<table>
<thead>
<tr>
<th>Project number:</th>
<th>258109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Acronym:</td>
<td>FastFix</td>
</tr>
<tr>
<td>Project Title:</td>
<td>Monitoring Control for Remote Software Maintenance</td>
</tr>
<tr>
<td>Call identifier:</td>
<td>FP7-ICT-2009-5</td>
</tr>
<tr>
<td>Instrument:</td>
<td>STREP</td>
</tr>
<tr>
<td>Thematic Priority:</td>
<td>Internet of Services, Software and Virtualization</td>
</tr>
<tr>
<td>Start date of the project:</td>
<td>June 1\textsuperscript{st}, 2010</td>
</tr>
<tr>
<td>Duration:</td>
<td>33 months</td>
</tr>
</tbody>
</table>

D5.4: 2\textsuperscript{nd} prototype of the execution recorder/replayer tool

Lead contractor: INESC ID

Author(s): João Garcia, Paolo Romano, Luís Rodrigues, João Matos, João Nuno Silva, João Barreto, Tobias Röhm

Submission date: December 2012

Dissemination level: Public

Project co-founded by the European Commission under the Seventh Framework Programme

© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This document describes the design and usage of the 2nd iteration of the execution recorder/replayer components of the FastFix project’s platform, a.k.a. fault replication. It aims to present improvements since the version described in deliverable D5.3. These components provide an enhanced, automated and anonymized error reporting mechanism by recording application execution and automatic generation of error reports. This mechanism allows developers to reliably reproduce field failures in their own environments. This is a companion document for the code submitted.
1 Introduction

A major motivation for the FastFix project is the realization that software maintenance teams, when debugging deployed applications, would often be aided by additional information besides the point of failure and core dump information that is traditionally available in error reports. For software maintenance engineers it is often difficult to replicate errors based on core dumps and the contents of user-submitted error reports.

The recorder/replayer tool of FastFix plays a central role in supporting FastFix applications. Its main components are the recorder tool (on the client device), the error report generation (on both client and server) and the error replayer (on the server machine). These components are marked with numbers 1 to 3 respectively in Figure 1: the recorder is generally included as a sensors monitoring a target application’s execution (1), the error reporting is generated and transmitted at the client (2) and received by the server (3), who stores the reports and replays them (3).

Figure 1 Error record/replay in the FastFix architecture.
The recorder tool, as its name implies, records the execution of FastFix-enabled applications in order to allow the application’s maintenance team to observe the sequence of user actions that led to the application’s error.

Once an error is detected by the application runtime or the user requests that an error report be sent, the FastFix system on the client device creates an anonymized error report and sends it to the maintenance team’s FastFix server containing a recorded trace of the application execution.

With the FastFix error report, the maintenance team can replay the execution and observe how the application error occurred and therefore more easily identify the error’s cause.

This document is a companion to the source code of the recorder/replayer tool in the FastFix platform. The following sections detail the design and usage of these components.
2 Overview

The recorder/replayer tool of FastFix is divided in client and server components. On client devices, FastFix records the execution of applications that are configured to use FastFix and creates anonymized error reports that are sent to the application’s maintenance team. The current version of the recorder tool monitors Java applications, both console applications and applications based on a graphical user interface (GUI), be it either the Eclipse Standard Widget Toolkit (www.eclipse.org/swt/) or Java AWT (java.sun.com/products/jdk/awt/). This covers the main types of existing Java applications. The recorder tool monitors the main source of application non-determinism which are user inputs (keystrokes and mouse events). The current tool focuses solely on user input, which is the most problematic source of non-determinism in terms of privacy, and does not tackle other source of non-determinism (such as random numbers or OS-specific identifiers). Additionally the recorder tool monitors two application triggers of error report: unhandled exceptions (for all types of applications) and explicit user requests for assistance (for SWT based applications).

Error reports are transferred to the FastFix server using the FastFix Communication System. Once at the server, error reports are stored and can be replayed at any time using the FastFix ticket browser.

2.1 Evolution since the First Iteration

There have been three main functional improvements to the FastFix recorder/replayer since the submission of deliverable D5.3, which corresponded to this component’s first iteration:

- Support for error report anonymization providing users with additional privacy guarantees regarding the user input that is transferred to the maintenance team’s servers.

- Broadening of the types of applications supported by the FastFix recorder/replayer, which include from now not only SWT-based applications but also AWT and console applications.

- Inclusion of a ticket browser giving maintenance teams a simple mechanism for
initiating the replay of error reports.

## 2.2 Code Metrics

Since the platform is based on the OSGi framework, the recorder/replayer tool is organized as a set of bundles hosted at https://repository.fastfixproject.eu/svn/fastfix (authenticated access). The bundles are:

- **eu.fastfix.client.faultReplication**: this bundle performs application execution information processing on the client side (7 classes, 33 methods, 239 lines of code [LOC]).

- **eu.fastfix.client.error.report.generation**: the bundle that manages the creation of error reports and their transmission to the FastFix Server (17 classes, 100 methods, 532 LOC).

- **eu.fastfix.targetapplication.faultReplication.sensing**: a context sensor which monitors the SWT application log for unhandled exceptions (10 classes, 31 methods, 223 LOC).

- **eu.fastfix.targetapplication.faultReplication.userReport**: Eclipse plug-in allowing application users to request assistance (2 classes, 6 methods, 35 LOC).

- **eu.fastfix.targetapplication.sensor.javaapplication**: Unhandled exception sensor and GUI anonymization code for AWT applications (26 classes, 207 methods, 2117 LOC)

- **eu.fastfix.targetapplication.sensor.moskitt**: Unhandled exception sensor for SWT applications (22 classes, 93 methods, 773 LOC)

- **eu.fastfix.targetapplication.sensor.reap**: Unhandled exception sensor and user input anonymization code for console applications (65 classes, 508 methods, 4264 LOC)

- **eu.fastfix.targetapplication.faultReplication.guiRecorder**: this bundle records relevant GUI events on a FastFix enabled application (15
classes, 43 methods, 393 LOC).

- `eu.fastfix.common.faultReplication.gui`: common utility classes for the GUI recorder and replayer (14 classes, 82 methods, 508 LOC).
- `eu.fastfix.server.faultReplication.guiReplayer`: GUI execution replayer (5 classes, 12 methods, 49 LOC).
- `eu.fastfix.server.faultReplication.guiReplayer.awt`: Replayer for AWT applications (7 classes, 22 methods, 296 LOC)
- `eu.fastfix.server.faultReplication.guiReplayer.swt`: Replayer for SWT applications (15 classes, 67 methods, 574 LOC)
- `eu.fastfix.server.error.reporting.abstractions`: Bundle with generic code for inserting error reports into a ticketing server (13 classes, 53 methods, 155 LOC)
- `eu.fastfix.server.error.reporting.jira`: Bundle for inserting error reports into a JIRA portal (56 classes, 942 methods, 8497 LOC).
- `eu.fastfix.server.error.reporting.eventum`: Bundle for inserting error reports into a Eventum portal (5 classes, 73 methods, 378 LOC).
- `eu.fastfix.server.error.reporting.trac`: Bundle for inserting error reports into a TRAC portal (21 classes, 119 methods, 749 LOC).
- `eu.fastfix.server.maintenance.ticketbrowser`: Bundle with the ticket browser for browsing and replaying error report tickets (11 classes, 31 methods, 199 LOC).

### 2.3 Document Organization

The remainder of this document’s focuses on the description of the functionality of the components of the record/replay tool: chapter 3 describes how FastFix records the execution of different applications, chapter 4 details the mechanisms for execution anonymization and chapter 5 illustrates how application replay operates in FastFix. Finally, chapter 6 describes
the details of how to configure and execute the tool.
3 Execution Recorder Tool

Execution recording is an essential part of creating an error report and requires monitoring the FastFix supported application. The mechanisms used for monitoring the applications depend on the application type (SWT, AWT or console) and are detailed in the following sections. The error report is sent by the Client Error Report Generation System described in D3.5.

3.1 Monitoring SWT applications

Any OSGi/SWT application that wants to use FastFix’s fault replication must use components (OSGi bundles in the present case) that perform two functionalities: recording the application’s execution and triggering an error report. The first aspect is assured by the eu.fastfix.targetapplication.faultReplication.guiRecorder bundle. From the beginning of the execution of any SWT/FastFix application all GUI events are recorded in a log file. The second part, error reporting can be automatic or explicit.

The automatic error reporting is performed by the eu.fastfix.targetapplication.faultReplication.sensing bundle which detects runtime errors in the form of unhandled exceptions. In SWT application unhandled exceptions are logged in an application log. Whenever this log monitoring sensor detects that an unhandled exception has been recorded it triggers an error report.

If error reporting is to be done explicitly by the user, the application must include the eu.fastfix.targetapplication.faultReplication.userReport bundle in its OSGi execution configuration. This will insert an option called “Request Assistance” in the “Help” menu. By selecting this option users will trigger an error report. The report will include a user written assistance request (as well as the execution log describing what the user did before requesting assistance).

The error report (unhandled exception or assistance request) is communicated within the platform by broadcasting an “Unhandled Exception” event on the FastFix Context Bus.
This context event is detected by the Client Fault Replication System, which then processes the information gathered by the application and its sensors and prepares the report to be sent to the FastFix server side. In the case of OSGi/SWT applications, all input data and GUI events are sent unmodified to the FastFix server.

3.2 Monitoring Java Console Applications

The execution record mechanism for Java console applications records all user input so that a deterministic replay of the application is possible in case of an application crash. This is achieved by instrumenting the compiled Java application using the SOOT bytecode instrumentation tool (www.sable.mcgill.ca/soot) in order to log all user input in a transparent fashion. Note that, in order to ensure deterministic error replay, one should log all sources of non-determinism of the program, and not solely user input. In the FastFix prototype, we have focused on recording user input in order to tackle the complex issue of error report anonymization.

Applications have to be instrumented by the development team before being deployed. When the application is deployed to the clients, two copies of the application have to be deployed, an instrumented one and a non-instrumented one. The instrumented version is the one normally executed by the users. The non-instrumented version is used by the anonymization process in order to search for execution paths within it.

Console applications’ user input is recorded with SOOT instrumentation that intercepts all calls to the `java.io` classes and records the input data. This is stored in a file which, when an application unhandled error report is triggered, is the input for the anonymization phase described in the next chapter.

3.3 Monitoring AWT Applications

In the case of AWT applications, the instrumentation mechanism is similarly based on SOOT instrumentation. However the instrumentation that is added to applications is different. In the case of AWT application, the recorded log contains an entry for each beginning and end of each event listener that is called from the user interface. Event listeners are the methods that are associated to specific graphical events in the application, e.g. the
code that is executed when a button is clicked or a menu item selected. Additionally it contains an entry for each variable that is read from a GUI graphical component (widget). These readings of widget contents are called *listener preconditions* in the sense that the widgets read by a listener must be filled in order for the listener to fulfil its goal. This monitoring activity results in a log file containing a sequence of listeners, and their preconditions, that fully describes what user code contained in event listeners was executed and which user input was read from the GUI widgets.

All tools for AWT instrumentation and monitoring are included in the eu.fastfix.targetapplication.sensor.javaapplication bundle.

### 3.4 Limitations

The main limitation of the FastFix application recording is that the current prototype does not completely gather all the possible sources of application non-determinism as it only records the main sources of user-input: *Java.io* operations, reading from *Java.util.Scanner* and the most common operations on graphical interfaces (writing in textboxes and mouse clicks).
4 Error Report Anonymization

The FastFix recorder/replayer tool provides anonymization for user input in AWT and console applications. It is unavailable for SWT application due to issues described in the “Limitations” section below. The two types of anonymization are user input anonymization and GUI anonymization. Console application can benefit form user input anonymization if the system is correspondingly configured. GUI applications can use user input and/or GUI anonymization, again, depending on the FastFix client configuration in place. The anonymization algorithms are executed at the client side between the application crash being detected and the resulting anonymized error report being sent to the application maintenance team at the FastFix server. The techniques described in sections 4.1 and 4.2 are detailed in two technical reports included as Appendixes A and B, which will be soon published as conference papers.

4.1 User Input Anonymization

FastFix user input anonymization is based on the idea of maximizing the achievable degree of obfuscation by exploiting the presence of alternative execution paths in an application leading to the same failure.

Primarily, we use symbolic execution in order to derive a set of logical constraints of the user input, called a path condition, which guarantee that the application will re-execute along the same execution path that previously led to failure. Alternative inputs reproducing the bug can then be drawn from the set of all inputs satisfying the identified path condition. This approach was shown in work by Castro et al.\textsuperscript{1} to have the potential to achieve high obfuscation levels since large portion of the input data can often be replaced by unconstrained symbolic values.

FastFix extends the idea of generalizing user input by calculating the path condition of the original execution. We extend it by discovering first an alternative execution path that leads

to the same application crash and only then calculating the alternative path’s condition (and a instantiation of that path condition).

FastFix user input anonymization pursues this goal by proceeding through three phases. All these phases can be executed in the background during periods when the client machine is idle. The first phase takes the original user input $I$ and calculates both the exact execution path taken by the failed application and the path condition associated with the original path. The second phase aims to identify an alternative failure-inducing input $I'$ that maximizes dissimilarity with the original user input $I$. To this end, we introduce a framework that allows generating a family of search heuristics aimed to efficiently explore the space of an application’s execution paths by performing circumscribed “deviations” around the branching points of the original execution path. The search heuristics used in the first phase operate in a deterministic fashion which, on one hand, maximizes the chances of quickly identifying a failure inducing input $I'$ having maximum dissimilarity from $I$, but, on the other hand, may allow to trace back, starting from $I'$, the original execution path.

To tackle this issue, we rely on a second phase, in which it uses a non-reversible, randomized search heuristic to identify a second alternative failure-inducing execution path, and the corresponding failure-inducing input $I''$. This new alternative path $I''$ is a variation of $I'$ that, due to its randomness, cannot be reversed by a malicious attacker.

Due to the non-deterministic nature of the search heuristic used in this phase, $I''$ results indistinguishable from any other failure-inducing execution paths, which allows REAP to achieve the theoretical lower bound on information leakage.

This algorithm is implemented in FastFix’s eu.fastfix.targetapplication.sensor.reap bundle.

4.1.1 Original Input Anonymization (OIA) Phase

When an application failure $f$ is detected by the unhandled exception sensor, the anonymization tool is triggered re-executes symbolically the application feeding it with the original failure-inducing input $I$ (stored in the execution log file). The OIA phase pursues a twofold goal: i) identifying the sequence of program statements composing the original
failure-inducing execution path, which we denote as $\phi$ ii) computing the path conditions, $P$, associated with the execution path. The anonymization tool used is the symbolic engine of Java Path Finder with extensions that guide the execution with the goal of reaching the original application crash (main.OIA class in the reap bundle).

### 4.1.2 Residue Minimization (RM) Phase

Next, user input anonymization enters the so-called residue minimization phase. In this stage, it performs a second symbolic execution to identify an alternative failure-inducing execution path $\phi'$ that minimizes residue, i.e. such that the corresponding fault triggering input $I'$ has as few bytes as possible in common with the original user input $I$. To this end, we use a framework (main.Circumventor class in the reap bundle) that allows deriving a family of search heuristics for exploring the set of possible execution paths of a program. We discuss the search heuristics’ framework in the technical report in Appendix A, but the key idea at the basis of these algorithms is that the chances of encountering an alternative, failure-inducing execution path $\phi'$ are higher by remaining in proximity of the original execution path. Driven by this rationale, when we encounter a conditional test on some input-dependent variable, it can decide to explore an alternative execution path by forcing the symbolic execution engine to take a branch different than the one selected during the original execution. Note that each of these “detours” correspond to speculating on different values for the input. These speculations are tracked by the path condition that is dynamically built during the symbolic execution. Our framework allows customizing the behaviour of the search heuristics by means of three main parameters, that control the selection of the detouring branches as well as the maximum length of admissible detours (after which the search heuristic backtracks). The settings of these parameters affect the heuristics’ behavior and explore different trade-offs between execution time, and obfuscation. If this stage succeeds, it provides as output, a path condition $P'$ of an alternative failure-triggering execution path $\phi'$.

The path condition $\phi'$ allows generating failure-triggering input values $I'$ that are typically more obfuscated than those produced by the OIA phase alone. This is unsurprising, given that $I'$ triggers the bug along an execution path $\phi'$ that does not fully overlap with the
original one. It is important to highlight that, in order to achieve an actual enhancement of the user privacy, it has to be ensured that, given $P'$ (which can be automatically reconstructed from $I'$), it is impossible to identify which was the path condition $P$ associated with the original execution path. Note that this is not a priori guaranteed, since, in order not to unnecessarily constrain the search heuristics design space, we have not assumed their non-reversibility. This issue can, however, be circumvented by simply performing a logical disjunction between $P$ and $P'$, and using the resulting path condition $P \cup P'$ to determine a fault-inducing input $I^*$. This is, in fact, sufficient to make it impossible to distinguish whether the input $I'$ was associated with the original or the alternative execution path. Therefore, from an information theoretical perspective, the increase in obfuscation (measured in terms of leakage of information bits) achieved thanks to the discovery of $\phi'$ during the residue minimization phase can be quantified in terms of the increase of the number of input satisfying $P \cup P'$ with respect to the number of solutions satisfying only $P$.

However, we aim to achieve the lower bound on information leakage achievable given a failure $f$, i.e. the leakage achievable by randomly selecting an input from the set, $I_f$ containing every input triggering $f$.

### 4.1.3 Leakage Minimization (LM) Phase

To this end, we execute a third, so called, leakage minimization (LM) phase. In this phase, we perform a final symbolic execution that, starting from $\phi'$, relies on a non-reversible algorithm to identify a second failure-inducing execution $\phi''$. It is essential that the final result does not allow reversing the process and identifying the original input $I$ from which the process started, hence the random nature of this last phase. Our search heuristics’ framework (main.Circumventor class in the reap bundle) is sufficiently flexible to support the generation of non-reversible algorithms by simply picking random values for the parameters that control the heuristics’ behavior. Given the randomized nature of this search algorithm, the alternative failure-inducing execution path, $\phi''$, it identifies results indistinguishable from any other execution path in $\phi$. Consequently also the input generated using $\phi''$ (i.e. computed as a random solution of $\phi''$) results indistinguishable from any other
failure-inducing input in $I_f$. The randomness of this path provides us with a level of information leakage, in terms of the revealed information bits of the original input that is equivalent to having calculated all possible execution paths leading to the original crash.

The output of the LM phase will be used as the error report sent to the maintenance team for eventual error replay.

### 4.2 GUI Anonymization

The GUI anonymizer in FastFix targets Java/AWT applications. It is implemented in the `eu.fastfix.targetapplication.sensor.javaapplication`. This component takes the execution log of an application that has crashed and generates the minimum set of GUI operations needed to replay the crash using the Minimum-Set Listener Reduction algorithm, described below.

It should be noted that this technique creates user data anonymity by eliminating user input data that is unnecessary to the replay of a crash. However, contrary to the user input anonymization described above, it does not alter the user input data. This is why FastFix gives the option of combining it with user input anonymization (GUI anonymization first and the user input anonymization), which is described in the next section.

All the GUI anonymization actions are executed in the background so that the normal behavior of the client machine need not be disrupted.

The anonymizer uses two auxiliary components, the execution generator\(^2\) and the tester, which we describe before the main algorithm for clarity. These are respectively responsible for converting a listener log into an event log and for testing if a given log produces a specific error.

#### 4.2.1 Execution Generator

The execution generator is used by the anonymizer component in order to convert a listener sequence into an event sequence which can then be injected into the GUI of the target application. This is needed because, although it is much more efficient to record and

\(^2\) In Appendix B, the execution generator is referred to as the converter.
anonymize sequence of listeners (and not low level graphical events), in order to act on the GUI of an application it is necessary to identify exactly which low level graphical events (mouse moves, key presses, etc.) need to be created. The original sequence of listeners is processed from the beginning to end. For each listener, its preconditions are analyzed and converted into a sequence of events in the end the listener itself is converted into a sequence of events. Finally all the events are added into a new sequence of events. When a listener sequence is fully converted, the resulting event sequence is recorded onto a file so that other modules can use it later.

For example, if a given listener has a precondition which states that a read operation was made from a text field widget with a given id, and the value read was a string, the execution generator is going to create all the necessary events to put the specific value onto the indicated widget. In this case it would be something similar to: 1) selecting the widget, and 2) typing all the characters of the string (one event per character). If the listener is registered for clicks and is placed in a widget that is a button, then the execution generator knows it has to generate a click in that widget in the end.

4.2.2 Tester
This module is entrusted with receiving an event log and testing it, in order to check if it triggers an exception, and if that exception is the same as the one which happened in the original execution. The tester injects the events in the log into the target application in the same way the replayer (on the FastFix server) does, which is, recreating the events and injecting them in a given widget that is found using the DWI. When the log is fully replayed the module compares the resulting exception, if any, with the one in the original log. If they match, then the hypothesis is marked as a valid one.

4.2.3 The Minimum-Set Listener Reduction Algorithm
The Minimum-Set Listener Reduction algorithm is applied in two phases, the delimitation phase, and the reduction phase. In the first phase, the goal of the algorithm is to find the shortest suffix of the recorded sequence of listeners that is needed to trigger the observed error. The second phase finds out, given the suffix sequence, which listeners are unnecessary and can be removed.
In the delimitation phase the algorithm generates all the test cases with the \( n \) last listeners from the original sequence, starting with \( n = 1 \) and ending when \( n \) reaches the size of the original sequence. After all the test cases are generated, they are sorted by size in ascending order, and are then tested. We generate all tests before testing any of them, because it is much cheaper to generate tests than to run them and therefore the algorithm generates all tests first in order to test them in ascending length order. The first test that succeeds in replaying the observed error is the shortest suffix of the original sequence that can reproduce the error.

```java
oseq = original listener sequence
seqlist = ∅
newseq = ∅
for oseq.size ≠ 0 do
    newseq.addFirst(oseq.removeLast())
    seqlist.add(newseq)
end for
```

**Algorithm 1 Delimitation Phase Test Generation**

The reduction phase uses as fixed points the first and the last listeners of the suffix sequence, and then generates all the possible combinations of the listeners in between, always maintaining the order of the listeners in the original sequence. In order to generate all the possible sequences we developed a combinatorial algorithm, which transverses the list and for each element creates two scenarios, one in which the element is on the list, and one in which is not. After this, each possibility recursively calls the algorithm. With this, we are able to generate all combinations of listeners while still preserving the order of the original sequence. After all the sequences of the reduction phase are found, they are sorted in increasing size order and tested until a valid one is found. In the end we will have a reduced sequence that is the shortest sequence that still triggers the error.

```java
oseq = output listener sequence from delimitation phase
seqlist = ∅
newseq = ∅
if oseq.size > 2 then
    newseq.addLast(oseq.getFirst())
    newseq.addLast(oseq.getLast())
    Scramble(oseq, newseq, seqlist, 1)
```

**Algorithm 2 Reduction Phase Test Generation**
Algorithm 2 Reduction Phase Test Generation

```plaintext
definition SCRAMBLE(oseq, newseq, seqlist, i)
  if i < oseq.size - 1 then
    Scramble(oseq, newseq, seqlist, i + 1)
    aux = seq
    aux.removeLast()
    aux.addLast(oseq.get(i))
    aux.addLast(oseq.getLast())
    seqlist.add(aux)
    Scramble(oseq, aux, seqlist, i + 1)
  end if
end function
```

Algorithm 3 Combinatorial Algorithm

As one can notice both phases of the algorithm rely on brute-force techniques. However, the algorithms we developed work for modern graphical user interfaces. This is because of the average size of the listener sequences, which trigger an error, is small.

We also developed a heuristic to optimize the process of testing for valid test cases, therefore mitigating the use of brute-force techniques. The goal of the heuristic is to identify invalid tests and remove them without the need to convert and test them. After each turn of generating tests, the system consults a file describing the application’s widget graph and checks whether all the constraints between listeners are maintained; if not the test case is deleted. This is done by consulting the graph and observing if each listener of a given sequence needs to have any specific listener executed before it can be invoked. For example, if a listener $L3$ requires that another listener $L1$ be executed beforehand, all test sequences that include $L3$ without $L1$ before it, can be discarded. An example of this is a login window. Let’s say we have a window containing widget $W1$ with listener $L1$ registered to it, when that listener is invoked a new login window is created. In the new window there is another widget $W2$ with a listener $L2$ registered to it. This means that, if a sequence has listener $L2$ but does not include before $L1$, then the sequence is not valid since there is no way of generating an event for a window that is not created.
4.3 Combination of GUI and User Input Anonymization

In the case of AWT applications, the user input and the GUI anonymization techniques can be combined therefore providing a higher level of anonymization of the error reports.

This is achieved simply by submitting the log of listeners (and their preconditions), before or after user input anonymization (depending on the FastFix client configuration) to the user input anonymization logic.

This results in an error report containing the minimum set of listeners, output by the GUI anonymization, where all user input written into widgets is also anonymized. For this, the user input anonymization, as described above, is applied not to the application as a whole but to each one of the listeners in the minimum listener sequence. This sequence is obtained first by running the GUI anonymizer. Currently, in this case of the combination of the two types of anonymization, only the OIA phase of user anonymization is applied as justified in the following section.

4.4 Limitations

The main limitations, and potential future work, of the anonymization mechanisms in the Fastfix record/replay components are:

- Currently, no information is displayed to the user about the difference between the original user data she input and the data that was not anonymized and that is included in the error report included in the submitted error report.

- The current prototype only performs the first phase of user input anonymization when used together with GUI anonymization. The justification is that the second and third phases of user input anonymization can only be performed if there were a mechanism that, even while using alternative execution paths, the application would execute the same operations that create widgets. The most common case of widget creation is opening a window. If the second phase of user input anonymization takes the application to a path that changes the widgets that are created, then the GUI actions listed in the listener log are no longer valid. This mechanism to ensure that alternative execution paths do no circumvent operations that create widgets is not in
There is no anonymization for SWT-based applications. This is due to the fact that it is not possible in the SWT graphical framework to intercept GUI operations so that widgets can be uniquely identified. This renders the recording of GUI operations on the client side meaningless since widget identifiers recorded on the client side will have no meaning on the FastFix server machine where the error is replayed. Hence, applications are recorded at the level of operating system graphical events and cannot therefore be submitted to the same GUI anonymization we described above.
5 Execution Replayer Tool

The FastFix execution replayer tool replays executions recorded at the FastFix server site. Error reports are stored in a ticketing system server. There are plug-ins to FastFix to support using both the TRAC and Jira ticketing systems (eu.fastfix.server.error.reporting.trac and eu.fastfix.server.error.reporting.jira). Additional files related to each of the error tickets are stored in the FastFix server’s file system. FastFix supports replaying error reports coming from all the types of applications it monitors: console, AWT and SWT applications.

When the maintenance engineers using the FastFix server replay the error associated to a particular ticket, the corresponding application is automatically started (see Figure 1 top right corner).

![Screenshot of the replay of an error report from Moskitt, a FastFix-enabled application.](image)

The engineers should insert the error report’s ticket id (obtained from the ticketing system’s
interface) and the exact execution which triggered that ticket will be run. The execution replay can happen at the same speed as it originally happened on the client or it can be replayed with a user-defined time interval between replay steps. The replacer makes it possible to fast-forward the replay to a particular point in time. Additionally, the replacer displays context events that happened at the client device during the execution of the replayed application such as mouse clicks and key presses (see for example Figure 1 bottom right corner). An important restriction of the SWT replay is that the replayed application must be placed in the same screen location as it was originally run and the screen resolution of the maintenance machine must be at least as large as that of the original client device. This is due to the fact that, in the case of SWT applications, the original graphical events that were recorded on the client device are replayed exactly where they happened on the original screen and so the replay application needs to be in the same position.

5.1 Replaying from the Ticket Browser

The standard way to start the replay of a FastFix enabled application is from the ticket browser.
Figure 3 FastFix's ticket browser.

The ticket browser is an Eclipse plug-in. It lists all FastFix tickets in the server’s ticketing system. By right clicking on any ticket, the maintenance engineer can start the replay of a ticket or browse that ticket in the ticketing system’s website. The ticket description has embedded the information describing how to replay a ticket. This is transparent to the end-user.
6 Running FastFix Record-Replay

Currently the FastFix platform is run as a set of OSGi components. The best way to test the components described in this document is in the context of the FastFix development environment.

6.1 Preparing Applications: Instrumentation

6.1.1 SWT Applications

Setting up SWT applications for FastFix error record/replay requires no previous instrumentation. This is due to the fact that the sensor described in 3.1 records all the necessary event for SWT replay.

6.1.2 Console Applications

In the case of console applications, the compiled application needs to be instrumented in order to support FastFix record/replay.

Assuming that the main class of the console application is called MainClass. We must run:

```java
java -cp .:<path-to-FastFix_server>/plugins/eu.fastfix.targetapplication.sensor.reap
edu.hkust.leap.transformer.LEAPTransform -Xmx2g MainClass
```

This creates a version of the application for running at the client in a subfolder called instrumentResult.

Both the original and the instrumented Java class files of the application need to be copied, by the deployment process, to the client using them.

In order to deploy the application, it must be connected to the sensor that will monitor potential application crashes (unhandled exceptions). The shortcut to start the application should point to the sensor’s main class `eu.fastfix.targetapplication.sensor.reap.StartSensor`. In order for the sensor to start the correct application, there is a configuration file in the sensor bundle’s resources folder named appProperty, which should be configured to point to the application
to be run (see Figure 4 below).

```plaintext
# enable/disable the user input anonymization
EnableUserInputAnonymizer=no

# path to the application
MainClassPath=/Users/myUser/AppFolder/AppMainClass

# path to the application’s instrumented version
InstrumentedClassPath=/Users/myUser/AppFolder/instrumentResult

# application’s main class
MainClassName=AppMainClass

# main class’ arguments
ArgumentsForMainClass=null
```

Figure 4 appProperty file for configuring an AWT application and its sensor.

### 6.1.3 AWT Applications

In the case of AWT, the machine where applications are instrumented must have the GUIAnon.jar which comes from generating an executable jar of eu.fastfix.targetapplication.sensor.javaapplication and contains all utilities for GUI anonymization. If you run:

$java -jar GUIAnon.jar

, you’ll be provided with all available options of GUIAnon (see Figure 5 below).

<table>
<thead>
<tr>
<th>Usage Help</th>
<th>java -jar GUIAnon.jar -h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch app</td>
<td>java -jar GUIAnon.jar -launch</td>
</tr>
<tr>
<td>Instrument MainClass</td>
<td>java -jar GUIAnon.jar -instrument</td>
</tr>
<tr>
<td>Rip Graphical Interface</td>
<td>java -jar GUIAnon.jar -rip</td>
</tr>
<tr>
<td>Record Events MainClass</td>
<td>java -jar GUIAnon.jar -recevents</td>
</tr>
<tr>
<td>Record Listeners MainClass</td>
<td>java -jar GUIAnon.jar -reclisteners</td>
</tr>
<tr>
<td>Convert Tracefile</td>
<td>java -jar GUIAnon.jar -convert12e</td>
</tr>
<tr>
<td>Generate anonymized</td>
<td>java -jar GUIAnon.jar -anonymize</td>
</tr>
<tr>
<td>guistructure tracefile MainClass</td>
<td>java -jar GUIAnon.jar -anonymize</td>
</tr>
</tbody>
</table>

```
Generate anonymized: java -jar GUIAnon.jar -smartanonymize guistructure tracefile MainClass

Figure 5 GUIAnon usage instructions

To instrument the application (assuming a main in MainClass), we must run:

$java -jar GUIAnon.jar -instrument MainClass

which generates an instrumented version of the application in a folder placed in the same parent folder as the main class and which is called instrument_result.

Then, we must extract the graphical model of the AWT application by running:

$java -jar GUIAnon.jar -rip MainClass

Select all options available in the application’s GUI (click every button, open every menu, etc….). Then close the application, which generates a file called model.txt.

Just as in the case of console applications, in order to deploy the application, it must be connected to the sensor that will monitor potential application crashes (unhandled exceptions). The shortcut to start the application should point to the sensor’s main class eu.fastfix.targetapplication.sensor.javaapplication.StartJavaApplicationSensor. In order for the sensor to start the right application, there is a configuration file in the sensor bundle’s resources folder named appProperty, which should be configured to point to the application to be run (see Figure 6 below).

```
# enable/disable the GUI anonymization
EnableAnonymizer=no

# enable/disable the user input anonymization
EnableUserInputAnonymizer=no

# path to the application
MainClassPath=/Users/myUser/AppFolder/AppMainClass

# path to the application’s instrumented version
InstrumentedClassPath=/Users/myUser/AppFolder/instrumentResult

# application’s main class
MainClassName=AppMainClass

# main class’ arguments
ArgumentsForMainClass=null
```
D5.4: 2nd prototype of the execution recorder/replayer tool

6.2 Server Configuration

The server’s configuration.xml file must be configured to point to the correct issue tracker (see Figure 7 below). The IssueTracker section is used to point to the issue tracker server.

```xml
<IssueTracker id="fastfix_issuetracker_configuration" url="ticket_server.mydomain.tld/path">
    <!-- The Ticket element has parameters to configure several aspects of the tickets for the target issue tracker. -->
    <Ticket attachmentMaxSize="2048" />
</IssueTracker>
```

Figure 7 Server configuration.xml sections for error record/replay

6.3 Client Configuration

The client’s configuration.xml file must be configured to point to the correct FastFix server (see Figure 8 below). The LogStore section is used to reference the ssh daemon at the FastFix server (host, user and password) and the folder where error reporting logs may be stored (location).

```xml
<FaultReplication id="fastfix_fault_replication_configuration">
    <!-- The LogStore element has parameters to copy the logs to a remote server. -->
    <LogStore host="test.domain.com" user="fastfix_replication" password="IHaveNone"
              location="/fault_replication/logs" />
</FaultReplication>
```

Figure 8 Client configuration.xml sections for error record/replay

6.4 Running FastFix Record-Replay as an End-User

A. On the FastFix Server Machine:

Requirements:

- MySQL installed (e.g. MAMP or LAMP)
• Java 1.5 installed
• SSH server / daemon running.
• The application you want to run within FastFix.
• Access to a TRAC or Jira ticketing server.

Installation:
1. **Download FastFix Server.**
   Unzip the downloaded file into a directory “fastfixserver”.
2. **Start** the FastFix Server by double-clicking on the JAR or EXE file in the “fastfixserver” directory.
3. Open the preferences menu (Menu “FastFix Server -> Preferences…”)
4. FastFix has to connect to the MySQL instance. Please make sure that the specified **host**, **port**, and **credentials** match your MySQL installation.
5. **Download MOSKitt replayer bundles**. These bundles should be added to the runtime of the OSGi application that you want to run within FastFix.

B. On the “Client Machine”:

Requirements:
• Java 1.5 installed
• The SWT OSGi application you want to run within FastFix or, in the case of AWT or console applications, the two versions of the application, original and instrumented.

Installation:
1. **Download FastFix Client** from
   Unzip the downloaded file into a directory “fastfixclient”.
2. **Start** the FastFix Client by double-clicking on the JAR or EXE file in the “fastfixclient” directory.
3. **Start your application.** If it’s an SWT/OSGi application make sure that the run configuration includes the following FastFix bundles:
   eu.fastfix.client.faultReplication.guiRecorder,
   eu.fastfix.client.faultReplication.sensing,
   eu.fastfix.client.faultReplication and
   eu.fastfix.common.faultReplication.gui

Usage:
1. **Switch** to the FastFix Client.
2. Open the preferences menu (Menu “FastFix Client -> Preferences…”)
3. FastFix client has to send application execution traces to the “Server Machine” via SCP. Please make sure that the specified **host**, **username**, and **password** match the SSH / SCP parameters on the “Server Machine”.
4. In the UI you should see **two sensors** as shown in Figure 3.

![Figure 9: FastFix Client with MOSKitt Sensors](image)

5. FastFix is now **up and running**.
6. To test, just work normally with your application.

7. If FastFix detects an **unhandled exception**, you can find the corresponding error report at your TRAC server.
8. **To replay** the application execution that lead to this exception, switch to the “Server Machine”.
9. **Start your application** on the “Server Machine”
10. You should see **two new replayer windows**, as shown in Figure 4
11. **Do not use the application features** now.
12. Enter the **ticket number** of the ticket to replay in the replayer menu.
13. Hit **Replay** to replay the error.
Figure 10: An OSGi SWT application being replayed by FastFix on MS Windows
7 Appendix A – Technical Report on User Input Anonymization
REAP: Reporting Errors using Alternative Paths

- authors removed for double blind submission -

Abstract

Software testing is often unable to detect all program flaws. These bugs are most commonly reported to programmers in error reports containing core dumps and/or execution traces, that frequently reveal users’ private information. Hence, these mechanisms are sparsely used due to users’ data privacy concerns. This paper presents REAP, a new fault replication method for error reporting, which provides an unprecedented privacy level. REAP uses symbolic execution to calculate alternative execution paths leading to an observed error, relying on randomization techniques to achieve the theoretical upper bound on the information leakage necessary to replay a given bug. Our results show that REAP achieves, in a scalable way, reductions up to 85% of residue and up to 96.89% on the number of bits revealed.

1. Introduction

Despite more than half of the resources in a typical development cycle being invested in testing and bug fixing, it is common for software errors to manifest themselves after software is released and persist long after [1]. Software bugs represent several billion dollars per year worth of maintenance costs in Europe and in the US [2] alone.

Error reporting tools are currently the most popular tools to provide developers with information about application crashes ([3, 4] for example). These tools aim to allow software vendors to fix bugs in a timely manner. However, error reports usually include solely partial snapshots of the memory, stack traces of the process and a description of the faulty scenario, which is often insufficient to reproduce the error.

Fault replication mechanisms address the shortcomings of classical error reports, by allowing to reproduce, at the development site, a faulty execution taken place at the client side. These mechanisms monitor the target applications in an efficient manner in order to gather enough information for execution reproduction, while imposing the least overhead possible. Numerous fault-replication mechanisms have been developed in the past years and are becoming more and more capable of efficient application monitoring and successful bug reproduction ([5–7] to name a few).

Unfortunately, privacy and security concerns have prevented widespread adoption of many of these techniques and, because they rely on user participation, have ultimately limited their usefulness [8]. In fact, whether the user is working on a confidential document or has input personal information, private sensitive information is likely to be included both in the memory snapshot taken to generate an error report or in the non-deterministic sources logged by fault replication mechanisms [9].

A promising approach aimed at tackling privacy concerns of existing fault replication mechanisms is based on the idea of obfuscating sensitive information inserted while still ensuring the reproduction of the faulty execution ([8, 9, 11]). These mechanisms use symbolic execution (e.g. [12]) in order to derive a set of logical constraints of the user input, called path condition [13], which guarantee that the application will re-execute along the same execution path that previously led to failure. Alternative inputs reproducing the bug can then be drawn from the set of all inputs satisfying the identified path condition. This approach was shown to have the potential to achieve high obfuscation levels since large portion of the input data can often be replaced by unconstrained symbolic values. However, the degree of obfuscation achievable by these techniques is directly dependent on the restrictiveness of the constraints of the path condition (i.e. on the cardinality of the set of inputs that match a given constraint), which can be critically affected by the application’s structure and bug placement in the code.

In this paper we propose REAP (Reporting Errors using Alternative Paths), a novel approach based on the idea of maximizing the achievable degree of obfuscation by exploiting the presence of alternative execution paths leading to the same failure. The work developed in [14], took the first step towards this goal, showing that the potential privacy enhancements achievable by taking into account all possible failure-inducing execution paths (and not only the original one) can be in practice extremely relevant. Indeed, by identifying the entire set of failure-inducing path conditions, it is possible to attain, the theoretical lower bound on information leakage, that is, the least amount of bits (of the in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied] . . . $10.00
put) revealed while still reproducing the same failure. On the down side, that work highlighted the difficulty of attaining this theoretical lower bound, and the severe scalability issues affecting approaches (such as the one proposed in that paper), which rely on symbolic execution to identify the entire set of alternative failure-inducing path conditions.

The system presented in this paper, REAP, is the first solution capable of achieving the theoretical lower bound on information leakage in a practical and scalable way. Specifically, REAP allows achieving the same obfuscation degree attainable by mechanisms that determine the entire set of failure-inducing path conditions, while incurring solely in the cost of identifying two alternative failure-inducing path conditions.

REAP pursues this goal by relying on two phases. The first phase aims to to identify an alternative failure-inducing input $I'$ that maximizes dissimilarity with the original user input $I$. To this end, we introduce a framework that allows to generate a family of search heuristics allowed to efficiently explore the space of an application’s execution paths by performing circumscribed “deviations” around the branching points of the original execution path. The search heuristics used in the first phase operate in a deterministic fashion which, on one hand, maximizes the chances of quickly identifying a failure inducing input $I'$ having maximum dissimilarity from $I$, but, on the other hand, may allow to trace back, starting from $I'$, the original execution path.

To tackle this issue, REAP relies on a second phase, in which it uses a non-reversible, randomized search heuristic to identify a second alternative failure-inducing execution path, and the corresponding failure-inducing input $I''$. Due to the non-deterministic nature of the search heuristic, used in this phase, $I''$ results indistinguishable from any other failure-inducing execution paths, which allows REAP to achieve the theoretical lower bound on information leakage.

We present the results of an experimental analysis based on 6 publicly available applications and 8 different bugs, which is aimed at assessing the feasibility of the proposed solution in realistic settings, and at quantifying the obfuscation quality enhancements achievable with respect to state of the art solutions. The results show that, when compared to state of the art solutions analyzing solely the conditions of the original execution path, REAP can achieve, on average, up to 87% reduction of information revealed and in a comparable execution time. Further, REAP can identify alternative inputs in the matter of minutes even when employed with large scale applications, for which it would not be viable to employ solutions requiring the identification of all failure-triggering executing paths [14].

This paper is organized as follows. In Sec. 2 we overview existing obfuscation mechanisms and discuss their main strengths and limitations. Section 3 presents the REAP system. We evaluate the proposed system in Sec. 4 before presenting some concluding remarks.

2. State of the art and motivations

2.1 Final application state error reporting

Initial approaches to automatic error report, such as Windows Error Reporting [3] and Mozilla Crash Report [15] involved mainly information collected at the end of a failed program execution. When an application crashes, the error reporting system gathers information acritically from the state of the process at the moment of the crash and submits it as an error report, if authorized by the user. Two major disadvantages of these methods stand out: i) there is no filtering of the submitted information regarding users’ privacy preservation, which means that sensitive information may end up be incorporated in the dump of the application state performed upon the occurrence of the bug [9]; ii) the generated report does not provide any historical information on how the error was reached, which typically makes the reproduction of the bug a complex and time consuming task. In fact, the search for the causes of the state reported is one of the tasks on which programmers spend more time [16].

One of the first systems to attempt to filter user private information from error reports was Scrash [17]. Scrash applications have all their sensitive data marked as such during development, and allocated in a specially reserved area of memory. When an error report is submitted for a Scrash enabled application, all the sensitive variables are removed. This approach has three main problems. First, it requires access to an application’s source code. Second, it assumes that the application programmers are trustworthy and will mark all sensitive data as such. And finally, error reports that have been amputated of relevant data may not allow for the full replay of the original error.

2.2 Input anonymization in fault replication systems

Fault replication systems are way more useful for software maintenance than core dumps or stack traces. On the other hand, fault replication systems that transmit to the maintenance site, the original user input, raise arguably even larger privacy concerns. In literature, two main approaches have been proposed to identify anonymized, failure-inducing inputs: techniques based on input minimization [10] and on path condition analysis [8, 9].

Input minimization techniques [10] were originally designed to speed-up testing/debugging and attempt repeated random removals of input chunks, in order to identify input fragments that are irrelevant for the reproduction of the bug. By allowing to purge irrelevant inputs, these techniques can enhance privacy. However, as discussed in previous works [8, 9], due to their purely random nature, input minimization techniques typically fail in frequent scenarios in which valid inputs must respect precise structural conditions (e.g. a credit card number must be composed of exactly 16 digits.
Approaches based on path condition analysis \cite{8, 9} overcome these limitations, by reasoning on the logical constraints imposed by the conditional branches that were taken during a failure-inducing execution, i.e. its path condition. For example, if a program asks the user for her age (e.g. 35), stores it in a variable \textit{age} and then successfully tests whether she is over 18, the path condition will include the logical clause \textit{age} > 18, reflecting the fact that this execution path can be replayed if the \textit{age} variable has any value larger than 18. In other words, the logical restrictions imposed by a path condition delimit the domain from which input values can be chosen and still trigger the same error. These solutions in \cite{8, 9} re-execute symbolically with the original input in order to determine the path condition that lead to the application crash. Once the path condition is identified, a logical solver \cite{18–20} can then be used to generate alternative, obfuscated inputs that satisfy the path condition of the faulty execution.

The two main metrics to evaluate privacy in this context are also put forward by these authors: \cite{9} uses the \textit{number of leaked information bits} (henceforth called \textit{leakage}) and \cite{8} uses the \textit{residue}. Leakage is strongly connected to information theory and is therefore a more formal representation of privacy. The leakage of a particular path condition is calculated from the proportion \(\alpha\) of the domain of the application input variables that satisfy the path condition. The leakage is equal to \(-\log_2(\alpha)\). The residue is a more intuitive and user-friendly metric defined as the number of input characters that remain unchanged after an anonymization process.

These techniques \cite{8, 9} have been shown to achieve significant enhancement of the bug report’s privacy in a range of application contexts. However, the degree of obfuscation attained by these approaches is critically affected by the restrictiveness of the logical clauses in a path condition.

The simple code excerpt in Figure 1, is used in the remainder of this paper to motivate and illustrate the behavior of REAP and exemplifies both the potentialities and limitations of the two aforementioned approaches. The code excerpt suffers of a divide by 0 exception caused by a wrong initialization of variable \(n\) in line 5, which manifests itself in line 12. Now let us assume that the user has input: \textit{age} = 26, \textit{isMale} = true, and \textit{isMarried} = false. The path condition derived from the execution with this data yields the constraints:

\[
\text{age} \in [25, \text{MaxInt}] \land \text{isMale} \land \neg\text{isMarried} \tag{1}
\]

In such a scenario, the age input by the user can be (partially) obfuscated by replacing it with any value larger than 25. The remaining input values, on the other hand, will have to be fully disclosed.

It should be noted that in the program of Figure 1, it is actually possible to achieve total input anonymization, given that the bug manifests in all possible execution paths, and, hence, independently from the value of the user input. As already mentioned, MPP \cite{14} is, to the best of our knowledge, the only system that attempts to exploit the presence of multiple failure-inducing execution paths in order to maximize the obfuscation level of a bug report.

By considering the disjunction of the path conditions of all execution paths leading to a bug, MPP can achieve, at least for small scale programs, the theoretical lower bound on information leakage, identifying all the possible inputs that replay the bug. Unfortunately, MPP suffers from significant scalability limitations for two main reasons. MPP relies on an off-line reachability analysis that performs a symbolic execution of the program, and produces as output, for all lines of code, a path condition and a triggering input for all the execution paths that traverse that line of code. Even if MPP employed a number of optimizations to minimize the costs of this off-line phase, experimental data clearly highlight that these can be prohibitive in complex/large scale applications. Also, as not all the execution paths identified during the symbolic execution may actually trigger the bug, the client needs to re-execute all of them, in order to verify which subset of the paths actually reproduces the error. This can be quite inefficient especially if the bug is located in a line of code that happens to be reachable through a high number of paths.

In the considered code example, MPP would generate 8 path conditions, each one associated with different combinations of the three tests on the input variables. As all of these paths lead to the bug, the disjunction of their path conditions yields a total relaxation of the constraints on the input variables, and achieves perfect anonymization. Unfortunately, the price for attaining such a striking boost of the input obfuscation grows very quickly with the size of the program. Assume, for instance, that the program contained a generic

1\textit{http://www.idate.co.uk/}
number $n$ of tests on different input variables. In this case, MPP would incur in costs (e.g., number of paths traversing the line that generated the bug) that grow exponentially with the value of $n$.

REAP seeks an innovative balance between efficiency and anonymity in the design space of privacy preserving fault replication mechanisms, as illustrated by Figure 2. At the extreme of lowest anonymity are systems like Castro et al. and Camouflage (see Sec. 2.2), which explore only one execution path (the one taken by the application user). On the other hand, MPP, by exploring all execution paths leading to the observed point of crash, can provide the maximum possible anonymity, but suffer of severe scalability limitations. As we will show in Sec. 3, REAP strikes a balance between these two extremes by achieving an anonymity level (in terms of leakage) equivalent to that of MPP, while circumventing its scalability issues thanks to the joint usage of efficient search heuristics and randomization techniques.

3. REAP

This section is devoted to presenting REAP. We start by providing an overview of the system. Next we present the framework that REAP uses to generate search heuristics aimed at identifying alternative failure-inducing execution paths. Finally, we discuss the anonymization capabilities of REAP.

3.1 Overview of the system

The various stages of execution of REAP are illustrated by the the diagram in Figure 3, and described in the following.

Original Input Anonymization phase. Similarly to existing fault-replication systems [8, 9, 14], REAP relies on automatic code instrumentation to log user inputs in a transparent fashion. When an application failure $f$ is detected, REAP re-executes symbolically the application feeding it with the original failure-inducing input $I$ (just as in the systems in [8, 9]). The OIA phase pursues a twofold goal: $i)$ identifying the sequence of program statements composing the original failure-inducing execution path, which we denote as $\phi$; $ii)$ computing the path conditions, $P$, associated with the execution path $\phi$.

Residue Minimization phase. Next, REAP enters the so-called residue minimization (RM) phase. In this stage, REAP performs a second symbolic execution to identify an alternative failure-inducing execution path $\phi'$ that minimizes residue [8], i.e. such that the corresponding fault triggering input $I'$ has as few bytes as possible in common with the original user input $I$. To this end REAP introduces a framework that allows to derive a family of search heuristics for exploring the set of possible execution paths of a program. We discuss the search heuristics’ framework in detail in Section 3.2, but the key idea at the basis of these algorithms is that the chances of encountering an alternative, failure-inducing execution path $\phi'$ are higher by remaining in proximity of the original execution path $\phi$. Driven by this rationale, when REAP encounters a conditional test on some input-dependent variable, it can decide to explore an alternative execution path by forcing the symbolic execution engine to take a branch different than the one selected during the original execution. Note that each of these “detours” correspond to speculating on different values for the input. These speculations are tracked by the path condition that is dynamically built during the symbolic execution.

As we will discuss more in detail in Section 3.2, our framework allows customizing the behaviour of the search heuristics by means of three main parameters, that control the selection of the detouring branches as well as the maximum length of admissible detours (after which the search heuristic backtracks). The settings of these parameters affect the heuristics’ behavior and explore different trade-offs between execution time, and obfuscation.

If this stage succeeds it provides as output, a path condition $P'$ of an alternative failure-triggering execution path $\phi'$. As we will see in the evaluation in Section 4, at the end of this stage, the path condition $P'$ allows generating failure-triggering input values $I'$ that are typically more obfuscated than those produced by solutions based exclusively on the analysis of the original path, such as [8, 9]. This is unsurprising, given that $I'$ triggers the bug along an execution path $\phi'$ that does not fully overlap $\phi$.

It is important to highlight that, in order to achieve an actual enhancement of the user privacy, it has to be ensured that, given $P'$ (which can be automatically reconstructed from $I'$), it is impossible to identify which was the path condition $P$ associated with the original execution path $\phi$. Note that this is not guaranteed a priori since, in order not to unnecessarily constrain the search heuristics design space, we have not assumed their non-reversibility. This issue can, however, be circumvented by simply performing a logical disjunction between $P$ and $P'$, and using the resulting path condition $P \cup P'$ to determine a fault-inducing input $I^*$. 
This is, in fact, sufficient to make it impossible to distinguish whether the input $I^*$ was associated with the original or the alternative execution path. Therefore, from an information theoretical perspective, the increase in obfuscation (measured in terms of leakage) achieved thanks to the discovery of $\phi'$ during the residue minimization phase can be quantified in terms of the increase of the number of input satisfying $P \cup P''$ with respect to the number of solutions satisfying $P$. But, as already mentioned, REAP aims to achieve the lower bound on information leakage achievable given a failure $f$, i.e. the leakage achievable by randomly selecting an input from the set, $I_f$, containing every input triggering $f$ (this will be further discussed in Sec. 3.3).

**Leakage Minimization phase.** To this end, REAP executes a second, so called, leakage minimization (LM) phase. In this phase, REAP performs a final symbolic execution that, starting from $\phi'$, relies on a non-reversible algorithm to identify a second failure-inducing execution path $\phi''$. Our search heuristics’ framework is sufficiently flexible to support the generation of non-reversible algorithms by simply picking random values for the parameters that control the heuristics’ behavior. Given the randomized nature of this search algorithm, the alternative failure-inducing execution path, $\phi''$, it identifies results indistinguishable from any other execution path in $\Phi_f$. Consequently also the input generated using $\phi''$ (i.e. computed as a random solution of $\phi''$) results indistinguishable from any other failure-inducing input in $I_f$. This allows REAP to achieve the theoretical lower bound on the information leakage for reproducing a given error.

**Why not using exclusively the LM phase?** Note that if one applied the RM phase directly to the original execution path $\phi$, the alternative failure-inducing execution path $\phi^*$ would still result indistinguishable from other execution path in $\Phi_f$, hence achieving already the lower bound on leakage. However, the search heuristic employed in the LM phase, being of randomized nature, does not attempt to directly maximize the dissimilarity (or, equivalently, minimize the residue) between the alternative failure-inducing input it identifies and the original input.

This consideration is at the basis of the REAP’s design choice (illustrated also by the diagram in Figure 4) of executing the LM phase starting from an alternative failure-inducing execution path $\phi'$ that has been previously obtained using a search heuristic conceived to minimize residue. As it will be also shown in Sec. 4, if we skipped the execution of the RM phase (or if it fails), and applied the random algorithm of the LM phase directly on the original execution path $\phi$, we would risk identifying an alternative execution path closely correlated with $\phi$, and hence having significantly higher residue values. Conversely, by applying the LM phase starting from the less correlated execution path $\phi'$ determined by the RM phase, REAP aims to maximize the chance to obtain an alternative path $\phi''$ that optimizes not only leakage but also residue.

**Privacy evaluation and report submission.** Once $\phi''$ is obtained, REAP determines a feasible value for the input $I''$ that triggers the execution path $\phi''$ by finding a solution to the corresponding path condition $P''$. Further, REAP computes the residue associated with $I''$, and derives a conservative lower bound for the attained leakage level (more details in Section 3.3). Finally, the user is presented with the anonymized input, along with the corresponding leakage and residue values, and is asked to authorize the transmission of the bug report to the maintenance server.

### 3.2 Search heuristics framework

As we have already mentioned, the algorithms employed by REAP are based on the idea of searching for alternative failure-inducing execution paths by performing deviations of bounded length from the original faulty execution path $\phi$ (that we also call detours). Before detailing the algorithms, we introduce how an execution path is modeled in REAP.

REAP associates with the execution path $\phi$ of a program a directed acyclic graph, denoted as $G(\phi)$, where each node of the graph represents a sequence of statements comprised between two subsequent conditional tests on some input-dependent variable. The graph is built dynamically, during the symbolic execution of $\phi$, adding a node to the graph (and
connecting it to the previously generated node) every time that a branch of an input-dependent test is taken. Whenever a node is added to the graph, this is also labelled using the following triple: a location identifier composed by line of code and class signature; the current stack trace; the value of the current iteration of any cycle within which the node is being executed. This simple scheme allows us to avoid aliasing problems, ensuring that if a program statement is executed in two different execution contexts, two unique identifiers will be attributed to it.

Search Heuristics. We now introduce the framework used to generate the search heuristics employed by REAP. The framework is embodied by the function $\phi$SEEKER, whose pseudo-code is shown Figure 5. This function encapsulates the logic of a generic search heuristic that, given an execution path $\phi$ and a fault $f$, returns a pair in the domain $[\text{boolean} \times \text{PathCondition}]$, where the returned boolean is true if a different execution path $\phi'$ causing $f$ has been identified. In this case $\phi$SEEKER also returns the PathCondition of $\phi'$. Otherwise, if $f$ is not reproduced, or if $f$ is reproduced along an execution path identical to $\phi$, the pair $<\text{false},\perp>$ is returned.

The behavior of the search $\phi$SEEKER is flexibly customizable by means of the following input parameters:

- $\text{numDetours}$: the total number of detours that the search heuristic should attempt.
- $\text{maxDetourLength}$: the maximum number of nodes that the search heuristic can traverse after having performed a detour and before joining back the original path.
- $\text{Sorter}$: an external function that is used to order the set of nodes of $G(\phi)$ and select the ones for attempting a detour. As we will see, the ability to provide $\phi$SEEKER with arbitrary sorting functions via the Sorter parameter allows supporting, in a modular way, the design of both informed and non-informed algorithms.

The first two parameters allow exploring various trade-offs between cost (execution time) and coverage of the search for alternative failure-inducing execution costs. The semantics of the maxDetourLength can be better understood by analyzing the pseudo-code of Figure 5, according to which, whenever a node $n$ of $G(\phi)$, $\phi$SEEKER activates a Randomized Bounded Depth First Search (RBDFS) algorithm using a depth bound value equal to $\text{maxDetourLength}$. RBDFS algorithm guarantees that the exploration of alternative execution paths is aborted if this does not re-join the original path before $\text{maxDetourLength}$ hops. In this case the detour is aborted and the execution is restored at $n$, which is then executed along the original path. Hence, the larger the values of $\text{maxDetourLength}$ and $\text{numDetours}$, the wider the search space explored by the algorithm (and, consequently, its execution time). The RBDFS algorithm is a variant of classic BFS scheme, in which, whenever a node is visited, its descendants are inserted in the BFS’s stack in random order. This is necessary since we want to guarantee, in the LM phase, that every execution path in $\Phi_f$ can be obtained with a non-null probability.

Note that, in order to be able to backtrack the execution of detours, REAP relies on the built-in supports provided by Java Path Finder [21], the symbolic execution engine that we integrated in REAP, in order to transparently take/store checkpoints of the current application execution’s state (more details on the implementation of the REAP prototype can be found in Sec. 3.4).

Figure 5. Pseudocode defining the family of algorithms used to identify alternative failure-inducing execution paths.

```plaintext
[boolean, PathCondition] $\phi$SEEKER (ExecutionPath $G(\phi)$, Fault $f$, int $\text{maxDetourLength}$, int $\text{numDetours}$, SortingFunction $\text{Sorter}$)

Set $<\text{Node}>$ $\text{detours}$;
PathCondition $P$:
sort the nodes of $G(\phi)$ using $\text{Sorter}$, and store the resulting top $\text{numDetours}$ in $\text{detours}$;

for each node $n \in G(\phi)$ do
if ($n \in \text{detours}$) // $n$ is selected for detouring
    take a checkpoint before executing $n$;
    for each node $n'$ reachable from $n$ by taking a different branch from the one in the original execution path $G(\phi)$;
    visit the subtree rooted at $n'$ using Bounded Depth First Search and setting the bound on the depth at $\text{maxDetourLength}$;
    for each visited node $n$:
        if ($n^* \in G(\phi)$) // the detour re-joined $G(\phi)$
            add to $P$ the logical constraint of the curr. detour;
            exit from (R)BDFS successfully;
            execute node $n^*$;
            if (currently visited node reproduces $f$)
                add to $P$ the logical constraint of the curr. detour;
                return $\text{true},P$;
        if ((R)BDFS exited unsuccessfully) // failed to re-join $G(\phi)$
            backtrack to last checkpoint;
            execute $n$ along the same branch taken in the original execution path $G(\phi)$;
            add to $P$ the logical constraint of the branch taken;
        else // $n$ is not selected for a detour
            execute $n$ along the same branch taken in the original execution path $G(\phi)$;
            add to $P$ the logical constraint of the branch taken;
            if ($n$ reproduces $f$) do
                if (current execution path $\neq G(\phi)$)
                    return $\text{true}, \text{current path condition}$;
                else // $f$ manifested along the original execution path
                    return $\text{false}, \perp$;
        else // the execution bypassed the node where fault $f$ originally took place
            return $\text{false}, \perp$.
```

```
```

Framework usage in the RM and LM phases As already mentioned, both the RM and LM phases rely on the pre-

---

2 As in previous related works [8, 9, 14], we assume that faults are observable and uniquely identifiable.
sented search heuristic framework to identify alternative failure-inducing execution paths with different properties.

As the RM phase aims at maximizing the dissimilarity (number of shared bits) between the original and alternative inputs, the input parameters of the $\phi_{\text{SEEKER}}$ function are selected to maximize the number of detours from the original execution path. The rationale underlying this choice is that the higher the number of detours, the higher the chances of maximizing the dissimilarity between the path condition (and hence corresponding input values) of the alternative and original failure-inducing execution paths. Clearly, as performing a detour has a cost, it may be desirable to limit the maximum number of detours, which may possibly result in suboptimal residue values. We evaluate this trade-off in Section 4, where we assess the effects of the choice of $\text{maxDetourLength}$ and $\text{numDetours}$ on the effectiveness of the RM phase. Concerning the selection of the Sorter function, in the RM phase we use an implementation that orders the nodes of $G(\phi)$ based on the restrictiveness of its associated input-dependent conditional test. More precisely, this sorting function orders every node $n$ in $G(\phi)$ on the basis of the residue of the obfuscated input $I^*$ that is obtained by solving the input-dependent test associated with $n$. This sorting function allows deriving an informed, deterministic algorithm that, as it will be shown in Section 4, is particularly effective in identifying alternative execution paths having low residue values, and is therefore particularly attractive during REAP’s RM phase.

In the LM phase, we need to guarantee that the failure-inducing execution path is indistinguishable from any other failure-inducing execution path in $\Phi_f$. To this end, it is sufficient to select random values for the input parameters\(^3\) of $\phi_{\text{SEEKER}}$, as this allows for the generation of any alternative execution path of the application.

**What if the failure is not reproduced?** Given its heuristic nature, the search of alternative failure-inducing execution paths may be unsuccessful during either the RM or the LM phase. Note that we consider a search unsuccessful either if the execution of $\phi_{\text{SEEKER}}$ fails during the RM phase may end up being “cancelled”, increasing the overlap up. In this case REAP does not achieve the theoretical lower bound on information leakage, and outputs the path condition $\phi'$ that was generated by the RM phase.

### 3.3 Privacy

The privacy (in terms of both residue and leakage) provided by REAP has a lower bound, which coincides with that achieved considering the path condition of the original execution ($\phi$), as in Castro et al. and Camouflage. This situation arises in the case where REAP is unable to find any alternative failure-inducing execution path.

Whenever REAP is able to successfully complete the residue minimization phase, it achieves a reduction of both residue and information leakage. Residue is reduced as, whenever a successful detour is found, REAP determines alternative failure-inducing path conditions, and hence different failure-inducing user inputs. Also, the identification of a new failure-inducing path condition $\phi''$, allows REAP to draw new input values from a larger domain, defined by the unions of the inputs corresponding to the original path condition $\phi$ and by $\phi''$: this leads to a corresponding decrease of the information leakage.

Additional privacy is provided by REAP whenever the LM phase of the algorithm is successfully executed, resulting in the identification of a second failure-inducing path condition $\phi''$. This path condition has several important characteristics: as it was obtained using a randomized, irreversible algorithm that may generate any execution path in $\Phi_f$, it achieves the information leakage computable by considering the logical disjunction of the path conditions of all execution paths in $\Phi_f$ (i.e., the lower bound on information leakage for reproducing $f$).

However, computing the exact leakage provided by the union of all possible alternative paths is impractical for most real applications, as it would force to identify exactly the set $\Phi_f$ (which is what is done in MPP [14], and represents its greatest limitation). To circumvent these issues, REAP provides a conservative upper bound of the actual information leakage, by computing the information leakage of the disjunction of the path conditions output by the OIA, RM and LM phase, i.e., $\phi \cup \phi' \cup \phi''$. This is expected to be, in most cases, significantly higher than the real level of privacy provided but has, on the other hand, the advantage of being efficiently computable.

Finally, the alternative input ultimately generated by REAP is obtained by determining a random solution of the path condition $\phi''$ identified during the LM phase. It is interesting to highlight, as we will discuss also in the evaluation section, that the residue of the alternative input generated by the LM phase may be higher than the residue of the alternative input generated by the RM phase. During the LM phase, in fact, some of the detours performed during the RM phase may end up being “cancelled”, increasing the overlap between the execution path $\phi''$ and $\phi$ (with respect to $\phi'$). Nevertheless, as confirmed by our experimental data, the re-

\(^3\)Concerning Sorter, this is selected to return a random subset of the nodes in $G(\phi)$. 
duction of residue at the end of the LM phase, can extend 73% on average and up to the 85%.

3.4 Prototype Implementation
We implemented REAP for applications written in the Java language. This tool has three main components: the execution monitor, the symbolic execution engine and the anonymizer. The execution monitor instruments the compiled Java application using the SOOT [22] bytecode instrumentation tool in order to log all user input in a transparent fashion. Note that, in order to ensure deterministic error replay, one should log all sources of non-determinism of the program, and not solely user input. On the other hand, dealing with other sources of non-determinism is out of the scope of the REAP system for the following two main reasons: i) different types of non-deterministic sources could be tackled using dedicated solutions aimed at supporting deterministic replay [23, 24]; ii) from the privacy perspective, which represents the focus of our work, user inputs are arguably the most critical sources of non-determinism.

The symbolic execution engine is probably one of the most crucial components of REAP. REAP uses Java Pathfinder [12, 21] (JPF) for this purpose. By default, all variables that are affected by the execution of read calls of the java.io library are assumed to be user input and are therefore marked as symbolic. Our anonymization tool is implemented in Java and uses JPF’s constraint solving implementation to obtain new input from the path condition. The JPF solving implementation bridges JPF to the actual solver, which can be specified as a parameter. JPF’s constraint solving implementation supports several constraint solvers, but in our work we used z3 [18].

4. Evaluation
This section aims at evaluating both the anonymization quality and the scalability of REAP. We start by presenting the set of applications employed in our study, and then report the results of our experimental evaluation in which we evaluate the performance of REAP across its various execution phases. The experimental platform used in this study is a machine running the OSX Lion operating system, equipped with a 2.8 GHz Intel Core 2 Duo processor and 4 GB of memory.

4.1 Subjects
REAP was evaluated using six different applications. This evaluation suite comprises a mix of real and artificial bugs and of applications of diverse nature. Two of these applications were run with two different inputs and therefore this evaluation encompasses a total of eight different scenarios. The applications used in this evaluation and the respective bugs are described below. This description is strictly oriented to the aspects that are more relevant for our evaluation.

**Columba.** The Columba test case was already used in the work that introduced Camouflage [8]. Columba is an email client that contains a fault in its address book component. The contacts are stored in a csv file and if the content does not meet the expected format, Columba crashes while loading the file.

**Apache Xerces.** Xerces is a popular and large application for parsing and manipulating XML files. In this particular version of Xerces (1.4.2) there is a bug that causes a NullPointerException to be thrown when using external unparsed entities. In our experiments we used failure inducing inputs provided in Xerces bug repository.

**iDate.** iDate is an application for mobile devices that finds people matching a profile specified by the user, according to age, sex, height, amongst other information. This application crashes when users use different versions of this application, because they differ in the representation of the input values. More details about this test case can be found in [14]. For these experiments, iDate was adapted locally to run on a desktop computer.

**Painpai.** Painpai is an application from Sourceforge that provides facilities for managing finance information. To the best of our knowledge this application does not contain any bugs. However it is a very good example of a program that deals with very sensitive information, like bank account numbers and other personal information about the user. Therefore we manually injected a fault, that throws a NullPointerException at the end of the program execution.

**Cruise Control.** Cruise Control is a program that simulates the engine of a car and is available in the SIR [25] repository. We consider the mutant JDT-1, which contains a fault that causes failure on the insertion of the command on if the command engineOn was previously inserted. From this application we generated two test cases that differ only on the size of the input by 7×.

**Apache Commons CLI.** The Apache Commons CLI library provides an API for parsing command line options passed to programs. The bug considered in this evaluation can be found in the bug report 71 in the Apache JIRA repository. The problem consists on the parser treating arguments as commands in case of syntax similarities. For our experiments we used the failure inducing input provided in CLI svn repository (which we denote as CLI-Small), as well as a second test case, where we increase the input and also the number of options available (which we denote as CLI-BIG).

4.2 Residue Minimization Phase
In this subset of experiments, we aim to evaluate the performance of the Residue Minimization phase. In particular, we

---

4 We plan to open-source the REAP prototype soon.
focus our analysis on assessing the impact of the tuning of the parameters that determine the behaviour of SEEKER, namely maxDetourLength, numDetours and Sorter.

4.2.1 On the impact of maxDetourLength

In the first part of our evaluation we aim to understand what is the influence of the maxDetourLength parameter. Therefore we start by fixing the value of numDetours to the maximum value possible for each test case and run the experiments with growing sequential values for maxDetourLength. The goal is to determine to what extent larger values of maxDetourLength result in better privacy. The cost imposed is also evaluated. Figure 6 shows the residue obtained for each application while incrementing the value of maxDetourLength. We can see that the residue values do not vary with the growth of the detour length. This indicates that, in the considered applications, the detours succeeded, most of the times, in re-joining the original execution path within 2 hops. This implies that, during the performed detours, very seldomly were sequences of conditional tests (or loops) encountered while performing a detour. At the light of these results, which highlight that, at least for the considered test cases, there are no significant privacy gains in increasing the growth of maxDetourLength, we tuned REAP by setting, during the RM phase, maxDetourLength equal to 2. This has the significant benefit of narrowing considerably the space explored by the search heuristic, increasing the inherent scalability of the system.

Cost. Table 1 presents the overhead imposed by the growth of maxDetourLength. Note that, for each run, the total execution time is the sum of the value in the OIA phase column with the respective percentage. As expected, there is a growth in the execution time associated with the increment of maxDetourLength in the majority of the tests.

Figure 6 also demonstrate that the Residue Minimization phase of REAP is capable of decreasing massively the residue when compared with the OIA phase, which is an indication that most detour attempts are successful. It is also an indication that the test applications perform several restrictive logical tests, which force the leakage of significant portion of the user input (some of which are exemplified by the code excerpts reported in . Figure 7).

Why residue does not drop to zero? The residue is seldom reduced to nothing. Whenever an input value is anonymized it may occur that the new solution has some bytes in common with the anonymized value. These bytes are accounted as residue nonetheless. Below we exemplify this phenomenon by showing a similarity between the syntax of the existing commands in the Cruise control test case.

\[
\text{I} : \text{engineOn accelerator brake resume engineOff}
\]
\[
\text{I'} : \text{engineOff resume off accelerator engineOn}
\]

Also pointed out in [8], sometimes structural bytes need to remain unchanged in order for the failure to occur, like for example space characters that separate the options and arguments of a CLI input. In the Xerces test case REAP was not able to reduce the residue below 52%. The reason for this is that many xml terms like Entity or System need to be disclosed if the failure is to be reproduced and therefore REAP does not anonymize some parts of the input file. Nevertheless these parts are merely xml structural terms and do not reveal sensitive information about the user as depicted in Figure 8, an example provided in Xerces bug tracker.

![Figure 6. Plot showing the impact of the detour length on the residue](image)

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time [ms]</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OIA phase</td>
<td>2</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>Columba</td>
<td>320562</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columba</td>
<td>0.7</td>
</tr>
<tr>
<td>Xerces</td>
<td>79.5</td>
</tr>
<tr>
<td>iDate</td>
<td>9.3</td>
</tr>
<tr>
<td>Painpal</td>
<td>64.0</td>
</tr>
<tr>
<td>CC-Big</td>
<td>32.0</td>
</tr>
<tr>
<td>CC-Small</td>
<td>23.3</td>
</tr>
<tr>
<td>CLI-Big</td>
<td>44.4</td>
</tr>
<tr>
<td>CLI-Small</td>
<td>1916</td>
</tr>
</tbody>
</table>

Table 1. Overhead imposed by maxDetourLength
In this subset of experiments we aim to evaluate the other two main parameters of REAP, `numDetours` and `Sorter`. Knowing that there are no privacy gains having larger values of `maxDetourLength`, we fixed the value of `maxDetourLength` to 2. In order to evaluate `numDetours` we run experiments increasing sequentially its value up to the maximum possible for each test case. At the same time we evaluate the effects of using, as `Sorter` function, the informed sorting function described in Sec. 3.2 that orders every node of $G(\phi)$ using the residue of the obfuscated input, with a sorting function that simply uses the natural order of the nodes in the program. The results are presented in Figure 9. We considered the test cases Cruise Control in Figure 9(a) and CLI in Figure 9(b). We present the results solely for these two test cases, for space limitations and because the other test cases do not present any relevant differences.

The sorting function based on residue is labeled as informed. In Figure 9, and the one compliant with the program execution order as uninformed. We can see in both figures that both sorting functions end up achieving the same value for residue if we increase arbitrarily `numDetours`, but the informed sorting achieves target residue levels faster than the non-informed one. This is particularly clear for the case of CLI (Figure 9(b)), highlighting that the usage of informed sorting function may sometimes play a relevant role in the performance of REAP, and opening new research directions aimed at identifying alternative informed heuristics for prioritizing the selection of detouring points.

Further, the experimental data confirms the intuition that, in order to minimize the residue, it is beneficial to maximize the number of detours. In fact, if every conditional test (on input) in an application is as restrictive as exemplified in Figure 7 (which is the case of Cruise control), then the minimum residue possible can only be achieved by detouring at every node.

### 4.3 Leak Minimization Phase

In these experiments we measure the privacy gains of applying the leak minimization phase. Figure 10 presents the leakage of the union of the three path conditions given by $\phi \cup \phi' \cup \phi''$ and also the leakage of the the union of all the path conditions that lead to the failure, given by $\Phi_f$. The leakage of $\phi \cup \phi' \cup \phi''$ is automatically computed by REAP after the LM-phase is complete. However, as the problem of computing the leak of $\Phi_f$ is, in general, intractable, for this evaluation, the leakage values of $\Phi_f$ were calculated via a manual code inspection. In all the test cases the cardinality of $\Phi_f$ is sufficiently large to make the difference between $\phi$ and $\phi \cup \phi' \cup \phi''$ quite small, comprised between 0.07% and 0.96% for these test cases, and for this reason the corresponding bars in Figure 10 would be indistinguishable. Therefore Figure 10 does not include the leakage of $\phi$. The results of REAP presented in Figure 10 are the average of 10 runs, due to the random nature of the LM phase.

The results demonstrate that the leak of $\phi \cup \phi' \cup \phi''$ is quite large for most of our test cases. This confirms the restrictiveness of many logical tests performed in most of test cases, as illustrated in Figure 7. On the other hand, with the exception of the Xerces test, the leakage of $\Phi_f$ drops to values comprised between 0.005% and 20.45%. On average, the information leakage associated with $\Phi_f$ was of 86.1%, achieving a 57.6% average reduction with respect to the OIA phase. The leakage of $\Phi_f$ can go up to 20.45% for CLI and Cruise control, because they require the presence of specific tokens in order to reproduce $f$, for example, the Cruise control failure is reproduced only if the user inputs the commands `engineOn` and `on` in sequence.

The leakage of $\phi \cup \phi' \cup \phi''$ is specially high for the Painpai and Cruise-control (over 99% for both) subjects because all the logical tests performed in these programs restrict the symbolic variables to specific values (see the code excerpt in Figure 7). Like in the Xerces test case (see Figure 8), in the Columba test case the leakage and residue are caused by the structural bytes in the csv file, which, in this case, are the commas. Therefore the OIA phase alone is able to anonymize all sensitive information in the csv file.
Figure 9. Informed VS Uninformed

Figure 10. Bar charts showing the amount of bits revealed.

Figure 11. Bar charts showing the amount of residue.

Table 2. Execution time of OIA, REAP and MPP (in ms).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>OIA phase</th>
<th>REAP</th>
<th>MPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columba</td>
<td>320562</td>
<td>322824 (0.7%)</td>
<td>x</td>
</tr>
<tr>
<td>Xerces</td>
<td>22944</td>
<td>85346 (272.0%)</td>
<td>x</td>
</tr>
<tr>
<td>iDate</td>
<td>1275</td>
<td>1804 (41.5%)</td>
<td>17197</td>
</tr>
<tr>
<td>Painpai</td>
<td>3084</td>
<td>13811 (463.5%)</td>
<td>x</td>
</tr>
<tr>
<td>CC-Big</td>
<td>11590</td>
<td>75301 (549.7%)</td>
<td>x</td>
</tr>
<tr>
<td>CC-Small</td>
<td>2794</td>
<td>11476 (310.8%)</td>
<td>x</td>
</tr>
<tr>
<td>CLI-Big</td>
<td>8194</td>
<td>11516 (40.5%)</td>
<td>12082</td>
</tr>
<tr>
<td>CLI-Small</td>
<td>1916</td>
<td>3199 (67.0%)</td>
<td>6886</td>
</tr>
</tbody>
</table>

Figure 11 presents the residue results of this part of the evaluation. It also includes the residue results of REAP without applying the residue minimization phase which is labeled as LM phase. The results of LM phase are also the average of 10 runs. With this additional comparison we intend to demonstrate the usefulness of applying all stages of REAP. We can see in Figure 11 that in many of the test cases it pays to use both the RM and LM phase, as they result in lower residue. The same cannot be said about the test cases of Columba, Xerces and iDate. Nevertheless note that for the Columba and Xerces test cases, the residue is caused by language specific terms (for example see Figure 8) and the Residue Minimization phase cannot improve this.

4.4 Scalability

Table 2, gives a complete notion of the overhead of REAP when compared with the single phase process of obfuscating using only the original execution path. In addition, we implemented and run MPP. Note that, in several of the test cases MPP failed in identifying alternative failure-inducing inputs, as it either depleted all available memory or did not complete its execution 24h. These cases are marked with an x in Table 2. Results show that REAP takes, at most, a few minutes to finish. This is, in practice, perfectly admissible, especially if one consider that REAP can run as a background task executing in idle periods.

The address book component of Columba expects each contact of the file to be in a separate line, and for each contact the information is separated by commas. It uses a loop to iterate each line of the file and processes each line, using another loop, before it gets the next one. The OIA phase requires each line to be processed the same way as in the original execution, which took 320562ms to process the input file provided (we used the input files provided by the authors of [8]). Considering that a loop is also a logical test, the RM and/or LM phase of REAP attempt a detour which means that it attempt to skip the loops in the early stages of the iteration. This is followed by Columba trying to to save a contact with an unexpected format. This resulted in reproducing the same failure very quickly with little information processed, and therefore it imposes less than 1% of additional execution time. The alternative input generated was considerably smaller than the original one, and reproduces the same failure.
5. Conclusions, Limitations and Future Work

This paper presented REAP, a system that tackles the issue of user privacy in error reporting. It demonstrates that mechanisms similar to the OIA phase may reveal sensitive information if the target application performs restrictive logical tests on the input. REAP advances the state of the art by significantly increasing privacy through the exploration of alternative execution paths. Although it was visible that REAP’s testing of additional execution paths come with a cost most of the times, we should consider that REAP is designed to be executed offline when resources are available (e.g. overnight) and in the experiments conducted REAP took a few minutes at most, even in applications where MPP did not scale.

The leak minimization phase of REAP forces an opponent to consider all possible execution paths that induce the same failure. This provides better privacy but, on the other hand, does not allow REAP to exactly estimate the information leakage of the obfuscated failure-inducing input. Devising scalable techniques for more closely estimating the leakage achieved by REAP is an interesting research direction opened by this paper, which we intend to pursue in our future work.

References

[21] : Java pathfinder: http://babelfish.arc.nasa.gov/trac/jpf
8 Appendix B – Technical Report on GUI Anonymization
GAUDI: Graphical Anonymizer for User Domain Input

Abstract—Software error reports are currently insufficient both in terms of error reproducibility and user privacy. GAUDI is a new system which is capable of monitoring a GUI application, logging its execution and calculating the minimum subset of that execution’s input needed to reproduce the observed fault. GAUDI reduces the number of graphical interactions in an error report by an average of 82.2% and can be mixed with other error report anonymization techniques to further improve user privacy without compromising reproducibility.

Keywords—Anonymization; Privacy; GUI; Deterministic Replay;

I. INTRODUCTION

Increasingly software applications are released with errors, mainly due to the fact that completely test an application is a time consuming, expensive task and sometimes even impossible because of the complexity of the system. Some studies estimate that testing can consume fifty percent, or even more, of the development costs[1]. As a result, software vendors have to correct errors after their applications have been released. To achieve this, developers make use of bug report systems, which provide information to developers on how to further improve their products. However, most bug reports do not provide useful information needed to debug an application[2].

In these kinds of solutions an error report is created and sent to the debugging team when a crash occurs in the user machine. The report usually contains information about the state of the environment in which the error occurred. One of the most widely used error reporting tools is Microsoft’s Windows Error Reporting (WER)[4] which gathers information from a huge amount of users all over the world1. Essentially WER is a tool that, when it detects a crash, records a core dump and sends it to Microsoft’s servers upon the user’s consent. Afterwards, that information is analyzed by a debugging team in order to further understand the error.

One of the greatest disadvantage of this method is that it raises several privacy problems since there is no effort being made to prevent the disclosure of sensitive user information, e.g. a credit card number or a password2. Therefore, users often choose not to send the report since they do not know which information will in fact be revealed[5]. Moreover, as the report only contains information about the final state in which the error was detected, opposite to containing information about the execution that led to the error, finding the cause of the problem may turn itself to be a hard and complex task[6].

Another argument to take in consideration, is the way programmers comprehend software in order to debug a given application. Usually, developers start the debugging process by putting themselves in the role of the end-user and interacting the graphical user interface. In this way the developer is able to capture the user intent when the error was triggered and discover important information about the cause of the error, maybe locating some starting points to analyze the source code[14].

In order to enhance error reporting tools, several record and replay systems that try to deterministically replay a faulty execution have been developed [16], [3], [17], [4], [5], [10]. These systems work in two phases: the record phase in which the information needed to reproduce the error is recorded in a trace file, and the replay phase in which the information recorded is fed to the application so that the replayed execution is the same as the recorded one. To deterministically replay a crash, the trace files should include all the relevant sources of non-determinism that made the software fail[4]. In other words, all variables whose values may make two executions different. This enrichment of error reports has only worsened already existing privacy issues. If, in systems like WER, users often choose not to send the report since they do not know whether private data input into the application is contained in the error report [5], in record/replay systems the privacy concerns are even bigger since all user input data may be included.

Since nowadays most software applications are driven by the interaction between users and a Graphical User Interface (GUI)[11], user’s graphical input is one of the most relevant sources of non-determinism and therefore of bugs[9]. Despite the fact that the GUI is one of the main sources of non-deterministic inputs in addition to a generalized lack of privacy in bug reporting, there are no real solutions that address both problems simultaneously.

We propose an error reporting tool, GAUDI, that anonymizes execution traces by reducing recorded traces to the minimum sequence of events needed to trigger the observed error. Our approach is based on the fact that one can actually make use of a graphical execution to provide a better anonymization technique and a more intuitive way of comprehending the error. This paper presents some related systems (Sec II), GAUDI’s architecture (Sec.III) and its evaluation (Sec.IV) before concluding in Sec. V.

---

II. RELATED WORK

There are various approaches to correct bugs in a program, either through testing or through allowing debugging after the release of the software. We will focus on a specific subset of solutions, which use a technique called deterministic replay. Deterministic replay is a technique concerned with the reproduction of bugs, in particular those raised by non-determinism. To address non-determinism, this technique works in two phases: the record phase and the replay phase. During the first phase all relevant deterministic events are recorded into a trace file. Then, on the second phase, the trace file is used to replay the non-determinism events that were previously recorded, thus, enabling the replay of the error whenever needed. The sources of non-determinism can be divided in two sets, input non-determinism, and memory non-determinism [13]. Input non-determinism encompasses all inputs that are consumed by the system being recorded, which are not generated within the layer where the system is running, such as system calls, keyboard and network inputs, etc. In turn, memory non-determinism is created by the order in which the threads access the shared memory of a given process. This order may be different due to several differences in the overall state of the system.

When using a deterministic replay system for remote debugging, several security and privacy problems arise [3], [17], [4], [5]. For instance, the bug report may contain passwords, addresses and credit card numbers. Some systems have been developed to address this problem in console applications [4], [5]. These solutions focus on finding an alternative input, which originates the same crash as the original input but is otherwise unrelated. This is done by analyzing the original path and computing the path conditions. The path conditions are logic constraints that describe which path was taken in every branching instruction of the original execution. Finally, the path conditions are fed into a solver in order to find an alternative input. Other systems [10] increase the set of possible alternative solutions by also searching different paths from the original execution.

However, there are no solutions that provide a correct deterministic anonymized replay for GUI applications. Therefore, we developed a deterministic replay system that focus on anonymizing the graphical data generated by an user.

III. GAUDI

GAUDI is a system for the anonymization of error reports, without the need to recompile applications or change the source code. The system is divided into a client and a server. The client runs in the end-user machine and monitors a transformed version of the target application. When a faulty execution is detected, a log is generated and anonymized. Finally, the anonymized trace is sent to the server owned by the application’s maintenance team. All these actions are executed in the background so that the normal behavior of the target application is not disrupted. The server saves the anonymized logs, which can then be replayed by the maintenance team. Furthermore, the server also provides the tools to transform the original applications so that GAUDI can monitor them.

Most modern graphical frameworks work by converting user interactions into code invocations. When a user performs a certain action, an event is triggered in the GUI. If that event has any registered listeners, it will consequently cause the listener to be invoked. Ultimately, this will execute the application code and change the state of the program. This means that an execution can be described by the listeners, and consequently so can an error.

Moreover, in any given application the number of events triggered by a user is always greater than the number of listeners being invoked in the source code[11], [7], [8]. Therefore, we designed GAUDI to monitor which listeners are triggered during a user execution. In this way, the system is able to record only relevant changes to the logic layer while still being able to perceive the relevant graphical interactions made by the user. However, the listeners alone are not enough to completely reproduce the execution. This is because of the events, which despite not triggering any listener, change the state of the GUI (e.g. writing in a text box). As a result, GAUDI needs to save these changes to the global state of the application as preconditions to the listener that is going to be executed.

Figure 1. Example of a user interaction, the events that were triggered and the listeners invoked.

Therefore, we designed GAUDI to monitor which listeners are triggered during a user execution. In this way, the system is able to record only the relevant changes to the logic layer while still being able to perceive the relevant graphical interactions made by the user. However, the listeners alone are not enough to completely reproduce the execution. This is because of the events, which despite not triggering any listener, change the state of the application. As a result, GAUDI needs to save these changes to the state of the application as preconditions to the listener that is going to be executed. The system generates these preconditions by monitoring read commands made to graphical variables during a listener invocation. So, if during a listener call, a read command is done to some specific values from the GUI, then those values need to be available during the replay. When the listener is replayed, these values will be the preconditions to that listener. By representing an user execution with listeners instead of events, the system reduces the size of the trace files and automatically discards irrelevant events. Moreover, in this way, related actions
are aggregated and treated as a single step, which would not happened if we used event sequences to describe an execution.

With a listener sequence, and a representation of the GUI called the Widget and Listener Graph, GAUDI can then anonymize any given execution by reducing the sequence to the minimum set needed to reproduce the error. The WLG is extracted, in the server, from the original application and provides a static map to the application structure and behavior, which can be used to infer information about the user executions.

In order to add all these functionalities to the target application without the need to recompile it, we use an instrumentation technique that will transform an application to enable the monitoring and the recording. As such, GAUDI automatically transforms any given program, by injecting new compiled code, so that the application can communicate with the client-side of the system enabling all the recording functionalities.

A. GAUDI Architecture

As it was previously referred, GAUDI works in two separate phases, a pre-deployment phase and a post-deployment phase. The system functionalities are divided in order to remove from the client as much of the computational overhead as possible. Because of this, the instrumentation and the ripping of the GUI are done in the server application. This way, the client can use pre-calculated files that provide the needed information. However, the client still needs to be entrusted with the anonymization of the log because otherwise, there would be sensitive information being sent to the server, which ultimately invalidates the whole process. Therefore, the architecture which will be described next, was designed with three main goals in mind: 1) reducing the computational overhead on the client side, 2) not disrupting the target application normal behavior, and 3) providing developers with the tools they need to inject GAUDI into their own applications.

Server

The server application is composed of four main sub-systems as we can see in figure 2: the Dynamic Widget Identifier (DWI), the Transformer, the Ripper, and the Replayer. The DWI is a system designed to create unique identifiers for widgets throughout different executions, in order to provide a correct mapping between which widgets triggered each event. Nowadays, graphical frameworks do not provide a way to identify widgets within several executions of the same applications, because that information is not needed to run the GUI. However, in order to deterministically replay a graphical execution, we need to be able to identify widgets throughout different executions. The DWI was created to be executed at runtime on top of the graphical framework, generating and managing those IDs.

Transformer: This module is responsible for applying the GAUDI instrumentation to the target application’s compiled code. In order to do this the Transformer receives the file with the entry point for the application and automatically transverses all the files of the application, analyzing each method. For each function, the module checks if it is a listener, if so the transformerinjects a call to the GAUDI Recorder in the begin and in the end of the method in order to identify the beginning and the ending of a listener call. Moreover, we also instrument all the attributions which have as a right operand a graphical widget in order to build the preconditions for the listener call. With these modifications we are able to monitor when a listener call is made and which graphical variables were read within a listener call. Finally, the Transformer generates a instrumented version of the compiled code, as seen in figure 2, which when executed with GAUDI enables recording and anonymization. This version of the target application can then be distributed to the end-users.

1) Ripper: The Ripper is in charge of extracting the WLG. This process is semi-automatic, since the program cannot find windows which are not created when the application starts. In a similar way to the Transformer, this module receives as an argument the file with the entry point for the target application and executes it. When the application is fully loaded and all the root graphical components are initiated the ripping process starts. The Ripper automatically extracts the root windows of the application and after that initiates a depth first search for the children widgets until everything is ripped. In the process every widget is attributed a unique ID using the DWI. Furthermore, the relevant properties of the widget are recorded in the WLG e.g. the name of the class that represents the widget or its listeners.

After this, the developer needs to open the remaining windows of the application in order for the process to repeat itself. This could be done automatically. However, this way we are able to create a more accurate model of
the application. If this process was done automatically then some interactions with the GUI would not be captured and the developer team would need to correct the model by hand. For example, a login interaction, a creation of a new entry, or filling a form, need to have some values, and sometimes specific ones, inserted before the application may proceed. In order to capture all the windows, we decided to shift this effort to the developer team which knows the applications and can easily open all the windows enabling the Ripper to extract all the relevant information to create a accurate WLG.

When the ripping process is completed, one just has to shutdown the application and a static WLG is saved onto a file, which can then be distributed to the end-users.

2) **Replayer:** This sub-system is entrusted with replaying an event log to the developer who is analyzing the bug. The Replayer receives a log which it uses to reproduce the execution using a non-instrumented version of the target application, thereby providing a visual aid for the developer to further understand the error. This is done by recreating the events which are present in the log and injecting them in the specific widget at which they were recorded. In almost every modern graphical framework an event is defined by a pair \((e, w)\) in which \(e\) is the event which was triggered and \(w\) is the widget where the event was triggered. As such, the task of the Replayer is to reconstruct the recorded event, identify the widget in which the event happened through the DWI (as shown in figure 3) and inject the new event in the specific widget. In the end of the replay, the original exception, included in the log, will be shown to the developer in order to confirm that it was the same error.

**Client**

The client is responsible for most of the post-deployment phase: monitoring and recording the listener sequence and its preconditions, and anonymizing the log. The client is composed by five different modules as seen in figure 3: the DWI, the Recorder, the Anonymizer, the Converter, and the Tester.

**Recorder:** The Recorder is responsible for monitoring the instrumented target application and recording a listener log. The listener log is a structure composed of the raised exception at the time of the error and the sequence of listeners and their pre-conditions recorded since the beginning of the recording. This module is initialized simultaneously with the application being monitored. After this, each time a listener starts or ends, or a graphical variable is read within a listener the Recorder is called through of the code injected by the Transformer.

When a listener is called, a structure is created in order to the identify the listener later in the WLG. After this, every read command done to a graphical variable before the listener ends is recorded as a precondition to that listener. When the call ends, the listener and its preconditions are recorded as a single step in the listener log. When a read command is invoked, the application passes, as an argument, to the Recorder the type of value which was read, the concrete value which was read and the instance of the widget where the read occurred. With this, the Recorder uses the DWI to get the widget ID, which will be recorded along with all of the previous referred information as a read access.

The other task for which this sub-system is responsible is error detection. For this, the Recorder creates a special thread which is called when an exception is raised and not caught in the target application. When this happens, the thread stops recording and saves the current sequence, and the exception that was triggered, to a file.

**Anonymizer:** After the listener log is recorded, the resulting file is fed onto the Anonymizer which will try to find an alternative graphical execution that triggers the error. For this purpose, the Anonymizer makes use of two other bundles, the Converter and the Tester. These are respectively responsible for converting a listener log into a event log and for testing if a given log produces a specific error. This module applies the Minimum-Set Listener Reduction algorithm to a given listener log with the goal of reducing it. In order to apply the algorithm, the Anonymizer uses the WLG to infer information about the GUI e.g. in which widget a given listener is located.

After an error has been detected and saved onto a file, the Anonymizer reads it and manages all the conversions and tests that need to be done while applying the algorithm. In the end, the new anonymized execution is translated into an event log and sent to the server. All the auxiliary files created in the process (the converted logs and the test hypothesis) are deleted in the end.

**Converter:** The Converter is used by the Anonymizer module in order to convert a listener sequence into an event sequence which can then be injected into the GUI of the target application. The original sequence of listeners is processed from the beginning to end. For each listener, its preconditions are analyzed and converted into a sequence of events in the end the listener itself is converted into a sequence of events. Finally all the events are added into a new sequence of events. When a listener sequence is fully converted, the resulting event sequence is recorded onto a file so that other modules can use it later.

For example, if a given listener has a precondition which states that a read operation was made from a text field widget with a given id, and the value read was a string, the converter is going to create all the necessary events to put the specific value onto the indicated widget. In this case it would be something similar to: 1) selecting the widget, and 2) typing all the characters of the string (one event per character). If the listener is registered for clicks and its placed in a widget that is a button, then the Converter knows it has to generate a click in that widget in the end.
**Tester:** This module is entrusted with receiving an event log and testing it, in order to check if it triggers an exception, and if that exception is the same as the one which happened in the original execution. The Tester injects the events in the log into the target application in the same way the Replayer does, which is, recreating the events and injecting them in a given widget that is found using the DWI. When the log is fully replayed the module compares the resulting exception, if any, with the one in the original log. If they match, then the hypothesis is marked as a valid one.

Currently the Tester only checks for the same exception, if it is not the same exception it fails the test case. One improvement that could be made, in order to optimize the testing operation, would be to generate new logs to be anonymized again if a new error is found. In this way GAUDI could be performing an automatic error search and find new errors before the user has to deal with them.

**B. Dynamic Widget Identification**

GUIs have a hierarchical structure [12], [11], [8], [15]. As such, we will generate unique identifiers based of the hierarchical structure of the GUI. Therefore, GAUDI is able to generate the same IDs throughout different executions. As such, we start at the root windows and generate a identifier for each widget in the order they are created. The algorithm then proceeds downwards finding all the children (e.g. another window opened by clicking a button) and using the parent’s id as a prefix for the children’s id. The children’s ids are generated in a sequential manner, similar to the one used for the root windows.

**C. Widget and Listener Graph**

The widget and listener graph (WLG) is a structure that contains the hierarchical structure of a graphical application and the IDs of each widget. Both, the structure of the hierarchy and IDs are generated exactly as in the DWI. However, the WLG contains other data about the GUI. At the time when the Ripper extracts each widget, it also extracts information about the widget itself e.g. the class name, the type of events that it support or the listeners that have been registered. The addition of the listeners to the model enables GAUDI to infer information about the listeners relation or their location. GAUDI builds all this information into a graph that maps the hierarchical structure of the concrete GUI, and uses auxiliary hash tables to provide a faster search within the graph.
D. Minimum-Set Listener Reduction

The core part of GAUDI is the anonymization of graphical information, in order to protect the end-users’ private information and simplify the maintenance team’s debugging efforts. We consider that every graphical interaction between the user and the GUI could potentially reveal sensitive information, and as such, instead of trying to find which information should be anonymized we will try to anonymize everything we can.

The Minimum-Set Listener Reduction algorithm is applied in two phases, the delimitation phase, and the reduction phase. In the first phase, the goal of the algorithm is to find the shortest suffix of the recorded sequence of listeners that is needed to trigger the observed error. The second phase finds out, given the suffix sequence, which listeners are unnecessary and can be removed.

In the delimitation phase the algorithm generates all the test cases with the \(n\) last listeners from the original sequence, starting with \(n = 1\) and ending when \(n\) reaches the size of the original sequence. After all the test cases are generated, they are sorted by size in ascending order, and are then tested. The first test that succeeds in replaying the observed error is the shortest suffix of the original sequence that can reproduce the error.

**Algorithm 1** Delimitation Phase Test Generation

```plaintext
oseq = original listener sequence
seqlist = ∅
newseq = ∅
for oseq.size ≠ 0 do
    newseq.addFirst(oseq.removeLast())
    seqlist.add(newseq)
end for
```

The reduction phase uses as fixed points the first and the last listeners of the suffix sequence, and then generates all the possible combinations of the listeners in between, always maintaining the order of the listeners in the original sequence. In order to generate all the possible sequences we developed a scramble algorithm, which transverses the list and for each element creates two scenarios, one in which the element is on the list, and one in which it is not. After this, each possibility recursively calls the algorithm. With this we are able to generate all combinations of listeners while still preserving the order of the original sequence. After all the sequences of the reduction phase are found, they are sorted in increasing size order and tested until a valid one is found. In the end we will have a reduced sequence that is the shortest sequence that still triggers the error.

**Algorithm 2** Reduction Phase Test Generation

```plaintext
oseq = original listener sequence
seqlist = ∅
newseq = ∅
if oseq.size > 2 then
    newseq.addLast(oseq.getFirst())
    newseq.addLast(oseq.getLast())
    Scramble(oseq, newseq, seqlist, 1)
end if
```

**Algorithm 3** Scramble Algorithm

```plaintext
function SCRAMBLE(oseq, newseq, seqlist, i)
    if i < oseq.size – 1 then
        Scramble(oseq, newseq, seqlist, i + 1)
        aux = seq
        aux.removeLast()
        aux.addLast(oseq.get(i))
        aux.addLast(oseq.getLast())
        seqlist.add(aux)
        Scramble(oseq, aux, seqlist, i + 1)
    end if
end function
```

As one can notice both phases of the algorithm rely on brute-force techniques. However, the algorithms we developed work for modern graphical user interfaces. This is because of the average size of the listener sequences, which trigger an error, is small. In order to study the size of these sequences, we sampled a set of tickets from bug repositories of real-world complex applications: Eclipse, Firefox, Thunderbird, Seamonkey and OpenOffice. All the tickets retrieved contained a set of graphical interactions which explain how to trigger the bug. With this we analyzed each application, and convert each set of instructions to their equivalent set of listeners. Figure 5 shows our results.

E. Invalid Test Removal Heuristic

We also developed a heuristic to optimize the process of testing for valid test cases, therefore mitigating the use of brute-force techniques. The goal of the heuristic is to identify invalid tests and remove them without the need to convert and test them. After each turn of generating
tests, the systems consults the WLG and verifies if all the constrains between listeners are maintained; if not the test case is deleted. This is done by consulting the graph and observing if each listener of a given sequence needs to have any specific listener executed before it can be invoked. For example, if a listener $L_3$ requires that another listener $L_1$ be executed beforehand, all test sequences that include $L_3$ without $L_1$ before it, can be discarded.

An example of this is a login window. Let’s say we have a window containing widget $W_1$ with listener $L_1$ registered to it, when that listener is invoked a new login window is created. In the new window there is another widget $W_2$ with listener $L_2$ registered to it. This means that if a sequence has listener $L_2$ but does not include before $L_1$, then the sequence is not valid since there is no way of generating an event for a window that is not created.

IV. Evaluation

Our experimental study aims to evaluate the following aspects of the system: 1) anonymization and the quality of the heuristic developed, 2) the efficiency of the recording process, and 3) the overheads of instrumenting and ripping. To evaluate these aspects, we performed and analyzed several user interactions with applications being monitored by GAUDI. With this we were able to evaluate the percentage of listeners, which are removed from the original sequence. To evaluate the reduction of the recording overhead, we compared the number of events that are processed by the system to the number of listeners that are recorded. Finally, we also assessed the impact in the anonymization process of using or not the Invalid Test Removal heuristic.

We asked 28 users to perform 8 test scenarios using applications being monitored by GAUDI. We explained to each user how to use the applications involved. The test scenarios, while giving users some freedom, lead their actions to a known bug in the test applications. Our population is constituted by 32% males and 68% females, 57% of the sample are students and from the rest of them 19% work in software development. Most of the subjects have ages between 20 and 30 years.

We used a set of 5 different applications, three developed by us to test specific complex error cases, and two real-world applications. All applications are written in Java and use Swing or AWT in their GUI. All errors, both the ones developed for the scenarios and the real ones, were chosen because they are the most common errors types in software development[14], i.e. an unhandled exception and lack of input validation.

- **Calculator**: The Calculator is characterized by having only one window and several widgets, which contain each one a listener. Therefore, we chose it to illustrate how GAUDI will perform in cases in which all actions are available from the start. In that way, we will be able to show that even in this kind of environment, in which the user can achieve an error from a large set of different combinations of action, GAUDI will always be able to find the same reduced sequence for each type of error. This application has two errors built in: one happens when the user tries to divide by zero which triggers a null pointer exception is triggered, the other happens when a user produces a result bigger than the size of the variable which holds the value.

- **MyJpass**: MyJpass was developed to test GAUDI. It is based on a existing password management software called Jpass. This application was developed to test the most standard cases of anonymization. The application possesses various windows, which enable the creation and edition of password entries as shown in figure ??.

  With this we are able to test different execution which lead to the same error, and also the complexity of having to manage different windows at the same time. This application has three errors built into it, the first one is a null pointer exception which is triggered when a user tries to delete an entry from the table without selecting anything. The second and third ones, happen when a user tries to create or delete an entry with empty fields.

- **ZooManager**: ZooManager was developed to test the worst case scenarios we found when sampling the repositories for error ticket, in other words, when the error needs a large number of listeners to happened e.g. six or seven. For this we designed an application with a great number of dependencies between listeners, and also a large number of windows and widgets. ZooManager was designed to emulate a system that manages an entire zoo, a user can create animals, create housings for the animals, feed and clean them. The user is also able to rate the Zoo, the rating will be greater depending on how many animals will be fed and houses clean. With all these, we are able to test the worst and larger sequences one can have. This application has only one error: when a user is able to achieve a perfect score the application crashes triggering an exception.

- **Lexi**: is a Java word processor. Lexi implements a GUI in Swing, with several complex and not standard widgets and listeners as shown in figure ??.

  This real world application provides a complex user environment,
and enables us to test GAUDI in a real world context. For the experimental user evaluation we used a real error, which happens when a user tries to consult the main options of the program triggering a null pointer exception.

- **Pooka**: is a real world mail client. It is designed to manage several email accounts with different preferences and configurations, as shown in figure ?? . Pooka has an added complexity when compared with the other applications we chose, because it is a network application that can send and receive emails. For Pooka we used a real error existing in the present version of the application, when a user tries to create a new email without any account being created, the application throws an exception.

The testing scenarios were developed so that the users could have some guidelines without having a detailed set of instructions, which would lead to every execution being the same, and therefore invalidating the experiment. We developed eight scenarios with the previously mentioned applications. The tasks were designed to illustrate something that a user would actually do with the given software, guiding the user through the task without being too restrictive. S1 and S2 give users a mathematical problem to solve using the Calculator. S3, S4, and S5 ask users to manage different sets of users and passwords using MyJpass. In S6, users have to manage a zoo using ZooManager. S7 consists of writing a given text into Lexi. Finally, S8 asks the user to create an email profile on Pooka, send an email, and then perform some management tasks.

A. Anonymization and Heuristic Quality

When evaluating the anonymization process, we want to measure two aspects: 1) the efficiency of the Minimum-Set Listener Reduction algorithm, and 2) the relevance of the Invalid Test Removal Heuristic when compared to a version of GAUDI which does not use it.

To evaluate the efficiency of the Minimum-Set Listener Reduction algorithm we analyzed the number of listeners of the final anonymized sequences, and compared them to the ones which were previously recorded. We consider that every listener reveals something about the user interaction and that may be sensitive information. For that reason, and in order to facilitate the developer team task of correcting the error, GAUDI reduces the listener sequence to a minimum which triggers the error. In the process, all listeners not needed to trigger the error are removed, and consequently all information input to trigger those listeners is also removed.

The results in figure 6 show that GAUDI is able to reduce the amount on graphical information revealed in average to only 17.8% of the original execution. These values do not take into account the ignored events which do not trigger listeners. In scenario six, the event sequence needed to reproduce the error has seven listeners, which explains why that scenario does not have a similar reduction to the others.

To measure the gain of using the Invalid Test Removal Heuristic we anonymized each sequence two times, one with the heuristic, and one without. Figure 7 shows the results of the comparison between the number of sequences generated, both, with and without the Invalid Test Removal Heuristic. The figure shows that the heuristic is more effective on complex cases like scenario six. On the other hand it also shows that the usage of the heuristic never increases the number of generated sequences. One other aspect we evaluated was the time the algorithm needed to find a solution. Our experiments show that the heuristic is able to reduce the time to as much as 56% in complex cases.

B. Recording Efficiency

In this section we present the results regarding the recording efficiency. The values are taken from the traces produced by all 28 test users running the 8 test scenarios. To evaluate the recording process, we measured the gain of recording the listeners when compared to recording the events. For this reason we monitored the target application in order to count the number of events that were processed. Figure ?? shows the comparison between the number of events monitored during the recording process, and the number of listeners recorded.

The results show that recording listeners removes a large amount of useless information from the trace file while providing a good grouping of only relevant events. This
<table>
<thead>
<tr>
<th>Application</th>
<th>Original #Lines</th>
<th># Added Lines</th>
<th>Lines Overhead</th>
<th>Original # Classes</th>
<th># Added Classes</th>
<th>Classes Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculator</td>
<td>723</td>
<td>40</td>
<td>6%</td>
<td>18</td>
<td>33</td>
<td>183%</td>
</tr>
<tr>
<td>MyJpass</td>
<td>479</td>
<td>24</td>
<td>5%</td>
<td>11</td>
<td>33</td>
<td>300%</td>
</tr>
<tr>
<td>ZooManager</td>
<td>1437</td>
<td>50</td>
<td>3%</td>
<td>41</td>
<td>33</td>
<td>80%</td>
</tr>
<tr>
<td>Lexi</td>
<td>11696</td>
<td>315</td>
<td>3%</td>
<td>120</td>
<td>33</td>
<td>28%</td>
</tr>
<tr>
<td>Pooka</td>
<td>99166</td>
<td>1293</td>
<td>1%</td>
<td>835</td>
<td>33</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table II
**INSTRUMENTATION CODE OVERHEAD**

<table>
<thead>
<tr>
<th>Application</th>
<th>Instrumentation</th>
<th>Ripping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>#Widgets</td>
</tr>
<tr>
<td>Calculator</td>
<td>6.4</td>
<td>23</td>
</tr>
<tr>
<td>MyJpass</td>
<td>19.0</td>
<td>72</td>
</tr>
<tr>
<td>ZooManager</td>
<td>45.4</td>
<td>475</td>
</tr>
<tr>
<td>Lexi</td>
<td>66.3</td>
<td>653</td>
</tr>
<tr>
<td>Pooka</td>
<td>86.2</td>
<td>491</td>
</tr>
</tbody>
</table>

Table III
**INSTRUMENTATION AND RIPPING RESULTS**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Events Monitored</th>
<th>#Listeners Monitored</th>
<th>#Listeners Recorded</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2797.6</td>
<td>27.6</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>3866.1</td>
<td>26.9</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>11623.9</td>
<td>13.4</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>5233.9</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>8968.9</td>
<td>15.1</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>12465.9</td>
<td>13.2</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>17728.1</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>66089.9</td>
<td>17.9</td>
<td></td>
</tr>
</tbody>
</table>

Table I
**COMPARISON BETWEEN THE AVERAGE NUMBER OF EVENTS MONITORED AND THE AVERAGE NUMBER OF RECORDED LISTENERS.**

happens because saving the listener and its preconditions, is equivalent to recording the same actions as events but with the added information of which groups of events are related. Moreover, the listeners are recorded as their IDs, which greatly reduces the space overhead of the trace files. In our experiments all traces possessed sizes between 2KB and 6KB.

**C. Instrumentation and Ripping Overhead**

Table III shows the pre-deployment phase’s evaluation results. They show that our instrumentation is very quick even for larger applications like Pooka, which contain several hundreds of classes and thousands of methods. Regarding the ripping process, even for a developer that has not worked on the application, the process is very quick. Additionally, in order to reduce the time of the ripping, these could be done while testing the GUI during development.

Table II presents the measures for code injections. GAUDI only increase the amount of statements on an average of 4%, this is a very small overhead for the total dimension of the application.

We also measured the time overheads of the injected calls to analyze how GAUDI influences the performing of the applications being monitored. For this, we measured how many nanoseconds each type of call took about 100 times. The average of results for each type of call are shown in table IV. The time overhead is very low because we use only static calls to the monitor to pass the required data.

**V. CONCLUSIONS AND FUTURE WORK**

Allowing software maintenance teams to quickly identify the causes of errors is critical. Error reports are very useful tools but their quality and likelihood of being submitted needs to be improved. The structure of GUI applications can be used to anonymize and simplify the execution traces. We presented GAUDI, a system which provides anonymization of graphical executions. GAUDI anonymizes execution traces using the Minimum-Set Listener Reduction algorithm, which shortens the listener sequence to the minimum needed to reproduce the error. In our evaluation, we showed that GAUDI is able to reduce the original listener sequences on average by 82.2%, and that using our Invalid Test Removal Heuristic we are able to manage large and complex GUI applications. The anonymization provided by GAUDI can be further enhanced by techniques such as [4], [5], [10] which calculate alternative execution paths in console applications. GAUDI provides a reduced execution trace which is easier
to further anonymize both because it is shorter and because systems like MultiPathPrivacy can then look at each listener as a sequential piece of code outside of the traditional event loop of GUI applications.

We are continuing the work presented in this paper by integrating GAUDI with anonymization techniques in line with those of Castro et al., Camouflage or MultiPathPrivacy. These techniques allow us to take the concrete values present in the preconditions of listeners and calculate anonymized values for them thereby further enhancing the results of GAUDI.

REFERENCES


