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.

I. INTRODUCTION

Software error reporting tools are needed to communicate faults from client applications to its maintenance services. Most commercial tools, e.g. Microsoft’s Windows Error Reporting (WER), provide information about the final state of the application which is often insufficient for a quick identification of the fault. Furthermore, developers often start the debugging process by putting themselves in the role of the end-user and interacting the graphical user interface in order to identify the user’s intent previous to the crash [1]. Supporting this approach requires enriching error reports with more than just final state information.

In order to enhance error reporting tools, several record and replay systems that try to deterministically replay a faulty execution have been developed [2], [3], [4], [5], [6], [7]. 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 [5]. 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 [6], 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) [8], 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. 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 [10]. 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],
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 [8], [11], [12]. 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.

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 subsystems as we can see in figure 1: 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 compiled code. The Transformer instruments each listener, and each read command from a graphical component, in order make that information recordable.
- **Ripper:** The module 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. Finally, the information extracted is recorded into a Widget and Listener Graph (WLG).
- **Replayer:** The module is entrusted with replaying an event log. 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.
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 2: the DWI, the Recorder, the Anonymizer, the Converter, and the Tester.

- **Recorder:** The module is responsible for monitoring the instrumented target application and recording a listener log. Each time a listener is called, that information is recorded sequentially along with the respective preconditions. Each listener is identified with the help of the DWI, so that later it can be manipulated using the WLG.

- **Anonymizer:** After the listener log is recorded, the resulting file is fed onto the Anonymizer which will try to find a reduced graphical execution that triggers the error. This module applies the Minimum-Set Listener Reduction algorithm to a given listener log with the goal of removing sensitive information.

- **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.

- **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.

B. Dynamic Widget Identification

GUIs have a hierarchical structure[13], [8], [12], [14]. 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.

Figure 3 shows an example of the identification attributed to a root window and all its children. One other aspect that the system has to take into account are dynamically created windows. New windows cannot be treated as root windows or otherwise it would not be possible to generate the IDs in a deterministic way, since they can be opened whenever a user wants. For this reason, we map a new created window as a children of the widget which created it, e.g. when a button is pressed a new window is opened. Finally, we designed an update system, which updates the windows that are deleted and created more than once.

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.

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 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.

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 4 shows our results.

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Tickets</th>
<th>Average Sequence Size</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>10</td>
<td>3.90</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Firefox</td>
<td>6</td>
<td>2.83</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>6</td>
<td>3.33</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Seamonkey</td>
<td>7</td>
<td>2.71</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>8</td>
<td>5.18</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>37</td>
<td>3.83</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4. Bug repository sample.

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 L3 requires that another listener L1 be executed beforehand, all test sequences that include L3 without L1 before it, can be discarded.

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 an 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 choosen because they are the most common errors types in software development\cite{1}, i.e. an unhandled exception and lack of input validation.

- **Calculator**: This application is a simple Java Calculator developed by us. The goal of this application is to show how GAUDI will behave in environments where all actions are available from the start to the user.
- **MyJpass**: This is a password manager also developed by us. The objective is to test a more application which possesses a GUI hierarchy, such as creation and edition sub-windows.
- **ZooManager**: This application was developed to test the worst case scenarios we found when sampling the repositories for error ticket. This means that the user has to perform several specific steps before triggering the error. In that way, we aim to assess whether the heuristic is able to largely reduce the sequence of listeners needed to replay the error.
- **Lexi\textsuperscript{1}**: This is a real world Java word processor, which provides a complex user environment, and enables us to test GAUDI in a real world context.
- **Pooka\textsuperscript{2}**: This is a real world Java mail client, which provides a complex GUI, in a network type environment.

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 illustrated something that a user would actually do with the given software, guiding the user through the task without being too restrictive. \textbf{S1} and S2 give users a mathematical problem to solve using the Calculator. \textbf{S3}, \textbf{S4}, and \textbf{S5} ask users to manage different sets of users and passwords using MyJpass. In \textbf{S6}, users have to manage a zoo using ZooManager. \textbf{S7} consists of writing a given text into Lexi. Finally, \textbf{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.

![Figure 5. Comparison between the number of recorded listeners and the number of listeners in the final sequences.](image5)

The results in figure 5 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.

![Figure 6. Comparison between the number of test cases generated with and without the heuristic.](image6)

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 6 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.

\textsuperscript{1}http://lexi.sourceforge.net
\textsuperscript{2}http://www.suberic.net/pooka/
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 7 shows the comparison between the number of events monitored during the recording process, and the number of listeners recorded.

![Figure 7. Comparison between the number of events monitored and the number of recorded listeners.](image)

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 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.

![Figure 8. Instrumentation and ripping results](image)

C. Instrumentation and Ripping Overhead

Figure 8 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.

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 [5], [6], [7] 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.

ACKNOWLEDGMENTS

I am grateful to J. Garcia, P. Romano, P. Louro, J. Matos, and N. Machado for the fruitful discussions, ideas, and comments during the preparation of this document. This work was partially supported by project FastFix (FP7-ICT-2009-5-258109).

REFERENCES


