11	Addi - Compute mathematical expression
12	Addi - Compute complex mathematical expression
13	Anki - Edit and solve quiz question

Table 7.3: The interaction traces used as seeds for generating tests.

Acronym	Name and description
C	Capture: replace the user actions performed on an `Activity` with other actions performed on the same `Activity`
DT	Device turned: insert an event that simulates the user tilted the device to landscape and then back to portrait mode
RI	Random interaction: insert a random sequence of 100 user interactions
ST	Side tracking: insert previously executed user interactions
PH	Press `home`: insert an event that simulates the user pressed the `home` button, and then returned to the application
S	Swap order of steps: reverse the order of two consequent siblings in an interaction tree
T	Typos: modify text input to contain typos
US	User scrolled up and down: insert an event that simulates the user scrolled all the way down and then back up in a list 20 times
NA	Nearby widget after: insert an event that interacts with a neighbor of a UI widget immediately after the event that interacted with that widget
NB	Nearby widget before: insert an event that interacts with a neighbor of a UI widget immediately before the event that interacts with that widget
R	Replace: replace an event with one that interacts with a neighbor of the UI widget with which the original event interacted
MIB	Move user interaction before: swap a user-generated event with a previous event denoting an asynchronous request finished

Table 7.4: The errors injected by ReMuTeDroid.

cafeteria that provides them. For this, PocketCampus traverses the list of dishes, reconstructs the list of cafeterias, and allows users to choose cafeterias. The *Test Generation* pod generates tests in which users invoke this cafeteria-filtering functionality *before* PocketCampus finishes downloading the list of dishes. This causes the application to traverse a `null` list, generate a `NullPointerException` exception, and crash.

The second bug triggers an `OutOfMemoryError` exception that also crashes the application. PocketCampus allows

users to read the latest EPFL news. News items are presented in a list and for each item, the application downloads an image. To make the application responsive to users' interactions, images are downloaded only when their news item becomes visible. One of the generated tests simulates the user scrolling through the list of news items all the way down and back up 20 times, which makes PocketCampus use too much memory, and crash.

7.3.2.2 Code Coverage Improvement

We computed the code coverage obtained by running the tests generated by ReMuTeDroid. The results are in Table 7.5. Its columns identify each trace, the number of tests that were generated based on it, the initial trace's code coverage, the coverage obtained by running the tests, and the improvement in code coverage.

Trace id	Number of tests	Initial coverage (LOC)	Test coverage (LOC)	Improvement
1	92	2,496	7,826	3.1×
2	885	3,285	7,283	2.2×
3	168	2,796	7,749	2.8×
4	236	2,642	8,143	3.1×
5	249	2,678	7,800	2.9×
6	660	2,281	6,900	3.0×
7	657	2,855	7,164	2.5×
8	117	2,121	7,845	3.7×
9	183	2,370	7,588	3.2×
10	2,591	2,188	2,582	1.2×
11	429	4,236	5,016	1.2×
12	193	4,435	5,333	1.2×
13	1,269	5,362	7,799	1.5×

Table 7.5: The interaction traces used as seeds for generating tests and the code coverage improvement obtained by running those tests.

The code coverage improvement varies from 20% to 270%. This shows that Record-Mutate-Test automatically generates tests that significantly increase code coverage.

To understand where the code coverage improvement comes from, we present a breakdown of the code coverage achieved by the tests corresponding to each type of injected error. Table 7.6 contains the results. Each column, starting with the second one, corresponds to an error described in Table 7.4.

The results show that the biggest code coverage improvement comes from using the Capture error model. This is not surprising, because injecting this error causes a part of an interaction trace to become part of a test generated from a different interaction trace. The error type that increased second most the code coverage is the insertion of random events. Next, errors that cause users to tap on wrong UI widget or commit typos caused a 20% increase in the code coverage.

We conclude that even in the absence of existing interaction traces that can boost code coverage, ReMuTeDroid is able to generate tests that increase by 50% the code coverage, relative to the source interaction trace.

Trace id	C	DT	RI	ST	PH	S	T	US	NA	NB	R	MIB
1	2.7×	1.1×	1.7×	1×	1.1×	1.1×	N/A	1×	1.2×	1.2×	1.2×	0.9×
2	2×	1.2×	1.4×	1.2×	1.2×	1.2×	1.2×	1.1×	1.3×	1.3×	1.3×	1×
3	2.4×	1.2×	1.6×	1×	1.2×	1×	N/A	1×	1.3×	1.4×	1.4×	0.8×
4	2.9×	1.2×	1.6×	1.1×	1.2×	1×	N/A	1×	1.3×	1.4×	1.4×	BUG
5	2.6×	1.2×	1.4×	1.1×	1.2×	1.1×	N/A	1.1×	1.2×	1.2×	1.2×	1×
6	2.5×	1.2×	1.9×	1.2×	1.2×	1.2×	1.1×	1.2×	1.6×	1.6×	1.6×	1×
7	2.5×	1.2×	1.5×	1.2×	1.2×	1.1×	1.2×	1.2×	1.2×	1.2×	1.1×	0.9×
8	3.6×	1.1×	1.6×	1.1×	1.2×	1×	N/A	BUG	1.1×	1.1×	1×	1×
9	3.1×	1.2×	1.5×	1.1×	1.2×	1.1×	N/A	1×	1.2×	1.3×	1.2×	0.7×
10	N/A	1×	1.1×	1×	1.1×	1×	1×	1×	1×	1×	1×	N/A
11	N/A	1×	1×	1×	1.1×	1.1×	1.1×	1×	1×	1×	1×	N/A
12	N/A	1×	1.1×	1×	1.1×	1.1×	1.1×	1×	1×	1.1×	1×	N/A
13	N/A	1×	1.3×	1×	1×	1×	N/A	1×	1×	1.2×	1.2×	N/A
Average	2.7×	1.1×	1.4×	1.1×	1.1×	1.1×	1.1×	1.1×	1.2×	1.2×	1.2×	0.9×

Table 7.6: Code coverage improvement per injected error type.

7.3.3 Testing Web Servers Against System Administrator Errors

In this section, we show how realistic misconfigurations can cause the Apache httpd, Lighttpd, and Microsoft IIS 7 web servers to lose functionality or to reduce the security of the websites they host.

Our experimental setup is the following. We configure each web server to host a simple website that is served through HTTPS. The website is composed of a public welcome page and a protected page. The web server controls access to the protected page, by checking that users provide a valid and unique pair of username and password. Failing to provide valid credentials causes the server to respond with an `Unauthorized` 401 HTTP code [144]. The configuration file specifies the valid credentials.

To be able to detect the impact misconfigurations have on the functionality of a web server, we need a baseline for normal behavior. For this, we use two security auditing tools, w3af [145] and skipfish [146]. These tools crawl a target website and discover URLs served by its hosting server. Each auditing tool performs a sequence of HTTP requests to the target web server. A request contains the accessed URL, the method (e.g., `GET`), and an optional payload, for `POST` requests. The tools save the response HTTP status code (e.g., `200`) for each request.

We run each web server in a virtual machine. We reset its state when changing server configurations. The security auditing tools run on the host of the virtual machine.

We run the security auditing tools on each server when using our default configuration file. We record the tools' requests and the server's responses. We consider them as ground truth. We use the differences in the return codes for the same request as means to identify changes in a server's behavior.

Next, we use Sherpa to mutate each configuration parameter in each configuration file of each web server. Exceptionally, for IIS 7, we use Sherpa on a subset of its configurations parameters, because not all of its parameters affect our website. We inject up to two errors into each configuration parameter. Table 7.7 contains details regarding the number of parameters and the number of mutated configurations on which we ran tests. Note that we experiment with only a subset of all generated configurations.

For each web server, we randomly choose a configuration parameter and then randomly choose a configuration file that contains a mutated value of that parameter. Next, we run the security auditing tools on a server when using this mutated configuration. That is, we re-run the initial experiment to detect the baseline, but this time, we use a mutated

Web server	Number of parameters	Number of tests run
Apache httpd 2.0.53	19	1,451
Lighttpd 1.4.11	25	2,938
Microsoft IIS 7	20	2,863
Total	64	7,252

Table 7.7: Web server experiment details.

configuration.

Finally, we match the requests issued by the security auditing tools when the servers used the original configurations to those issued when the servers used the mutated configurations, and compare their responses. In the rest of this section, we highlight the results we obtained after performing the comparisons.

First, we compute the percentage of all tested, mutated configurations that cause a web server to lose functionality. We say that a web server lost functionality either when an HTTP request disappears from the logs or when the return code of the same request changes. We restrict our comparisons to requests that were originally successfully processed, i.e., they returned the HTTP `200` status code, which corresponds to OK.

We use the percentages to compare the three web servers. Figure 7.1 shows the results for the situation when requests disappeared. On top of the bars we show the number of configurations for which functionality is lost. The results show that errors in configuring IIS 7 have the highest probability of causing functionality loss.

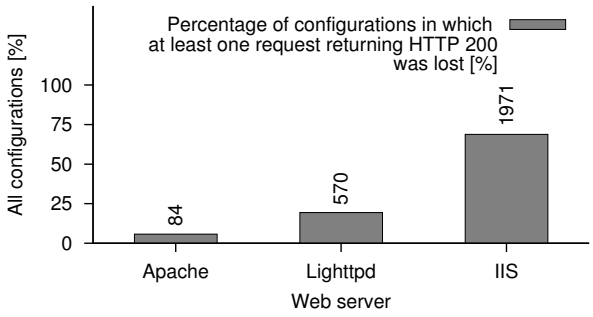


Figure 7.1: Comparison between Apache, Lighttpd, and IIS 7 w.r.t. mutated configurations that lead to requests disappearing.

We compare the three web servers with respect to the percentage of mutated configurations that cause a request originally returning the HTTP `200` status code to now return the HTTP `500` status code. This latter code corresponds to "Internal Server Error" and indicates that something inside the server malfunctioned. Figure 7.2 shows the results. On top of the bars we show the number of configurations that cause server malfunctions. The results show that errors in configuring IIS 7 have the highest probability of causing server malfunctions.

Second, we compute the percentage of configurations that cause the protected page to become publicly accessible, i.e., to no longer be protected. This situation manifests as requests for that page changing their status code from `401` to `200`. Figure 7.3 shows the results. On top of the bars we show the number of configurations in which the protected page becomes public. The results show that errors in configuring Lighttpd have the highest probability of unwillingly granting public access to protected pages.

We perform the reverse experiment, in which we configure the security auditing tools to access the protected web page using valid credentials. Now, requests to the protected page should return the `200` HTTP status code. We

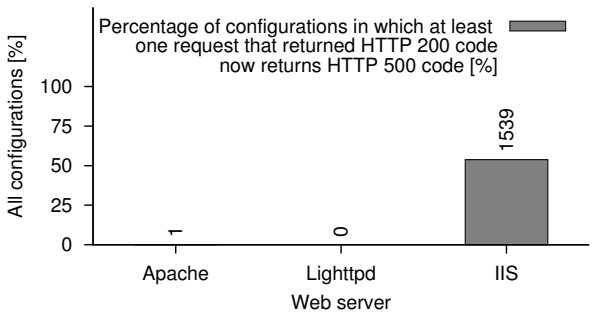


Figure 7.2: Comparison between Apache, Lighttpd, and IIS 7 w.r.t. mutated configurations that lead to requests whose HTTP status code changed from `200` to `500`.

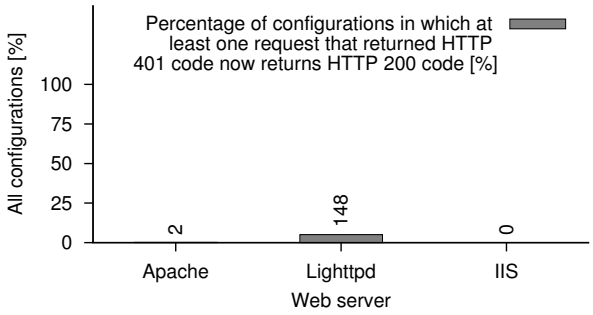


Figure 7.3: Comparison between Apache, Lighttpd, and IIS 7 w.r.t. mutated configurations that lead to requests whose HTTP status code changed from `401` to `200`.

compute the percentage of configurations that cause these requests to no longer return the `200` code. Figure 7.4 shows the results. On top of the bars we show the number of configurations that prevent users from accessing the protected page. The results show that errors in configuring IIS 7 have the highest probability of unwillingly removing legitimate access to protected pages.

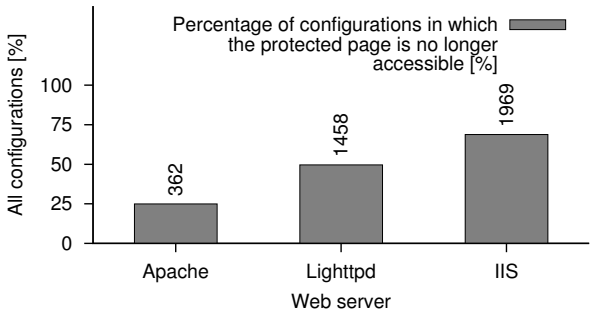


Figure 7.4: Comparison between Apache, Lighttpd, and IIS 7 w.r.t. mutated configurations that prevent legitimate users from accessing the protected page.

Finally, we compute the percentage of configurations that affect the performance of a web server. We use ApacheBench [147] and configure it to send 100 requests to each web server, with a concurrency level of 10. We

compute the average response time when using the original configuration files. We then count in how many cases the average response time for a web server using a mutated configuration is at least an order of magnitude longer compared to using the original configuration. Figure 7.5 shows the results. On top of the bars we show the number of configurations that impact performance. The results show that errors in configuring IIS 7 have the highest probability of causing a severe slowdown in response time.

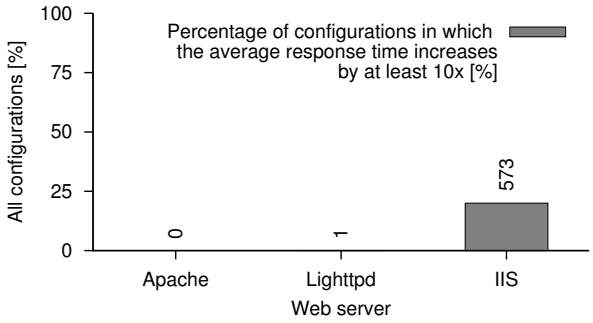


Figure 7.5: Comparison between Apache, Lighttpd, and IIS 7 w.r.t. mutated configurations that cause at least a 10× slow-down in response time.

The experiments described in this section enable us to conclude that no web server is completely resilient to system administrator errors. Such errors can cause all servers to lose functionality.

7.3.3.1 Bug in Apache httpd

The mutated configurations generated by Sherpa lead us to rediscover a bug in Apache httpd that affects "all versions from 2.0 up to 2.0.55. When configured with an SSL vhost with access control and a custom error HTTP `400` error page, Apache allows remote attackers to cause a denial of service (application crash) via a non-SSL request to an SSL port, which triggers a NULL pointer dereference" [148].

Initially, we configured Apache with a custom handler for requests that generate the `409` HTTP status code. Sherpa generated 187 errors starting from the string "409." Then, using the constraints gathered during concolic execution, it pruned them down to three values: 400, 407, and 408, of which the first one causes the crash.

7.4 Efficiency: Reducing the Number of Tests

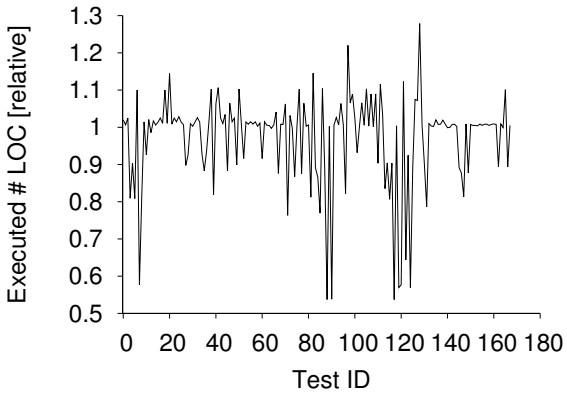
Recall that we say that a technique is efficient if it can achieve high effectiveness by using a reduced set of tests.

7.4.1 Using Test Clustering

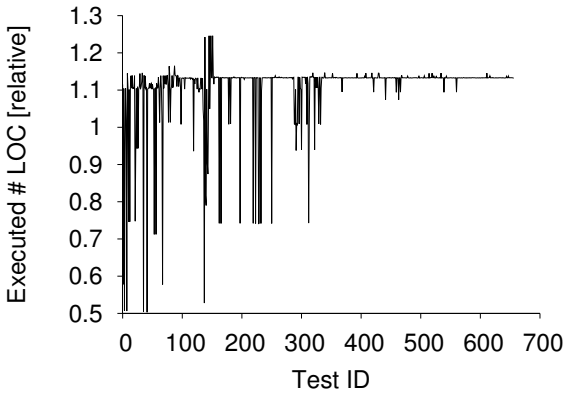
One way to improve efficiency is to discard redundant tests that cause the same lines of code to execute. We can cluster tests and run only the representative of each cluster, with the assumption that all tests from a cluster cover the same lines of code. In this section, we evaluate if this assumption is true.

We investigate what is the variance in the code coverage value obtained by individually running each test generated by Record-Mutate-Test. We can use variance as an empirical evaluation of the redundancy of generated tests. The rationale is that if a numerous fraction of the tests achieve the same code coverage value, then those tests, except for one, are redundant, because running them does not cause the application to explore new behavior.

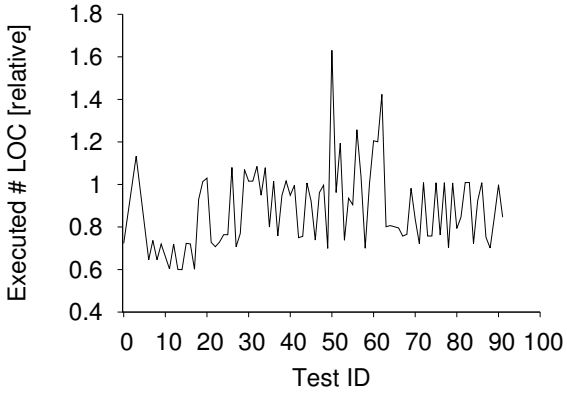
Figure 7.6 shows the results for the tests generated starting from the "Browse dishes based on their ingredients" (Figure 7.6a), "Set up public transport destination" (Figure 7.6b), "Check student card balance" (Figure 7.6c), and "Search for a person" (Figure 7.6d) interaction traces from Table 7.5.



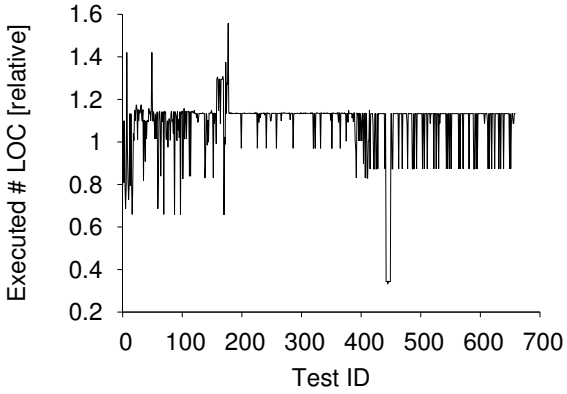
(a) The code coverage of each test generated from the "Browse dishes based on their ingredients" interaction trace.



(b) The code coverage of each test generated from the "Set up public transport destination" interaction trace.



(c) The code coverage of each test generated from the "Check student card balance" interaction trace.



(d) The code coverage of each test generated from the "Search for a person" interaction trace.

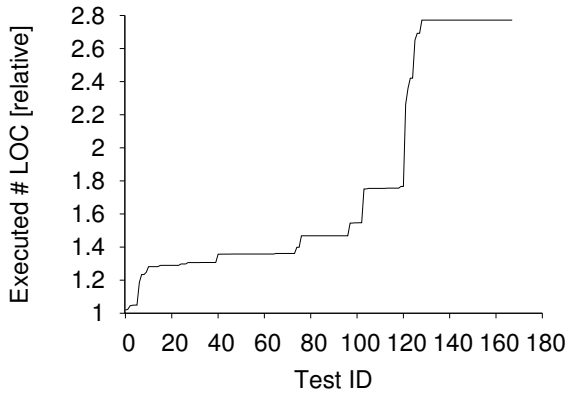
Figure 7.6: The code coverage obtained by an individual test relative to the coverage obtained by its originating interaction trace.

The results show a greater variance in the code coverage obtained by each test in Figures 7.6a and 7.6c than in Figures 7.6b and 7.6d. An explanation for these results is that typos in the name of a train station or a person are a large representative of the generated tests. Each typo generally yields an invalid train station or person name, which is processed identically by the application. This makes testing PocketCampus for each typo of little value.

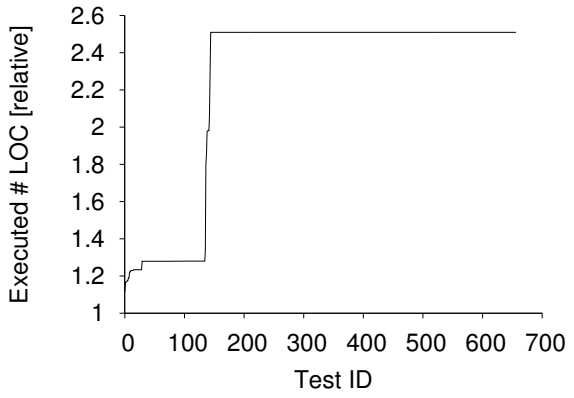
The drawback of using the simple code coverage metric to assess the redundancy of generated tests is that it does not take into account what source code lines were covered. If each test causes an application to execute a set of 100 lines of code, but the sets are disjoint, then the heuristic wrongfully classifies them as being redundant.

We set out to test whether this was the case for the tests ReMuTeDroid generates. In this experiment, we track how the code coverage achieved by tests evolves with each newly ran test. We use the same interaction traces as in Figure 7.6. The results are shown in Figure 7.7.

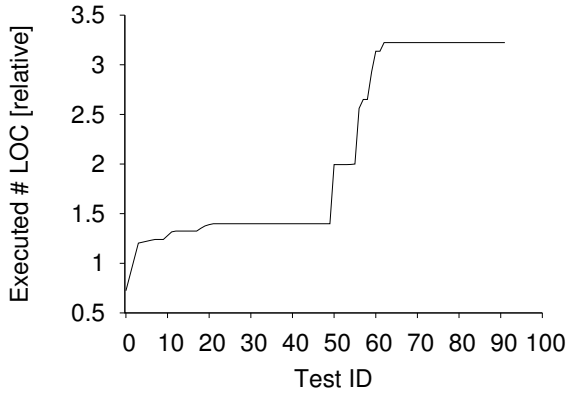
The results show that many tests are redundant since they do not lead to an increase in code coverage. One can



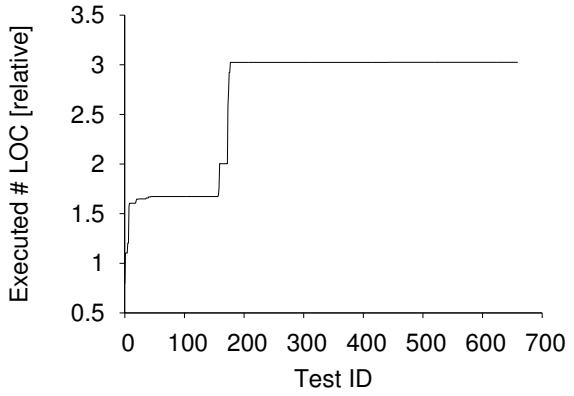
(a) Code coverage evolution for tests generated from the "Browse dishes based on their ingredients" interaction trace.



(b) Code coverage evolution for tests generated from the "Set up public transport destination" interaction trace.



(c) Code coverage evolution for tests generated from the "Check student card balance" interaction trace.



(d) Code coverage evolution for tests generated from the "Search for a person" interaction trace.

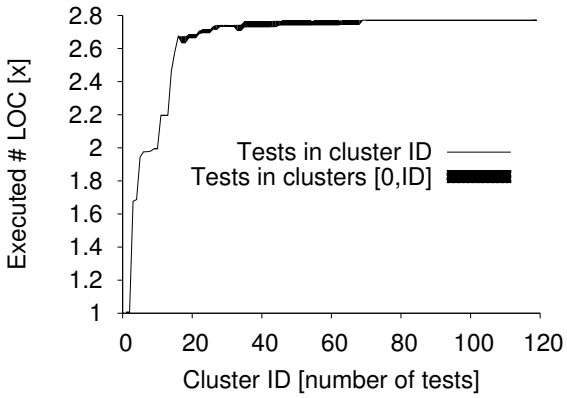
Figure 7.7: The cumulative code coverage achieved by tests relative to the coverage obtained by their originating interaction trace.

construct a subset of all tests that achieve maximum coverage more quickly by discarding the tests in Figure 7.7 that do not increase code coverage. Alas, this solution is useful only for regression tests, because there already exists information about which tests are redundant.

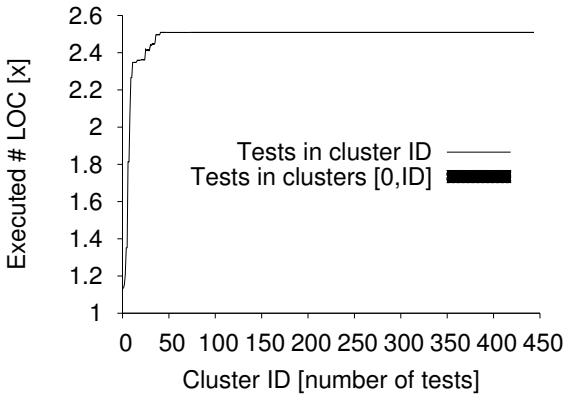
However, we are interested in seeing if applying the clustering algorithm described in Section 3.9.3 helps Record-Mutate-Test select a reduced set of tests that still achieve the same code coverage as the entire test suite.

In this experiment, we generate clusters of size K, starting with 1. For each value of K, we compute two code coverage values: 1) one achieved by running only the medoids of the K clusters, and 2) one achieved by running the medoids of the current K clusters, but also of all previous clusters, i.e., the K-1, K-2, $\cdots$, 1 clusters. We report the results in Figure 7.8.

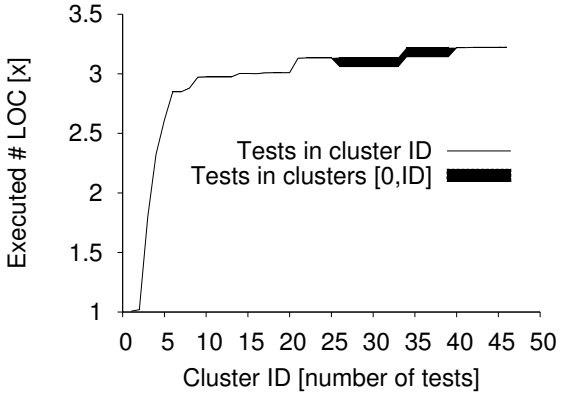
The results show that using the complete link agglomerative clustering algorithm to select which tests to run causes the maximum code coverage to be achieved more quickly, compared to Figure 7.7. Figure 7.8 shows that code coverage is not monotonically increasing when using only the medoids of the K clusters, but not those of previous clusters. The explanation is that when moving from K to K+1 clusters, some of the K medoids are replaced with others that cause the application to execute less code. To compensate for this code coverage drop, Record-Mutate-Test runs all the new



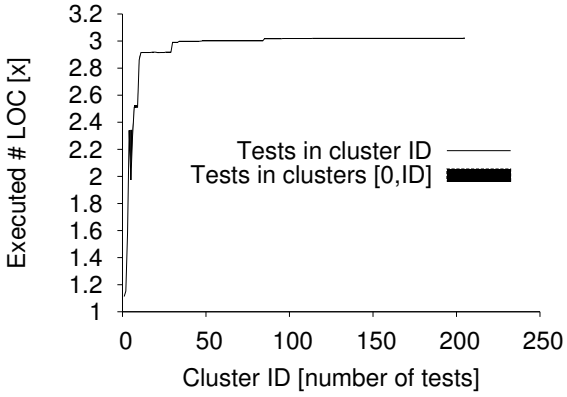
(a) Code coverage evolution for clusters composed of tests generated from the "Browse dishes based on their ingredients" interaction trace.



(b) Code coverage evolution for clusters composed of tests generated from the "Set up public transport destination" interaction trace.



(c) Code coverage evolution for clusters composed of tests generated from the "Check student card balance" interaction trace.



(d) Code coverage evolution for clusters composed of tests generated from the "Search for a person" interaction trace.

Figure 7.8: The evolution of code coverage obtained by running the medoids of K clusters as a function of K.

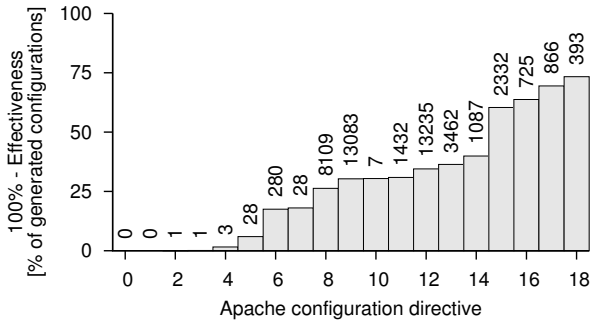
medoid, as described in Section 3.9.3.

7.4.2 Using Concolic Execution

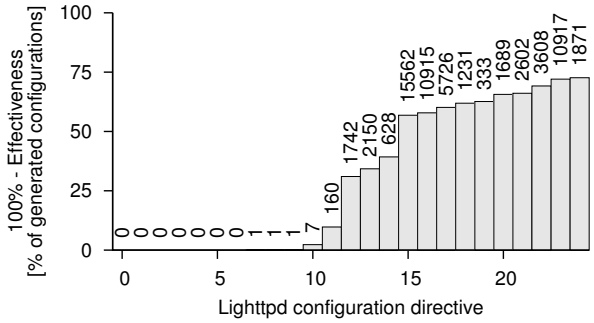
Recall that Sherpa uses concolic execution to learn what constitutes a valid configuration file. In this section, we evaluate Sherpa's efficiency. We define efficiency as the fraction of configurations that Sherpa prunes based on the inferred grammar. The larger this fraction, the more time developers save as a result of not having to run tests against invalid configuration files.

We computed Sherpa's effectiveness for Apache, Lighttpd, and IIS. Out of $144,875$ Apache configurations generated by error injection, Sherpa pruned $99,803$ (68.9%), because they did not satisfy the inferred grammar, out of $114,690$ Lighttpd configurations, it pruned $55,546$ (48.4%), and from $272,800$ IIS configurations Sherpa pruned $239,044$ (87.6%). Figure 7.9 shows the details. We show on top of the bars the number valid configurations.

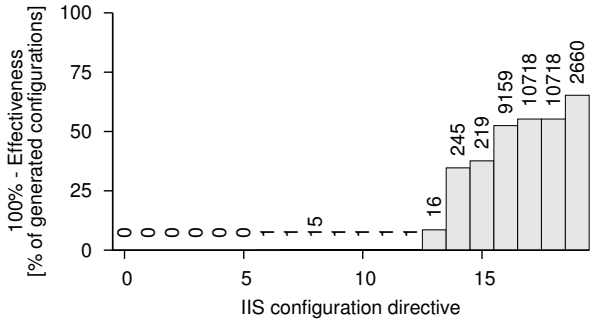
To evaluate how much time Sherpa saves developers, we run an experiment in which we check if a mutated config-



(a) Generated and pruned misconfigurations for each Apache directive.



(b) Generated and pruned misconfigurations for each Lighttpd directive.



(c) Generated and pruned misconfigurations for each IIS directive.

Figure 7.9: Generated and pruned misconfigurations for each mutable directive. These pruned configurations represent the advantage Sherpa has over ConfErr and Arugula.

uration is valid by trying to start a web server when using it. If the server starts, then we say the configuration is valid. For all three web servers, we chose a configuration directive with the same purpose, i.e., the one that specifies which port a virtual host listens to: for Apache, it is the `VirtualHost` directive, for Lighttpd, it is the `SERVER["socket"]` directive, and for IIS 7, it is the `bindingInformation` one. Table 7.8 shows the details.

The results show that Sherpa reduces the amount of time developers waste in trying out mutated configurations by an order of magnitude.

Web server	Sherpa execution time (s)	Total configuration validation time (s)	Savings
Apache	1,354	20,664	15 ×
Lighttpd	394	7,080	18 ×
IIS	405	20,897	51×

Table 7.8: Sherpa configuration file generation and pruning vs. validating all mutated configurations by trying to start the web server using them. Sherpa achieves an average saving of 22× in terms of time needed to verify configurations.

7.4.2.1 Syntactically Correct, but Pruned Configuration Files

As mentioned in Section 6.5.3, it is possible that Sherpa discards valid configurations files, because they do not respect the learned grammar. We run an experiment to assess to what degree this happens.

We use the results from the second experiment in the previous section to compute the fraction of all mutated configuration files, including the pruned ones, that were validated by the web servers. For Apache, we consider that a configuration directive was invalid if the web server did not start or if it decided to ignore the value. Table 7.9 shows the results.

Web server	Configurations validated by Sherpa	Configurations validated by the web server	Percentage of valid configurations generated by Sherpa
Apache	28	54	52%
Lighttpd	7	120 (49)	6% (14%)
IIS	219	718 (327)	31% (69%)

Table 7.9: Number of configurations validated by Sherpa vs. number of mutated configurations validated by attempting to start the web server when using them.

The results show that for Apache, Sherpa generates half the valid configurations.

For Lighttpd, Sherpa misses 94% of valid configurations. However, this number is misleading, because while the server documentation specifies that the `SERVER["socket"]` directive takes as parameter values of the form `ip:port`, with `ip` being optional and `port` being a port number [149], the server does accept `:4z3` as correct value, when it should not, since `4z3` is not a number. We investigated this anomaly and observed that Lighttpd stops processing the `port` value when it encounters a non-digit, i.e., Lighttpd listens on port 4. There is no mention of this behavior during startup, nor in the error log. We believe this to be an oversight of Lighttpd developers. Only when the `port` values starts with a non-digit character does Lighttpd report an error. If we disregard mutated configurations that do not contain as port number a valid integer, then there are 49 valid configurations, of which Sherpa discovered 14%.

For IIS, Sherpa generates a third of valid configuration files. However, this number is misleading too, because IIS does no validation of the parameter upon startup. For example, it accepts `*:44f:` as correct parameter. According to IIS documentation [150], the value between the two colons should be a port number, which `44f` is not. When investigating this anomaly, we observed that starting IIS from command line gives no indication of the error, but when using the IIS Service Manager GUI administration tool to start it, the tool complains about the invalid value. Thus, there exists a difference in how configuration file validation is performed depending on the starting method used. We believe that this difference should be removed. If we disregard mutated configurations that do not contain as port number a valid integer, then there are 327 valid configurations, of which Sherpa discovered 69%.

7.5 Relevance Measured with PathScore–Relevance

To evaluate whether $PathScore\text{-}Relevance$ can improve the efficiency of test development, it should be applied to a test development process. Given that it would be difficult to discount the human factor, we opted instead to use $PathScore\text{-}Relevance$ in the context of an automated test generation tool. We chose KLEE [124], which takes a program and, without modifying the program, can generate tests that exercise the program along different paths. KLEE's explicit goal is to maximize test coverage. KLEE relies on a symbolic execution engine and a model of the filesystem to discover ways in which the target program can be exercised. During symbolic execution, KLEE relies on a *searcher* to decide which statement it should explore next. The default searcher in KLEE employs random search along with some heuristics. When reaching an exit point, be it a normal exit or due to a bug, KLEE generates a test case that drives the application through that same execution path.

We implemented a new searcher for KLEE that, when a new choice is to be made, selects a statement inside the most relevant function, in accordance with $\mathcal{R}$. However, when the path score $\mathcal{P}$ for the respective function achieves a target goal, another statement from a function with a lower $\mathcal{R}$ is chosen, to increase that function's $\mathcal{PR}$. The process repeats either until overall $\mathcal{PR}$ reaches a set target or time runs out, thus emulating real-world testing.

We ran KLEE for 5 minutes on five COREUTILS [124] programs (*cat, sort, mkdir, tail, tr*) in three scenarios: one with KLEE's random searcher, which randomly picks the next statement to be symbolically executed, and two with the $\mathcal{PR}$-based searcher.

The two $\mathcal{PR}$ scenarios differ in how relevance is determined. In the first one, we collect usage profiles by compiling the applications with profiling support and exercising a sample workload for these tools. In the second one, we annotate some of the error recovery paths in the five programs, setting the *recovery* flag for the respective lines to 1. To ensure that error recovery paths are executed, we configured KLEE to generate a maximum of 50 system call errors along each execution path.

In Figure 7.10, we report the average number of paths explored by KLEE with the two searchers in the three scenarios. For each program, we normalized the numbers relative to the default KLEE searcher; the absolute values are shown above the bars.

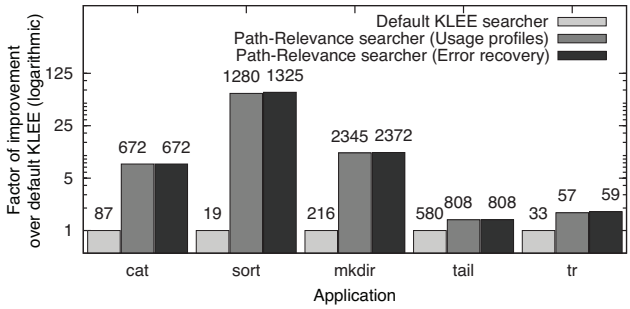


Figure 7.10: Number of explored paths resulting from 5-minute KLEE runs in 3 scenarios for 5 programs.

The $\mathcal{PR}$ searcher makes KLEE explore 1.7× to 70× more paths than the default KLEE searcher, in the same amount of time. A key reason for the speed-up is that the test generation process is steered toward exploring shorter paths before exploring the longer ones, so KLEE can find exit conditions more quickly.

Another reason is that the $\mathcal{R}$ component of the metric guides KLEE to explore more relevant code. In the case of usage profiles, KLEE was guided to exercise paths (and variants thereof) that the benchmark exercised, thus keeping KLEE from getting "trapped" in potentially infeasible parts of the execution tree—this effect is particularly visible

in the case of *sort* and *tr*, where the default KLEE searcher explored only few paths. In the case of error recovery annotations, KLEE was steered toward paths that reached exit points quickly (e.g., program exit due to error), so interesting conditions that result in test cases were found more quickly.

While these results are by no means predictive of efficiency improvements in an actual development organization, it is a data set that suggests $\mathcal{PR}$'s potential. A test suite with more tests explores more paths, so it increases the probability that it will find bugs in important parts of the code, thus improving test quality at no increase in cost.

7.6 User Anonymity

In this section we evaluate the k-anonymity ReMuTe can provide for proactive events (§7.6.1) and for reactive events (§7.6.2) and what is the cost of preventing query attacks (§7.6.3). We provide simulation results (§7.6.1) and empirical results (§7.6.2, §7.6.3).

7.6.1 Anonymity of Proactive Events

We analytically evaluate whether there exists behavior common across users. The more there is, the better ReMuTe can protect a user's anonymity. It is intuitive that as more users there are, the same behavior is more frequently encountered. Nevertheless, we wish to obtain estimates on the possible values of k that we can expect ReMuTe to achieve.

To estimate a realistic value for k, we simulate a user population of 100 million, representative of popular Android applications [151]. In this experiment, each user randomly interacts with a model of an application that contains four screens, each of which contains up to five UI widgets. Each user generates a trace that contains 15 random events, consistent with [152], split across all screens.

We evaluate the relation between population size and average trace frequency. Figure 7.11 contains the results.

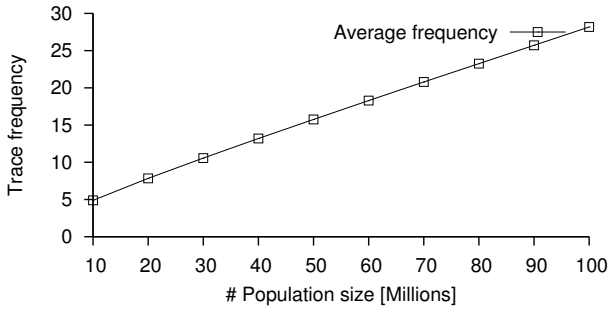


Figure 7.11: k-anonymity depending on population size.

Figure 7.11 shows that the average number of users exhibiting common behavior increases linearly with the population size. This means that there are no diminishing returns in new users joining ReMuTe, i.e., each new user improves the anonymity of other users by a constant measure.

Next, we evaluate the relation between the length of a trace and its frequency. The shorter a trace, the more common it should be. In this experiment, we wish to obtain plausible k values for traces of various lengths.

We generate 10 random traces and compute the frequency of each of their sub-traces in a population of 100 million traces. We report the minimum, maximum, and average frequency per trace length. Figure 7.12 shows the results.

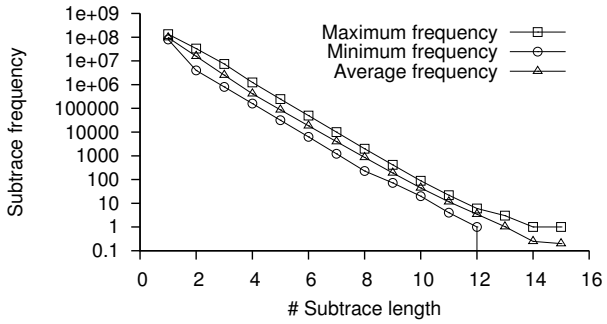


Figure 7.12: k-anonymity depending on trace length.

We find that a trace's frequency decreases exponentially with its length. Each new event in an interaction trace increases exponentially the number of traces that need be produced in order for k to remain the same. This result upholds our second observation underlying our definition of the k-disclosure metric, i.e., that the amount of personally-identifiable information (PII) in a trace is not just the sum of the PII in each event (otherwise, there should have been a linear relation).

7.6.2 Anonymity of Reactive Events

In this section, we evaluate the anonymity ReMuTe can provide for reactive events by quantifying how k-anonymous each field of a server response is after PocketCampus processes it. Intuitively, the higher the k, the less information that field contains about a user and the higher the possibility of it being recorded by other users.

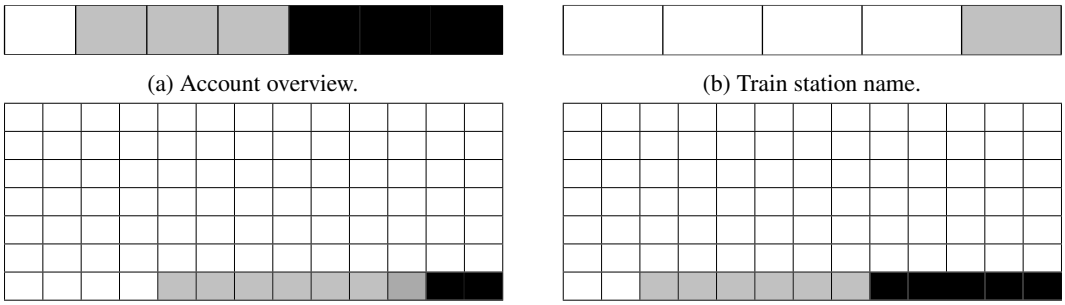
We focus on two functionalities provided by PocketCampus: check the balance of a student card account, and display the time of the next departure from a public transport station. In both cases, PocketCampus makes a request to a server, processes the response, and displays some information. Each server response contains multiple fields, and we compute how many alternatives ReMuTe can generate for each field.

Figure 7.13 contains the results, and shows that PocketCampus places few constraints on server responses, enabling ReMuTe to provide high k-anonymity for users. Figure 7.13a shows that the server's response to inquires about the account balance contains seven fields: one is not processed by PocketCampus (the white box), three for which ReMuTe generates more than 100 alternatives (the gray boxes), and three for which ReMuTe cannot generate alternatives (the black boxes), because they contain floating point values (not supported by our constraint solver) or because PocketCampus checks their value against constant values (e.g., the server's status).

Figure 7.13b shows the number of alternatives for the server's response when queried about the name of a public transport station. Figures 7.13c and 7.13d show the results for the same server response, but in different interaction traces. Figure 7.13c corresponds to PocketCampus showing the departure time, while Figure 7.13d corresponds to additionally displaying trip details, which causes PocketCampus to further process the response and place additional constraints.

We report the percentage of bits identical in the original server response fields and the ones ReMuTe generated, considering only those processed by PocketCampus. Figure 7.14 shows that the alternative responses reveal, on average, 72% of the original bits, thus being more similar to the original ones than randomly-generated ones.

A higher similarity of the generated responses to the actual response bits than to randomly generated bits is expected, because PocketCampus processes the server response, which restricts the set of values the fields may contain.



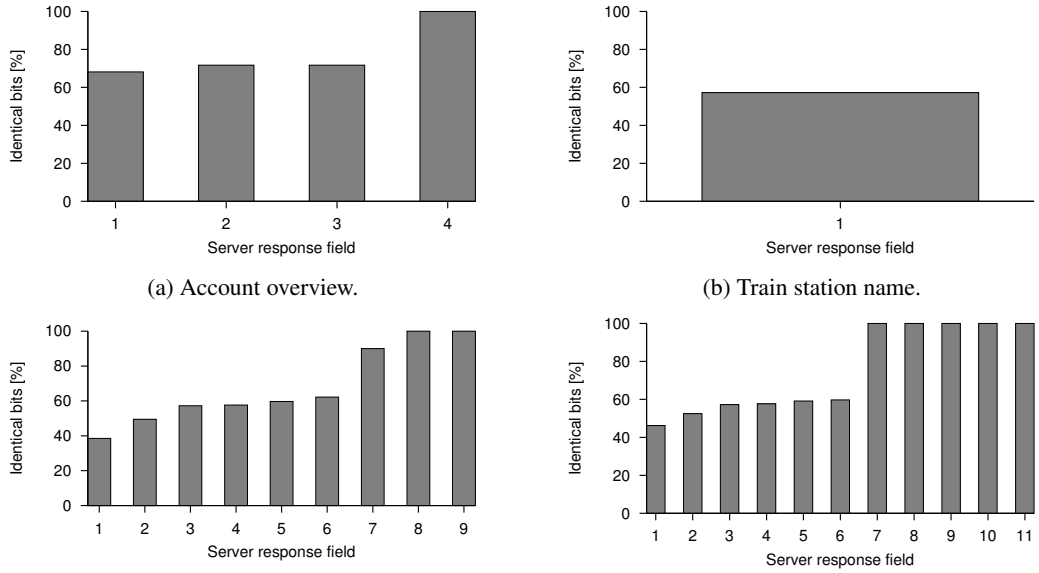
(a) Account overview.

(b) Train station name.

(c) Next departure time.

(d) Details of next departure.

Figure 7.13: k-anonymity for server response fields. White boxes show unprocessed fields, gray boxes show fields with $k \geq 100$, black boxes show fields with $k = 1$. Figures 7.13c and 7.13d depict the same server response, but at different interaction times, and show how a server response field changes its k-anonymity in response to user actions, which cause the application to constraint it further.



(a) Account overview.

(b) Train station name.

(c) Next departure time.

(d) Details of next departure.

Figure 7.14: Percentage of bits identical in the original server responses and the alternatives ReMuTe generated.

The two experiments above show that PocketCampus does not do extensive processing of server responses, a pattern we believe is common to information-retrieval mobile applications, such as Facebook or Twitter. This result is encouraging, because it shows that users of such applications can contribute interaction traces to developers with high levels of anonymity.

7.6.3 Performance Overhead of Preventing Query Attacks

We evaluate the time required by a program instance to check if the answer it provides to a query received from the ReMuTe hive poses an anonymity threat to the user.

More specifically, we are interested in how the time to answer evolves over time as more requests are received.

Given that we envision ReMuTe to be used by hundreds of millions of users, we foresee each user having to answer a large number of requests.

Figure 7.15 shows that the time to answer increases quadratically. The spikes are due to the ReMuTeDroid prototype not being able to submit the answer on the first try.

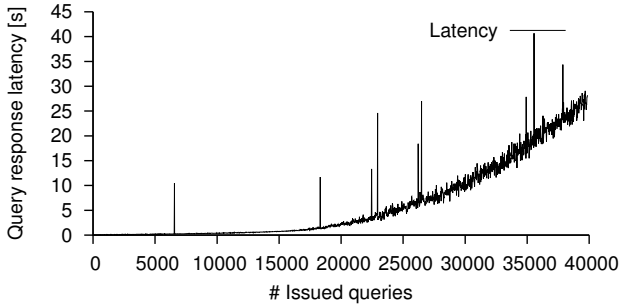


Figure 7.15: Impact of maintaining a history of received queries on the time to respond to a query.

Mobile devices are power constrained, specifically in processing time. Thus, we need to curb the time to answer. One way to achieve this is to monitor response times and clear the history of recorded interaction traces once the latency becomes too high, but maintain the sets of positively and negatively replied queries, so as to prevent becoming a victim of the threat described in §4.5.3.1. That is, when the delay becomes too high, the program instance pretends that it has attacked itself and maintains only the information it got out of the attack.

7.7 Tools Evaluation

In this section, we evaluate the impact that using programs developed based on Record-Mutate-Test may have on end users' experience. We focus on how recording interaction traces affects the responsiveness of an application. We quantify its impact as the runtime overhead expressed in milliseconds. We consider the impact to be acceptable if the runtime overhead is below the 100 ms human perception threshold [153]. We also investigate the storage space requirements for interaction traces.

7.7.1 WebErr

7.7.1.1 Runtime Overhead Introduced by Recording Interaction Traces

In order to be practical, WebErr must not hinder a user's interaction with web applications. We run an experiment, consisting of writing an email in GMail, to compute the time required by the WebErr Recorder to log each user action. The average required time is on the order of hundreds of microseconds and does not hinder user experience. Specifically, the minimum processing time required by the WebErr Recorder is $24\mu s$, the maximum time is $897\mu s$, and the average is $107\mu s$. Thus, the WebErr Recorder is lightweight.

7.7.1.2 Storage Size Requirements for Interaction Traces

A small interaction trace means that users need not expend large network communication traffic to share their trace with a web application's developers. We report storage requirements for six WebErr interaction traces. We show

the number of events, the raw storage space they claim, and the compressed size of the interaction trace. Table 7.10 contains the results.

Interaction trace description	Number of events	Interaction trace size (KB)	Compressed size (KB)
Edit Google Drive spreadsheet	48	2.8	0.7
Edit Google Sites website	46	4.1	0.7
Send email using GMail	81	10.7	1
Search on Google	24	4.5	0.6
Play Paparazzi on Google Guitar	98	4.3	0.4
Drag and drop square	652	26.6	2.6

Table 7.10: The storage space required by WebErr interaction traces.

The results show that a WebErr command requires, on average, 57 bytes of storage. Furthermore, WebErr interaction traces have a compressibility ratio of 8.8 to 1. We conclude that WebErr does not impose high storage requirements on users.

7.7.1.3 Interaction Trace Completeness

We compare the recording fidelity of the WebErr Recorder to that of Selenium IDE. For our experiment, we focus on four widely used web applications: Google Sites, GMail, the Yahoo! web portal, and Google Drive. We choose these applications because they are representative of modern web applications. Results are presented in Table 7.11 and show that the WebErr Recorder offers higher fidelity than Selenium IDE.

Application	Scenario	WebErr Recorder	Selenium IDE
Google Sites	Edit site	C	P
GMail	Compose email	C	P
Yahoo	Authenticate	C	C
Google Drive	Edit spreadsheet	C	P

Table 7.11: The completeness of recording user actions using the WebErr Recorder and Selenium IDE. In this table, C stands for complete, and P stands for partial.

7.7.2 ReMuTeDroid

7.7.2.1 Runtime Overhead Introduced by Recording Interaction Traces

To evaluate the runtime overhead introduced by the ReMuTeDroid record and replay infrastructure, we logged the time it takes for specific UI events to execute when there's no recording, during recording, and during replay. As previously described, ReMuTeDroid uses recording during replay to ensure the order of events is preserved.

Table 7.12 contains the results, expressed in milliseconds. The second column shows the number of events replayed. The third column contains average vanilla values, i.e., without recording nor replaying. The fourth and fifth column present average recording and replay overheads, respectively, expressed as additional time spent. The results were obtained using an Android emulator running Android 2.3

The table shows that ReMuTeDroid incurs less than 10 ms of runtime overhead during recording, thus not significantly hindering the application's responsiveness.

Event	Sample size	Vanilla	Record	Replay
Click	160	17.5	7.5	11.0
Scroll	624	0.03	2.1	2.1
Type	160	0.02	9.2	9.2
Touch	120	0.26	5.7	13.3

Table 7.12: Overhead time to record and replay, in ms, of various types of user interaction events using ReMuTeDroid.

7.7.2.2 Storage Size Requirements for Interaction Traces

We present a breakdown of the storage size requirements for ReMuTeDroid's interaction traces in Table 7.13. The table contains the details for seven PocketCampus interaction traces from Table 7.5. We report the storage size for all events except for network communication, for recorded network communication, total trace size, and archived trace size, in KB.

Trace id	# Events	Events (KB)	Network communication (KB)	Total (KB)	Compressed (KB)
1	17	2.1	2.1	4.3	5.4
2	81	11.5	31.1	42.6	20.2
3	22	5.0	16.9	21.9	8.5
4	29	7.0	14.8	21.8	8.2
5	27	5.3	4.5	9.9	7.7
6	46	9.5	5.3	14.8	10.1
7	48	6.5	25.0	31.5	17.6

Table 7.13: The PocketCampus interaction traces contain 17 to 81 events, with an average of 38 events. Most storage space is claimed by server communication. The average archiving ratio is 1.6:1.

We conclude that the storage space required by ReMuTeDroid interaction traces is negligible, especially given that today's smartphones ship with upwards of 16GB of persistent memory.

7.7.2.3 Interaction Trace Completeness

We showcase the completeness of the interaction traces recorded by ReMuTeDroid by describing a bug we discovered in PocketCampus and how the *Replay* pod consistently reproduced it, which enabled us to figure out what caused it.

PocketCampus offers users information about the Lausanne-area public transport schedule. Users can set metro, bus, or train stations as preferred destinations. We stumbled upon a bug in PocketCampus that prevents users from adding a destination to the set of preferred destinations. The bug manifests when there are no routes between EPFL and the new destination. In this case, the destination is not added to persistent storage.

We accidentally triggered the bug by having a misconfiguration in the time zone of our smartphone that caused all routes to appear as starting earlier than the current time of the phone. This caused the application to reject these routes.

We submitted the bug, using the ReMuTeDroid-enabled PocketCampus's "Contribute trace" feature, and reproduced it, even with correct time zone settings. This shows the benefit of having an always-on recording feature and highlights the effectiveness of ReMuTeDroid's record and replay mechanism.

7.7.2.4 Replay Portability

We wish to evaluate if the traces recorded by ReMuTeDroid are portable across different versions of the Android OS. For this, replayed three of the user interaction traces from Table 7.13 on three different emulators: one emulator running Android 2.3.3, another one running Android 4.0.3, and the last one running Android 4.1. Together, these versions cover more than 80% of the existing Android phones [154]. Each trace was replayed 20 times. Initially, the interaction traces were recorded on a real Android device running Android 2.3.

Table 7.14 contains the results, and they show that all the traces recorded with ReMuTeDroid were replayed on all the different Android versions.

Trace	**Android 2.3**	**Android 4.0**	**Android 4.1**
PocketCampus - Browse the day's menus and vote for a dish	✓	✓	✓
PocketCampus - Search for a person	✓	✓	✓
PocketCampus - Check student card balance	✓	✓	✓

Table 7.14: ReMuTeDroid trace portability.

7.7.2.5 Computing k-Proactive-Anonymity

We evaluate the time required by a program instance to compute how k-proactive-anonymous is a trace. We report the time required for an instance to compute the k value as a function of 1) the number of positive replies received to a query, K, 2) the formula used, and 3) the subject of the formula.

In our experiment, we use ReMuTeDroid to replay recorded interaction traces. We compute the value of k in two settings: 1) for a single event, and 2) for all the new sub-traces of the trace replayed so far, value necessary to compute the value of the k-disclosure metric. Figure 7.16 shows the results.

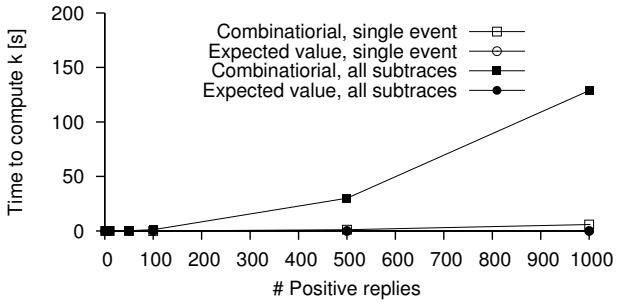


Figure 7.16: Time to compute k depending on formula used and subject, i.e., single event or all new sub-traces.

The results show that using the combinatorial formula imposes too large a runtime overhead, which cannot be neglected in the case of power-constrained mobile devices. As a solution, we envision to use the simple, expected value formula for such devices.

7.7.2.6 Concolic Execution Time

We evaluate the speedup in the concolic execution completion time brought by using the static analysis described in Section 4.4. Figure 7.17 shows that, when using the analysis' results, PocketCampus finishes processing a server

response within 10 minutes, as opposed to more than one hour when the analysis is not used.

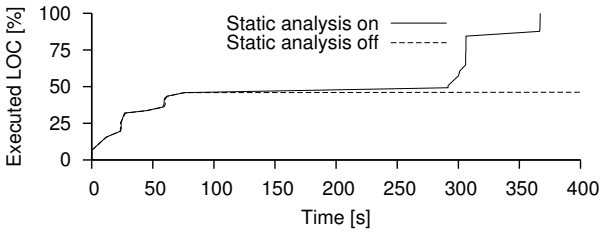


Figure 7.17: Concolic execution speedup obtained by using the static analysis described in Section 4.4. The figure can be read as a comparison, in terms of efficiency, between our algorithm and the one described in [66].

7.7.2.7 ReMuTeDroid Hive Scalability

We evaluate the scalability of the ReMuTeDroid hive as the average time a program instance waits for a response to a query. In this experiment, we vary the number of instances that issue queries and of those that answer queries.

In our experiment, instances either issue or answer queries. Each instance hat answers queries immediately replies positively, without doing any checks against its local history. Figure 7.18 shows the results.

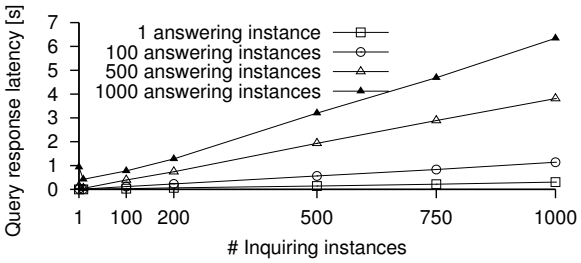


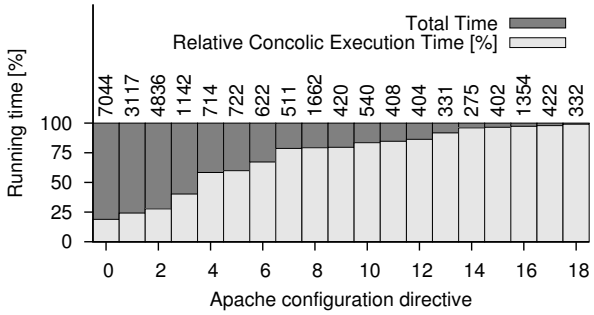
Figure 7.18: Scalability of the ReMuTeDroid hive depending on the number of instances that issue concurrent queries and of those that reply to queries.

The results show that the ReMuTeDroid hive scales linearly in both dimensions, meaning that its performance degrades gracefully as more users join the crowd.

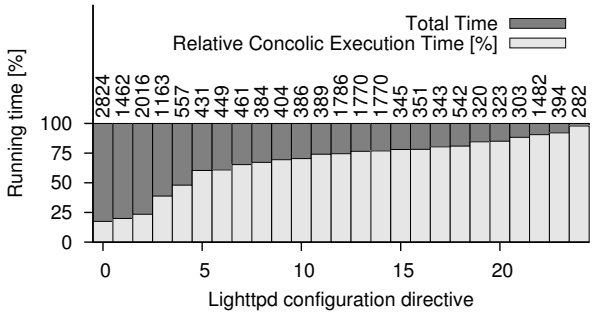
7.7.3 Sherpa

In this section, we evaluate the time needed by Sherpa to generate and prune configurations. We measured the time necessary for Sherpa to run Apache, Lighttpd, and IIS concolically and to generate and prune configurations on an 8-core machine with 20 GB of RAM. The average time is 1,160 seconds per configuration directive.

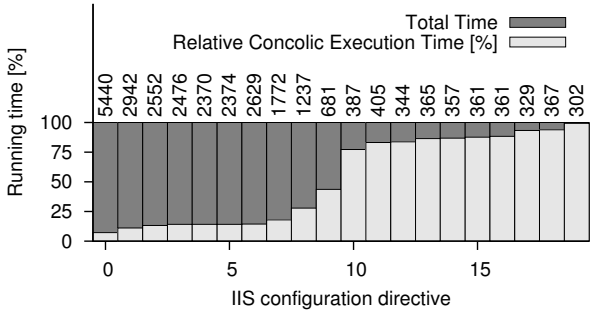
We also present a breakdown of the Sherpa execution time into concolic execution time and human error injection and pruning time. Figure 7.19 shows the results. The figure shows that in most cases, concolic execution takes more than half of the total time. Figure 7.19 shows on top of the bars the total running time.



(a) Breakdown of time to generate and prune Apache configurations.



(b) Breakdown of time to generate and prune Lighttpd configurations.



(c) Breakdown of time to generate and prune IIS configurations.

Figure 7.19: Breakdown of time required by Sherpa to generate mutated configurations.

7.8 Evaluation Summary

In this chapter, we evaluated Record-Mutate-Test and the tools we built using it. The results show that Record-Mutate-Test is effective at finding bugs in widely used applications and servers and is efficient in reducing the number of tests one needs to run. The evaluation also revealed that using real-world interaction traces can be used to guide the testing effort. Finally, its user anonymity protection schemes manage to protect users' identities while maintaining a trace's usefulness for testing.

The tools built using Record-Mutate-Test and targeting end users impose a reduced runtime overhead, thus not hindering user experience, and generate interaction traces with reduced storage space requirements.

Chapter 8

Conclusion

This thesis describes Record-Mutate-Test, a technique to test the resilience of software systems to realistic user errors. We define user errors as the commands a user issues to a software system that cause the system to perform an action with unwanted consequences that run contrary to the goal the user is pursuing. This thesis takes a pragmatic approach and focuses on how errors manifest, not on their psychological cause.

We say that a system is resilient to user errors if they cannot hinder the system's dependability [75]. Dependability is that property of a system which allows reliance to be justifiable placed on the service it delivers. Dependability has four properties: availability, reliability, safety, and security [1]. For example, a user of an e-banking application should be able to transfer money (availability), and tapping the wrong button in the application should not cause the application to crash (reliability), transfer all the user's funds away (safety), or make the user's account publicly accessible (security).

Record-Mutate-Test has three steps. First, it *records* the interaction between a user and a system into an interaction trace. An interaction trace is the sequence of events that determines a system's execution, and replaying it generates an execution that mimics the one that lead to the trace's creation.

Next, the technique *mutates* the interaction trace by injecting errors into it and generating new interaction traces. The injected errors correspond to the members of the user error taxonomy based on error manifestation described in Section 2.1.3. They affect either an event or the events' order in an interaction trace.

The newly generated interaction traces are effectively tests and correspond to users making mistakes. Thus, in its final phase, the technique *tests* the system against realistic user errors by replaying the new interaction traces.

Record-Mutate-Test possesses the following properties:

- *Accuracy*. Record-Mutate-Test is:
  - *Effective*: it finds bugs in widely used web applications, web servers, and mobile device applications, and the tests it automatically generates achieve higher code coverage than the interaction trace that served as seed.
  - *Efficient*: it reduces the number of tests needed to achieve high effectiveness. Record-Mutate-Test achieves these savings by clustering tests or by using dynamic program analysis techniques.
  - *Realistic*: it injects erroneous user behavior that is representative of the behavior actual users exert when interacting with a system.

  - *Extensible*: one can define new user error operators and easily incorporate them in the tools built upon Record-Mutate-Test.
  - *Relevant*: $PathScore\text{-}Relevance$ helps developers make best use of their limited testing time, by requiring them to focus on important parts of the system and ensuring they are thoroughly tested.

- *Automation*. Employing Record-Mutate-Test requires minimal developer effort. Developers are able to automatically generate and run tests.

- *Scalability*. Record-Mutate-Test is able to test software systems with large bodies of code and is able to express user errors independently from the system it is testing. We applied Record-Mutate-Test to web servers, web applications, and Android smartphone applications.

- *Protect users' anonymity*. ReMuTe protects users anonymity by expunging personally identifiable information from interaction traces and ensures that user behavior described by the trace is not unique.

- *Collaborative*. Record-Mutate-Test enables users to take part in testing software systems. Users can contribute interaction traces that serve as seeds for tests or they can donate their machine for test execution.

We applied Record-Mutate-Test to web applications, smartphone applications, and web servers. The tests the technique generates uncovered bugs in each category of systems and show that user errors cause new code to execute, which can contain bugs. The technique manages to preserve the anonymity of contributing users and collecting usage scenarios does not hinder user experience.

The main conclusion to be drawn from this thesis is that there exist systems we use everyday and on which we depend to manage aspects of our lives that are not yet resilient to user errors. This is worrisome. Fortunately, the technique and tools described in this thesis can help developers be aware of what can possibly go wrong so that they can fix the issues.

Bibliography

[1] I. Sommerville, *Software engineering*, 6th ed. Addison–Wesley, 2001.

[2] S. D. Wood and D. E. Kieras, "Modeling human error for experimentation, training, and error-tolerant design," in *The Interservice/Industry Training, Simulation & Education Conference*, 2002.

[3] L. Fastnacht, "EPFL wrong email list," 2013, private communication.

[4] S. Casey, *Set Phasers on Stun: And Other True Tales of Design, Technology, and Human Error*. Aegean Pub Co, 1998.

[5] "Facebook outage," http://on.fb.me/bE0J5K, September 2010.

[6] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer*, July 1993.

[7] "2010 Annual Report of the Interception of Communications Commissioner," http://www.official-documents.gov.uk/document/hc1012/hc12/1239/1239.pdf.

[8] A. B. Brown, *Queue*, vol. 2, no. 8, Nov. 2004.

[9] D. Diaper and N. Stanton, *The Handbook of Task Analysis for Human-Computer Interaction*. Lawrence Erlbaum Associates, 2004.

[10] E. Hollnagel, "The phenotype of erroneous actions," in *International Journal of Man-Machine Studies*, 1993.

[11] D. A. Norman, *The Design of Everyday Things*. Basic Books, 2002.

[12] J. Reason, *Human Error*. Cambridge University Press, 1990.

[13] A. B. Brown, "A recovery-oriented approach to dependable services: Repairing past errors with system-wide undo," Ph.D. dissertation, U.C. Berkeley, Dec. 2003.

[14] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A tool for assessing resilience to human configuration errors," in *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2008.

[15] A. Swain and H. Guttman, "Handbook of human reliability analysis with emphasis on nuclear power plant applications," Nuclear Regulatory Commission, Tech. Rep. NUREG CR-1278, 1983.

[16] "Testing at the speed and scale of google," http://googletesting.blogspot.ch/2011/06/testing-at-speed-and-scale-of-google.html.

[17] L. Bainbridge, "Ironies of automation," in *Automatica*, 1983.

[18] World Nuclear Association, "Three mile island accident," http://www.world-nuclear.org/info/inf36.html.

[19] R. A. Maxion and R. W. Reeder, "Improving user-interface dependability through mitigation of human error," *Int. J. Hum.-Comput. Stud.*, vol. 63, no. 1-2, 2005.

[20] E. Kycyman and Y.-M. Wang, "Discovering correctness constraints for self-management of system configuration," 2004.

[21] C. R. P. dos Santos, L. Z. Granville, L. Shwartz, N. Anerousis, and D. Loewenstern, "Quality improvement and quantitative modeling using mashups for human error prevention," in *Integrated Network Management*, 2013.

[22] J. E. A. Jr, R. Conway, and F. B. Schneider, "User recovery and reversal in interactive systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, no. 1, pp. 1–19, 1984.

[23] A. B. Brown and D. A. Patterson, "Undo for operators: Building an undoable e-mail store," in *USENIX Annual Technical Conf. (USENIX)*, 2003.

[24] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Symp. on Operating Sys. Design and Implementation (OSDI)*, 1999.

[25] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in Internet services," in *Symp. on Operating Sys. Design and Implementation (OSDI)*, 2004.

[26] "junit," http://junit.org.

[27] "nunit," http://www.nunit.org.

[28] "Robotium," http://code.google.com/p/robotium/.

[29] "Testdroid," http://testdroid.com.

[30] G. Logic, "Monkeytalk," https://www.gorillalogic.com/monkeytalk.

[31] "Selenium," http://seleniumhq.org/.

[32] "Jenkins," http://jenkins-ci.org.

[33] S. Alisauskas, "Community based testing," 2013.

[34] "Xamarin test cloud," http://xamarin.com/test-cloud.

[35] "Perfecto Mobile," http://www.perfectomobile.com.

[36] "Cloud Monkey Appliance," http://www.gorillalogic.com/cloudmonkey.

[37] A. Brown, L. C. Chung, and D. A. Patterson, "Including the human factor in dependability benchmarks," in *Proc. DSN Workshop on Dependability Benchmarking*, 2002.

[38] A. Whitten and J. D. Tygar, "Why johnny cant encrypt: A usability evaluation of pgp 5.0," 1999.

[39] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and validating database system administration," in *USENIX Annual Technical Conf. (USENIX)*, 2006.

[40] M. Vieira and H. Madeira, "Recovery and performance balance of a COTS DBMS in the presence of operator faults," in *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2002.

[41] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.

[42] "Fiddler," http://www.fiddler2.com/fiddler2/.

[43] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic capture and replay for JavaScript applications," 2010.

[44] R. Atterer, M. Wnuk, and A. Schmidt, "Knowing the user's every move - user activity tracking for website usability evaluation and implicit interaction," in *Intl. World Wide Web Conference*, 2006.

[45] U. Kukreja, W. Stevenson, and F. Ritter, "RUI: Recording user input from interfaces under Windows and Mac OS X," in *Behavior Research Methods*, 2006.

[46] J. Alexander, A. Cockburn, and R. Lobb, "AppMonitor: A tool for recording user actions in unmodified windows applications," in *Behavior Research Methods*, 2008.

[47] H. Okada and T. Asahi, "Guitester: A log- based usability testing tool for graphical user interfaces," in *EICE Transactions on Information and Systems*, 1999.

[48] G. Al-Qaimari and D. McRostie, "Kaldi: A computer-aided usability engineering tool for supporting testing and analysis of human-computer interaction," in *Computer-Aided Design of User Interfaces II*, 1999.

[49] D. Uehling and K. Wolf, "User action graphing effort (UsAGE)," in *CHI 95: Conference on Human Factors in Computing Systems*, 1995.

[50] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Symp. on Operating Sys. Design and Implementation (OSDI)*, 2002.

[51] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman, "Execution replay on multiprocessor virtual machines," 2008.

[52] "uTest," http://www.utest.com.

[53] "AppLover," http://applover.me.

[54] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults - based on field data," in *Annual Symposium on Fault Tolerant Computing*, 1996.

[55] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions of Software Engineering*, vol. 32, no. 11, 2006.

[56] A. J. Offutt, "A practical system for mutation testing: Help for the common programmer," in *IEEE International Test Conference on TEST: The Next 25 Years*, 1994.

[57] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *EEE Transactions on Software Engineering*, vol. 37, no. 5, 2011.

[58] J. Allen Troy Acree, "On mutation," Ph.D. dissertation, Georgia Institute of Technology, 1980.

[59] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," 1980.

[60] S. Hussain, "Mutation clustering," 2008.

[61] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Intl. Conf. on Software Engineering (ICSE)*, 1995.

[62] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *International Working Conf. Source Code Analysis and Manipulation*, 2008.

[63] J. A. Whittaker and M. G. Thomason, "A markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, 1994.

[64] S. Cornett, "Code coverage analysis," http://www.bullseye.com/coverage.html, Dec 2008.

[65] L. Sweeney, "K-Anonymity: A model for protecting privacy," in *Intl. Journal on Uncertainty, Fuzziness and Knowledge-based Systems (IJUFKS)*, 2002.

[66] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[67] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Intl. Conf. on Software Engineering (ICSE)*, 2011.

[68] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *ACM SIGMOD Conf. (SIGMOD)*, 2000.

[69] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," 2009.

[70] "onmouseover Event," http://www.w3schools.com/jsref/event_onmouseover.asp.

[71] "ICT facts and figures 2013," http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013.pdf.

[72] "Google I/O," http://googleblog.blogspot.ch/2012/06/chrome-apps-google-io-your-web.html.

[73] "Number of active users at Facebook over the years," http://news.yahoo.com/number-active-users-facebook-over-230449748.html

[74] "The NIST definition of cloud computing," http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[75] "Dependability," https://wiki.ittc.ku.edu/resilinets_wiki/index.php/Definitions#Dependability.

[76] "Hierarchical task analysis example," http://www.uxmatters.com/mt/archives/2010/02/hierarchical-task-analysis.php.

[77] J. Hobart, "Principals of good GUI design," http://www.alberscvhs.com/6_visualBasic/PrincipalsGUI-Design.pdf.

[78] "iOS Storyboard," https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.h

[79] "51 amazing Facebook stats," http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/.

[80] "Google Chrome has 310 million active users," http://www.theverge.com/2012/6/28/3123639/google-chrome.

[81] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," vol. 38, no. 1, Jun. 2010.

[82] Z. Yang, M. Yang, L. Xu, and H. Chen, "Order: Object centric deterministic replay for Java," in *USENIX Annual Technical Conf. (USENIX)*, 2011.

[83] "AspectJ," http://www.eclipse.org/aspectj.

[84] "Wikipedia:lists of common misspellings/for machines," http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines.

[85] K. Sen, "Concolic testing," in *ACM Intl. Conf. on Automated Software Engineering (ASE)*, 2007.

[86] B. Everitt, *Cluster Analysis*. E. Arnold, 1993.

[87] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, 1964.

[88] B. Krishnamurthy and C. E. Wills, "On the leakage of personally identifiable information via online social networks," in *Workshop on Online Social Networks (WOSN)*, 2009.

[89] P. Samarati and L. Sweeney, "Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression," SRI International, Tech. Rep., 1998.

[90] C. Zamfir, G. Altekar, G. Candea, and I. Stoica, "Debug determinism: The sweet spot for replay-based debugging," in *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2011.

[91] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated debugging," in *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2010.

[92] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[93] A. Shamir, "How to share a secret," 1979.

[94] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison–Wesley, 1975 (revised 1995).

[95] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage-based testing tools," in *Proceedings of the 2006 international workshop on Automation of software test*, 2006.

[96] Y. W. Kim, "Efficient use of code coverage in large-scale software development," in *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '03, 2003.

[97] P. J. Guo and D. Engler, "Linux kernel developer responses to static analysis bug reports," in *USENIX Annual Technical Conf. (USENIX)*, 2009.

[98] A. H. Watson and T. J. McCabe, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology, 1996.

[99] "Halstead complexity metrics," http://www.verifysoft.com/en_halstead_metrics.html.

[100] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Using developer information as a factor for fault prediction," in *Intl. Conf. on Software Engineering (ICSE)*, 2007.

[101] "Top ranking applications (Wakoopa)," http://www.favbrowser.com/top-ranking-applications-wakoopa/, 2009.

[102] "Google Drive," https://drive.google.com/#my-drive.

[103] "Selenium IDE," http://seleniumhq.org/projects/ide.

[104] "WebKit," http://www.webkit.org/.

[105] "XML path language (XPath)," http://w3.org/TR/xpath.

[106] S. K. Card, T. P. Moran, and A. Newell, *The psychology of human-computer interaction*. Lawrence Erlbaum Associates, 1983.

[107] "Google Sites," http://sites.google.com.

[108] "AJAX," http://www.w3schools.com/ajax/.

[109] "SPDY," http://www.chromium.org/spdy.

[110] "Watir," http://watir.com.

[111] "Chrome's rendering architecture," http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome.

[112] "WebDriver," http://google-opensource.blogspot.com/2009/05/introducing-webdriver.html.

[113] "ChromeDriver," http://code.google.com/p/selenium/wiki/ChromeDriver.

[114] "1.5 million Android devices are activated daily," http://www.androidcentral.com/larry-page-15-million-android-devices-activate

[115] "102b AppStore downloads globally in 2013," http://techcrunch.com/2013/09/19/gartner-102b-app-store-downloads-globally-in-

[116] C. Hu and I. Neamtiu, "Automating gui testing for android applications," 2011.

[117] "Android tops 81 percent of smartphone market share in Q3," http://www.engadget.com/2013/10/31/strategy-analytics-q3-2013-p

[118] "1 billion Android devices," https://plus.google.com/u/0/+SundarPichai/posts/NeBW7AjT1QM.

[119] "PocketCampus," http://www.pocketcampus.org.

[120] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotk, O. Lhotk, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "abc: An extensible AspectJ compiler," in *Aspect-Oriented Software Development Conference*, 2005.

[121] "Bugzilla," http://www.bugzilla.org/.

[122] A. Kirchner, "Data leak detection in android applications," 2011.

[123] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conf. (USENIX)*, 2005.

[124] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symp. on Operating Sys. Design and Implementation (OSDI)*, 2008.

[125] "Protocol buffers," http://code.google.com/p/protobuf/.

[126] "Apache thrift," http://thrift.apache.org.

[127] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do Internet services fail, and what can be done about it?" 2003.

[128] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Networked Windows NT system field failure data analysis," in *Proc. Pacific Rim Intl. Symp. on Dependable Computing*, 1999.

[129] S. Pertet and P. Narasimhan, "Causes of failures in web applications," Carnegie Mellon University, Tech. Rep. CMU-PDL-05-109, 2005.

[130] "PostgreSQL documentation," http://www.postgresql.org/docs/8.3/static/runtime-config-resource.html#RUNTIME-CONFIG-RE

[131] "Apache httpd," http://httpd.apache.org, 2013.

[132] "Lighttpd," http://www.lighttpd.net.

[133] "Microsoft IIS 7," http://www.iis.net.

[134] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," 2009.

[135] "Secrets for android," http://code.google.com/p/secrets-for-android/.

[136] "Addi," http://code.google.com/p/addi/.

[137] "AnkiDroid," http://code.google.com/p/ankidroid/.

[138] "Gmail to use more HTML5 features," http://googlesystem.blogspot.ch/2010/06/gmail-to-use-more-html5-features.html.

[139] "(S)LOC count evolution for selected OSS projects," http://www.consecom.com/publications/papers/tik_report_315.pdf.

[140] "New York Times Most Searched," http://www.nytimes.com/most-popular-searched.

[141] "Google Trends Hot Searches," http://www.google.com/trends/hottrends.

[142] "Myth #24: People always use your product the way you imagined they would," http://uxmyths.com/post/1048425031/myth-24-people-always-use-your-product-the-way-you-imagi.

[143] Y.-M. Wang, D. Beck, J. Wang, C. Verbowski, and B. Daniels, "Strider typo-patrol: Discovery and analysis of systematic typo-squatting," in *Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet*, 2006.

[144] "HTTP/1.1 Code Definitions," http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[145] "w3af," http://w3af.org.

[146] "skipfish," http://code.google.com/p/skipfish/.

[147] "ApacheBench," http://httpd.apache.org/docs/2.0/programs/ab.html.

[148] MITRE, "CVE-2005-3357," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3357, 2005.

[149] "Lighttpd configuration documentation," http://redmine.lighttpd.net/projects/1/wiki/Docs_Configuration.

[150] "IIS 7 configuration documentation," http://www.iis.net/configreference/system.applicationhost/sites/site/bindings/binding.

[151] "Facebook Android app," https://play.google.com/store/apps/details?id=com.facebook.katana.

[152] U. I. Engineering, "Testing the three-click rule," http://www.uie.com/articles/three_click_rule/.

[153] J. V. Forrester, *The eye: basic sciences in practice*. Elsevier Health Sciences, 2002.

[154] Google, "Platform versions," http://developer.android.com/about/dashboards/.

Silviu ANDRICA

Ècole Polytechnique Fèdèrale de Lausanne
School of Computer and Communication Sciences Dependable Systems Laboratory
Building INN Room 329
Station 14 CH-1015 Lausanne

Phone: (+41)-21-6938188
Email: silviu.andrica@epfl.ch
Homepage: http://people.epfl.ch/silviu.andrica

Research Objective

Evaluate the impact human errors have on software systems. To achieve this, I use models of how humans err and apply them to a variety of user interactions with various systems, ranging from configuring a system to interacting with highly-dynamic web applications and smartphone applications.

Education

Ph.D candidate at Ècole Polytechnique Fèdèrale de Lausanne, September 2008-present

Thesis: Testing Software Systems Against Realistic User Errors

Advisor: George Candea

Dipl.Eng. in Computer Science from "Politehnica" University of Timisoara, 2003-2008

Work Experience

Research Assistant in Dependable Systems Laboratory, 2008-present

University: Ècole Polytechnique Fèdèrale de Lausanne

Advisor: George Candea

Research Intern in Dependable Systems Laboratory, April 2008-September 2008

University: Ècole Polytechnique Fèdèrale de Lausanne

Advisor: George Candea

Teaching Experience

Teaching Assistant for Software Engineering, Fall Semester 2009-2013

University: Ècole Polytechnique Fèdèrale de Lausanne

Teacher: George Candea

Teaching Assistant for Software Development Project, Fall Semester 2010-2012

University: Ècole Polytechnique Fèdèrale de Lausanne

Teacher: George Candea

Proceedings

Mitigating Anonymity Concerns in Self-testing and Self-debugging Programs

Authors: Silviu Andrica and George Candea

Venue: International Conference on Autonomic Computing (ICAC), San Jose, CA, June 2013

Efficiency Optimizations for Implementations of Deadlock Immunity

Authors: Horatiu Jula, Silviu Andrica, and George Candea

Venue: International Conference on Runtime Verification (RV), San Francisco, CA, September 2011

WaRR: High-Fidelity Web Application Recording and Replaying

Authors: Silviu Andrica and George Candea

Venue: International Conference on Dependable Systems and Networks (DSN), Hong-Kong, China, 2011

iProve: A Scalable Approach to Consumer-Verifiable Software Guarantees

Authors: Silviu Andrica, Horatiu Jula, George Candea

Venue: International Conference on Dependable Systems and Networks (DSN), Chicago, USA, 2010

PathScore-Relevance: A Metric for Improving Test Quality

Authors: Silviu Andrica, George Candea

Venue: Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep), Lisbon, Portugal, 2009

Honors

Prime Speciale, 2010 Ècole Polytechnique Fèdèrale de Lausanne

Prime Speciale, 2011 Ècole Polytechnique Fèdèrale de Lausanne

Languages

English, fluent

French, fair

Italian, fair

German, limited

Romanian, mother tongue