DE GRUYTER OPEN Proceedings on Privacy Enhancing Technologies ; 2016 (2):4–21

João Matos, João Garcia, and Nuno Coração

Isolating Graphical Failure-Inducing Input for Privacy Protection in Error Reporting Systems

Abstract: This work proposes a new privacy-enhancing system that minimizes the disclosure of information in error reports. Error reporting mechanisms are of the utmost importance to correct software bugs but, unfortunately, the transmission of an error report may reveal users' private information. Some privacy-enhancing systems for error reporting have been presented in the past years, yet they rely on path condition analysis, which we show in this paper to be ineffective when it comes to graphical-based input. Knowing that numerous applications have graphical user interfaces (GUI), it is very important to overcome such limitation. This work describes a new privacy-enhancing error reporting system, based on a new input minimization algorithm called GUIMIN that is geared towards GUI, to remove input that is unnecessary to reproduce the observed failure. Before deciding whether to submit the error report, the user is provided with a step-by-step graphical replay of the minimized input, to evaluate whether it still yields sensitive information. We also provide an open source implementation of the proposed system and evaluate it with well-known applications.

Keywords: Privacy, Error reporting, Fault-replication, Software maintenance, Combinatorial testing.

DOI 10.1515/popets-2016-0002
Received 2015-08-31; revised 2015-11-30; accepted 2015-12-02.

1 Introduction

Most software products are released with errors that pass undetected through the program's testing phase. Error reporting is an essential part of software maintenance because it signals new errors that happen on software deployed at the clients and provides maintenance engineers with information on how to reproduce and, subsequently, investigate the error. Correcting software errors is a hard and time-consuming task that represents several billion dollars per year worth of software maintenance costs in Europe and the USA alone [1–3]. Consequently, error reporting has evolved to try to reduce the costs of software maintenance.

One of the main hindrances to error reporting is the concern of privacy, especially when using applications that deal with confidential/sensitive information. Nowadays, security breaches and identity theft are real dangers, hence users are likely to take a hard look at some information-gathering mechanisms on personal devices [4]. Whether users are working on a confidential document or have clicked/typed in personal information, sensitive private data is likely to be included in the error report [5]. Since, nowadays, most computers include error reporting systems, we believe that addressing the privacy concerns in such systems is fundamental.

In the past years, some privacy-enhancing systems were proposed based on the replacement of potentially private information with alternative data, while still ensuring the reproduction of the observed failure [5–10]. Given a log containing all user data input by a user during an execution of the program, these systems re-execute the application symbolically and obtain the set of logical constraints on the user input (a.k.a. path condition [11]) that leads to the failure. It is assumed that the user input is the source of information that raises the most privacy concerns. Once the path condition is obtained, these systems resort to an SMT solver (Satisfiability Modulo Theories [12–15]) to obtain alternative solutions that satisfy the path condition. Unfortunately, these systems are unable to obfuscate graphical-based input, for reasons that we explain and exemplify in this paper (Sec. 3). Given that most modern applications have graphical user interfaces, this is a critical flaw in current systems. Learning which graphical components were targeted by the user, may indeed reveal a substantial amount of sensitive information, e.g. preferences selected by mouse events in a confidential form.

We show in this paper that it is possible to minimize the disclosure of private information by removing subsets of graphical-based input that are unnecessary to reproduce the observed failure. Unfortunately, the ex-

João Matos: INESC-ID / Instituto Superior Técnico da Universidade de Lisboa, E-mail: jmatos@gsd.inesc-id.pt
João Garcia: INESC-ID / Instituto Superior Técnico da Universidade de Lisboa, E-mail: jog@gsd.inesc-id.pt
Nuno Coração: Vodafone, E-mail: nunocoracao@gmail.com

isting input minimization systems (e.g. delta-debugging systems [16]) do not cope with the structure of the GUI, thereby causing a severe performance degradation, which limits their usefulness in the context of GUI minimization. We propose a new error report minimization system that relies on a new input minimization algorithm called GUIMIN that takes into account the structure of the application's GUI. This system is, to the best of our knowledge, the first to address privacy concerns of graphical input in error reports. This paper starts with a brief description of previous systems in Sec. 2 and examples exposing both their potentialities and limitations in Sec. 3. Then, this paper presents the proposed system in Sec. 4, and its contributions are organized in the following way:

- Section 4.2 presents a new record and replay system of graphical input that also logs the GUI structure;
- Section 4.3 presents a new input minimization algorithm called GUIMIN that minimizes graphical input, while respecting the structure of the GUI;
- Section 4.4 presents a new system that:
  - Minimizes graphical input using GUIMIN;
  - Provides integration capabilities with previous obfuscation systems;
  - Provides a graphical step-by-step demonstration/replay to the user of the minimized input, before asking for permission to transmit the error report;
  - Provides the same demonstration to the developers, if the error report was indeed transmitted;
  - Is open-source.

Finally, Sec. 5 provides an evaluation of the proposed system with a comprehensive set of real-world applications and bugs, before some concluding remarks presented in Sec. 7. It is also worth mentioning that this paper includes appendices at the end of this document.

2 Related Work

This section presents some related work on privacy enhancing technologies for error reporting systems.

Error Reporting and Fault-Replication Systems. Automatic error reporting systems automatically incorporate in error reports information about the final state of a crashed application (e.g. stack trace, memory snapshot), before asking the user for permission to transmit it. This information is often insufficient. Thus, error reports can be complemented with fault-replication systems [17–20]. Fault-replication systems are record and replay systems that monitor the user execution and log sources of non-determinism (e.g. input, thread interleaving) enabling maintenance teams to deterministically replay the observed failure.

Input Obfuscation Systems. The obfuscation of fault-replication logs by replacing the user's input with an alternative input that induces the same failure was first proposed by Castro et al. [5] and later extended by other works [6–8]. These approaches re-execute the program symbolically through the original execution path that leads to the observed failure (i.e., using the original user inputs) and record the sequence of logical constraints imposed by the logical tests performed on the input (a.k.a. path condition [11]). Then an SMT solver is used (e.g. Z3 [12]) to obtain an alternative input that satisfies the same sequence of logical constraints. Later, two systems were proposed that perform symbolic execution through execution paths other than the original one: *i*) MPP [9] performs symbolic execution through all execution paths that reproduce an observed failure; however this was demonstrated not to be scalable except for micro-benchmarks; *ii*) REAP [10] performs only one additional symbolic execution through an execution path different from the original one.

Input minimization systems. A straightforward approach to address the privacy concerns in error reports is to remove all privacy-sensitive data.

●*Scrash.* The Secure Crash System [21] removes all potentially sensitive information, disregarding its potential relevance to error reproduction. This raises two problems: *i*) it may amputate information necessary to locate the causes of the observed failure from the error report, making the error fixing task substantially harder and sometimes impracticable; *ii*) it does not necessarily protect the users' privacy and may mislead users to believe that their data is safe when it may not be the case (e.g. if the reported failure is reproduced only by a specific and confidential input).

●*Delta-debugging.* To the best of our knowledge, delta-debugging was proposed by Zeller and Hildebrandt [16] and was created for debugging purposes. The idea is to facilitate the task of the developers in finding the cause of an observed failure, by providing them with a smaller input that also reproduces the observed failure. Other systems [22–25] have been presented in the past years that extend the work of Zeller and Hildebrandt. Unfortunately, none of the systems published so far have been optimized for graphical-based input and for this reason their performance is likely to be sub-optimal for applications with GUIs.

3 Motivation

This section details the most relevant state-of-the-art privacy enhancing error reporting systems and explains, using examples, why they are not suitable to address the privacy concerns of graphical-based input.

3.1 Input Obfuscation Systems

Input obfuscation [5–8] replaces private information rather than omitting it. Given an $\mathcal{F}$-inducing user input $\mathcal{I}^u$ (a sequence of input events $e_1, e_2, ..., e_n$ that induce an observed failure $\mathcal{F}$), these approaches re-execute the application symbolically, along the execution path originally traversed, in order to derive the set of logical constraints satisfied by $\mathcal{I}^u$. This set of restrictions is called a path condition [11] and we denote with $\mathcal{PC}^u$ the path condition satisfied by $\mathcal{I}^u$. In short, the symbolic execution procedure of these systems is as follows:

1. Given the $\mathcal{F}$-inducing input $\mathcal{I}^u$, the application's symbolic execution starts with $\mathcal{PC}^u = \emptyset$;
2. Execute the next instruction. If $\mathcal{F}$ was induced, terminate the symbolic execution and return $\mathcal{PC}^u$;
3. If the application reads an input value and assigns it to a variable, mark this variable as symbolic;
4. If the current instruction is a logical test performed on a symbolic variable, two new possible paths are generated, one if the test returns *true* and another if it returns *false*. Only one of them satisfies $\mathcal{I}^u$, so the symbolic execution engine follows that path, adds the respective constraint to $\mathcal{PC}^u$ and continues in step 2;

In addition to the procedure described above, the REAP framework [10] performs a second symbolic execution to compute a second path condition $\mathcal{PC}^a$, thereby granting the input space of the union $\mathcal{PC}^u \cup \mathcal{PC}^a$.

The final step in all described obfuscation systems is to use an SMT solver to draw an input from the set of inputs that satisfy $\mathcal{PC}^u$, or from the set of inputs that satisfy $\mathcal{PC}^u \cup \mathcal{PC}^a$ if REAP is used. In both cases, their effectiveness is explained next.

Effectiveness. The potential and limitations of path condition analysis in input obfuscation systems is directly related to the number of solutions that satisfy the computed path condition(s). An adversary may use the same symbolic execution procedures to compute the path condition(s) used to generate the alternative input $\mathcal{I}^a$. This means that the uncertainty of the adversary, in terms of which was the original input $\mathcal{I}^u$, is proportional to the number of different inputs that satisfy the path condition(s). The uncertainty is zero if only one input is satisfiable and, oppositely, is complete if any input satisfies the path condition.

3.1.1 Example: TV-Browser

The TV-Browser [26] application is a popular TV guide that allows the user to subscribe over 1000 channels and radio stations. Figure 1 depicts, on the left side, a representation of the TV-Browser graphical user interface and, on the right side, a code excerpt of a function that filters, amongst all the available channels, those whose names contain the word input by the user in the text field "search".

3.1.1.1 String Obfuscation

First, assume that the goal is to obfuscate the word input by the user in the "search" text field of Fig. 1, e.g. the string "cnn". If the application starts by testing, using the `isSubSeq`, function whether the channel name "showtime" (parameter `w2`) contains the input "cnn" (parameter `w1`), the path condition $\mathcal{PC}^u$ is:

$$\begin{array}{l}
w1.charAt(0) \neq 's' \wedge w1.charAt(1) \neq 'h' \wedge w1.charAt(2) \neq 'o' \wedge \\
w1.charAt(0) \neq 'h' \wedge w1.charAt(1) \neq 'o' \wedge w1.charAt(2) \neq 'w' \wedge \\
w1.charAt(0) \neq 'o' \wedge w1.charAt(1) \neq 'w' \wedge w1.charAt(2) \neq 't' \wedge \\
w1.charAt(0) \neq 'w' \wedge w1.charAt(1) \neq 't' \wedge w1.charAt(2) \neq 'i' \wedge \\
w1.charAt(0) \neq 't' \wedge w1.charAt(1) \neq 'i' \wedge w1.charAt(2) \neq 'm' \wedge \\
w1.charAt(0) \neq 'i' \wedge w1.charAt(1) \neq 'm' \wedge w1.charAt(2) \neq 'e' \\
w1.charAt(0) \neq 'm' \wedge w1.charAt(1) \neq 'e'
\end{array}$$

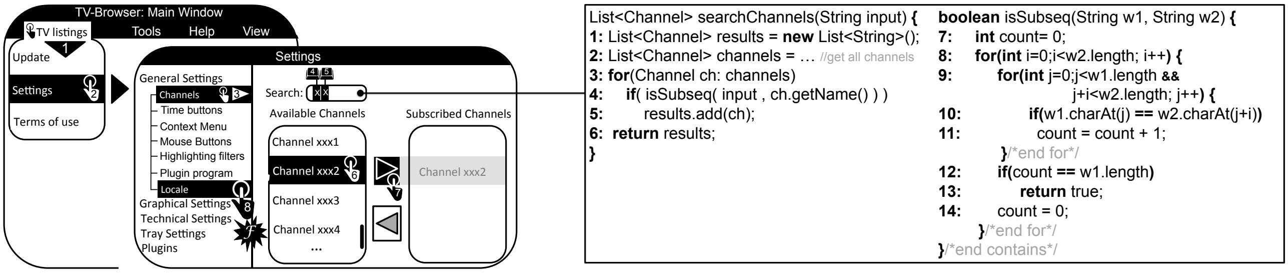
in which w_1 was treated as symbolic and w_2 was not. It is straightforward to conclude that numerous inputs satisfy $\mathcal{PC}^u$: the solver is able to pick an alternative $\mathcal{PC}^u$-satisfying input $\mathcal{I}^a$ and an adversary cannot identify the input of $\mathcal{I}^u$ amongst all possible inputs.

On the other hand, if the application tests whether "cinemax" contains the input "cnn", then $\mathcal{PC}^u$ is:

$$\begin{array}{l}
\underline{w1.charAt(0) = 'c'} \wedge w1.charAt(1) \neq 'i' \wedge \underline{w1.charAt(2) = 'n'} \wedge \\
w1.charAt(0) \neq 'i' \wedge \underline{w1.charAt(1) = 'n'} \wedge w1.charAt(2) \neq 'e' \wedge \\
w1.charAt(0) \neq 'n' \wedge w1.charAt(1) \neq 'e' \wedge w1.charAt(2) \neq 'm' \wedge \\
w1.charAt(0) \neq 'e' \wedge w1.charAt(1) \neq 'm' \wedge w1.charAt(2) \neq 'a' \wedge \\
w1.charAt(0) \neq 'm' \wedge w1.charAt(1) \neq 'a' \wedge w1.charAt(2) \neq 'x' \wedge \\
w1.charAt(0) \neq 'a' \wedge w1.charAt(1) \neq 'x'
\end{array}$$

in which the constraints are redundant due to the underlined ones. In this case, $\mathcal{PC}^u$ is useless for obfuscation, as it is only satisfied by the input "cnn" and consequently, the uncertainty of the adversary is zero.

The REAP framework was created to address this problem. By computing an alternative path condition $\mathcal{PC}^a$, REAP is indeed an effective approach assuming that $\mathcal{PC}^u \cup \mathcal{PC}^a$ permits a significant amount of inputs.



```
List<Channel> searchChannels(String input) {
List<Channel> results = new List<String>();
List<Channel> channels = … //get all channels
for(Channel ch: channels)
    if( isSubseq( input , ch.getName() ) )
        results.add(ch);
 return results;
}
```

```
boolean isSubseq(String w1, String w2) {
    int count= 0;
    for(int i=0;i<w2.length; i++) {
        for(int j=0;j<w1.length &&
                    j+i<w2.length; j++) {
            if(w1.charAt(j) == w2.charAt(j+i))
                count = count + 1;
        }/*end for*/
        if(count == w1.length)
            return true;
        count = 0;
    }/*end for*/
}/*end contains*/
```

Fig. 1. An illustrative example, in which the user subscribes channels in the Tv-Browser application [26]. The user inputs the events {1, 2, . . . 8} and events {4,5} trigger the code exposed in the right side of the picture. Finally, a click in the “locale" component, results in an immediate failure, which was reported in 2008 [27].

```
/*class EventQueue*/
void dispatchEvent(AwtEvent e) {
  Component src = getSource(e);
  src.dispatchEvent(e);
}
```

```
Component getSource(AwtEvent e) {
  for(int i=0;i<UI.getComponents().length; i++) {
      Component comp = UI.getComponents()[i];
      if( comp.contains( e.getX() , e.getY() ) ) {
          return comp;
 }/*end for*/}/*end method*/
```

Fig. 2. Example of widget identification.

3.1.1.2 GUI Obfuscation

In short, obfuscation systems are not suitable privacy-wise to obfuscate graphical-based input, because, as already mentioned, *their effectiveness relies on the existence of path condition(s) satisfied by several inputs*, which is not the case in the context of graphical input. In fact, when the goal is to conceal the graphical components targeted by the user, each and every path condition achievable is satisfied by only one sequence of graphical components.

Using the above-described obfuscation systems to obfuscate graphical-based input implies performing symbolic execution in the GUI libraries. For example, in the Java programming language, in order to detect which graphical component was acted upon by a mouse event, the *EventQueue* class verifies which graphical component contains the coordinates of the current mouse event, as exemplified by the code excerpt in Fig. 2. Knowing that each pair of mouse coordinates fall within only one of the widgets in the current visible window, performing symbolic execution in GUI classes, would generate a path condition with a structure similar to the following:

$$ev_1.X, ev_1.Y \in \mathcal{C}_i \quad \wedge \quad ev_2.X, ev_2.Y \in \mathcal{C}_j \quad \wedge \quad \dots$$

in which $ev_*.X, ev_*.Y$ are the coordinates of the mouse event ev_* and $\mathcal{C}_*$ is the target component of the GUI. There are two possible approaches to perform symbolic execution in this context, which are described in the following and ultimately result in the same problem.

- The coordinates are symbolic variables and the widgets are not, which for the example of Fig. 1, would generate a path condition $\mathcal{PC}^u$ similar to:

$$ev_1.X, ev_1.Y \in \text{Component<JMenu,“TV listings”>} \ \wedge$$
$$ev_2.X, ev_2.Y \in \text{Component<JMenuItem,“add/rem”>} \ \wedge \quad \dots$$

- The coordinates are concrete variables and the widgets are symbolic, consequently, $\mathcal{PC}^u$ would be similar to:

$$(2, 10) \in \mathcal{C}_i \quad \wedge \quad (3, 7) \in \mathcal{C}_j \quad \wedge \quad (10, 7) \in \mathcal{C}_k \quad \wedge \quad \dots$$

In both cases, $\mathcal{PC}^u$ is limited by a problem similar to the one exemplified in the second part of Sec. 3.1.1.1: a pair of coordinates is contained by only one widget in the current window. Therefore, there is only one sequence of widgets that satisfies $\mathcal{PC}^u$ and, since the goal is to conceal which were the widgets targeted by the user, prior obfuscation systems [5–8] based on path condition analysis cannot be used to obfuscate graphical input.

REAP can go further than these systems, as it achieves one additional path condition $\mathcal{PC}^a$, thereby guaranteeing the solutions of the set $\mathcal{PC}^u \cup \mathcal{PC}^a$. Unfortunately, for the exact same reasons, $\mathcal{PC}^a$ is also satisfied by only one sequence of widgets. Thus, because the path conditions describing very concrete sequences of widgets have each a very small input domain, the union of these path conditions does not significantly enlarge the input domain and therefore does not provide a significant privacy enhancement.

Furthermore, knowing that the *boolean satisfiability problem is NP-complete* [28], an SMT solver may require from fractions of a second to several days to decide whether the path condition output by the symbolic execution is satisfiable, depending on its size and complexity. Knowing that a typical interaction between the user and the GUI can easily generate thousands of events [29], the SMT solver may take a substantial amount of time to process the resulting path condition. This is further aggravated in the case of REAP, as it requires multiple invocations to the solver during the symbolic execution.

4 Proposed Framework

In a typical interaction between the user and a GUI, thousands of events are generated by the user and, intuitively, a large subset of them is unlikely to be related to the observed failure $\mathcal{F}$. Thus, privacy protection can be achieved by subtracting from the recorded input, which is an ordered list of events triggered by the user $\mathcal{I}^u = \{e_1, e_2, \ldots, e_{|\mathcal{I}^u|}\}$, a subset of events that we find not to be related to $\mathcal{F}$. In other words, we propose input minimization as a privacy protection technique for GUI-based applications.

We start by introducing the baseline input minimization algorithm proposed by Zeller and Hildebrandt [16] and explain why it is not suitable to minimize graphical input, before presenting the proposed system.

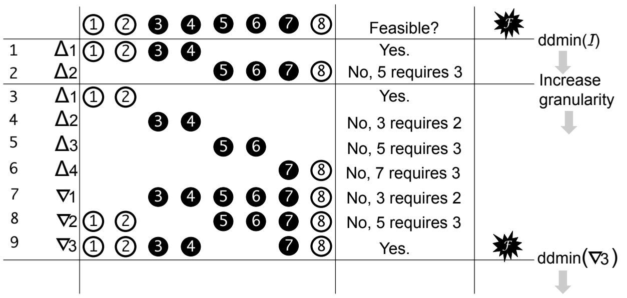
4.1 Baseline Comparison

The goal of input minimization is to achieve a (smaller) $\mathcal{F}$-inducing sublist $\mathcal{I}^a$ such that $\mathcal{I}^a \subset \mathcal{I}^u$. We can calculate such a sublist by re-executing the application and replaying (smaller) sublists of $\mathcal{I}^u$, to test for $\mathcal{F}$. Once we find a smaller $\mathcal{F}$-inducing input sublist of $\mathcal{I}^u$, we continue minimizing $\mathcal{I}^u$ by repeating the procedure starting from the newly achieved $\mathcal{F}$-inducing sublist.

The ddmin algorithm. The first minimization algorithm was ddmin, which was presented by Zeller and Hildebrandt [16]. This type of approach is also known as delta debugging and was created for debugging purposes. The idea is to simplify the search for the causes of $\mathcal{F}$, by providing maintenance teams with a smaller $\mathcal{F}$-inducing input. The *ddmin* algorithm works as follows:

1. Start with $\mathcal{I}^a = \mathcal{I}^u$.
2. Split $\mathcal{I}^a$ into n sublists, of equal size if possible. Test each sublist for $\mathcal{F}$. If one of them induces $\mathcal{F}$, then it becomes $\mathcal{I}^a$ and resume the algorithm at Step 2.
3. Test the complement of each sublist. If any of them induces $\mathcal{F}$ then it becomes $\mathcal{I}^a$ and the algorithm is resumed at Step 2.
4. Attempt to split $\mathcal{I}^a$ into smaller partitions, $2n$ if possible and resume at Step 2. If $\mathcal{I}^a$ cannot be split into smaller sublists, the procedure is finished.

A practical hindrance. One of the main differences between minimizing graphical input and minimizing a file of any type, is the requirement that GUI inputs be replayed interactively. In the case of file-based input, we can test a smaller version of a file for $\mathcal{F}$ simply by passing it as a parameter to the starting application. On the other hand, to test a sublist of graphical events, one has to re-execute the application and dispatch each event in the GUI, individually, in the correct order and when the target widget is available/visible. Consequently, testing a sublist of graphical events is inevitably slower and each new test performed adds a significant scalability cost.



		①	②	❸	❹	❺	❻	❼	⑧	Feasible?	$\mathcal{F}$
1	Δ1	①	②	❸	❹					Yes.	
2	Δ2					❺	❻	❼	⑧	No, 5 requires 3	
3	Δ1	①	②							Yes.	
4	Δ2			❸	❹					No, 3 requires 2	
5	Δ3					❺	❻			No, 5 requires 3	
6	Δ4							❼	⑧	No, 7 requires 3	
7	∇1			❸	❹	❺	❻	❼	⑧	No, 3 requires 2	
8	∇2	①	②			❺	❻	❼	⑧	No, 5 requires 3	
9	∇3	①	②	❸	❹			❼	⑧	Yes.	$\mathcal{F}$

Fig. 3. An illustration of the *ddmin* algorithm applied to the motivational example of Fig. 1. Two thirds of the tests attempted are infeasible due to dependencies imposed by the GUI.

Why not use ddmin to minimize graphical input? The *ddmin* algorithm does not take into consideration any type of application and/or input structure. Disregarding the structure of the input leads to a substantial amount of unnecessary tests being performed, which in turn causes severe performance degradation.

Consider the illustration in Fig. 3. If we split the input $\mathcal{I}^u = \{1, 2, 3, 4, 5, 6, 7, 8\}$ uniformly into sublists of size n, several will correspond to sequences of events that are not viable input to the GUI application, mainly because they include events that are meaningless at the execution point where they are placed in the sublist. For example, splitting $\mathcal{I}^u$ may create an input sequence including a click event on a button that is not visible. These tests are infeasible and they pollute and delay the minimization process. For example, if we split this list into two sublists of equal size, the sublist $\{5, 6, 7, 8\}$ is not feasible, because it contains events that occurred in a specific window that becomes visible as a consequence of some events within the first sublist. This problem is aggravated if we perform finer-grained splits. Eventually, *ddmin* minimizes $\mathcal{I}^u$, however it requires an unreasonable amount of testing and time, even if we consider running it during the user device's idle periods.

Next, we present a record and replay system for GUI-based applications, before introducing a new input minimization algorithm geared towards graphical input.

4.2 A GUI-based Record and Replay System

We require a recording system that monitors the user execution and logs the interaction with the GUI. Furthermore, due to the hindrance described in Sec. 4.1, we also require a replayer capable of reproducing each graphical component, individually and in the correct order, for a three-fold purpose: *i*) test sublists of $\mathcal{I}^u$, during the minimization process; *ii*) to allow the user to visualize the minimized input, before asking for permission to transmit the error report; *iii*) to allow developers to visualize the minimized $\mathcal{F}$-inducing input.

Our system copes with the structure of the GUIs at the record&replay level by recording the application not just as a sequence of events but with additional context information. The structure of the GUI is explored by considering three properties of graphical-based input:

- A user input event e_i is an action upon a target graphical widget w_i;
- Widgets are contained by other widgets also known as *containers*;
- An input event e_i may trigger the appearance of a *container*.

The proposed record and replay system is formalized in Algorithm 1 and explained in Secs. 4.2.1 and 4.2.2. We believe that the proposed model is general enough to be applied for most GUI frameworks.

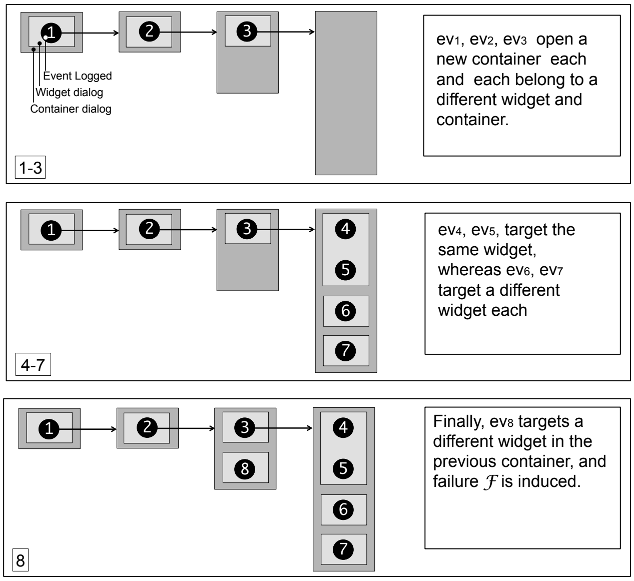
Algorithm 1: Pseudo code defining our record and replay procedures.

```
Interface ItemLogged()
DataStructure $c$Dialog implements ItemLogged(
    $w$Dialog $w$head, $w$tail;//first&last widgets
    $c$Dialog prev;)//container
DataStructure $w$Dialog implements ItemLogged(
    widget $w$;
    List<$\epsilon$Logged> $\mathcal{E}$;
    $w$Dialog next;)//widget
DataStructure $\epsilon$Logged implements ItemLogged(
    Event $ev$;
    $c$Dialog $next$;)//event
Shared:
$c$Dialog chead, ctail; //first&last recorded containers
Failure $\mathcal{F}$; //the observed failure

Record()
void
begin
    while The application is running do
        Wait for the next event $e_i$ triggered by the user;
        RecordEvent($e_i$);
        if The program fails then
            $\mathcal{F}$ ⟵ catch the observed failure;
            break;

RecordEvent(Event $e_i$)
void
begin
    if $ctail$ = null then
        ctail ⟵ initialize;
        ctail.prev ⟵ null;
        chead ⟵ ctail;
    if $ctail.wtail$ = null then
        ctail.wtail ⟵ new empty $w$Dialog;
        ctail.wtail.w ⟵ target widget of $e_i$
        ctail.whead ⟵ ctail.wtail;
    else if $ctail.wtail.w \neq$ target widget of $e_i$ then
        ctail.wtail.next ⟵ new empty $w$Dialog;
        ctail.wtail ⟵ ctail.wtail.next;
        ctail.wtail.w ⟵ target widget of $e_i$;
    $\epsilon$Logged $\epsilon_i$ ⟵ new empty $\epsilon$Logged;
    $\epsilon_i$.ev ⟵ $e_i$;
    add $\epsilon_i$ to ctail.wtail;
    if a new container became visible then
        $\epsilon_i$.next ⟵ new empty $c$Dialog;
        $\epsilon_i$.next.prev ⟵ ctail;
        ctail ⟵ $\epsilon_i$.next;
    else if ctail is no longer visible then
        ctail ⟵ ctail.prev;

Replay(List<ItemLogged> noReplay)
Returns: true if $\mathcal{F}$ is reproduced; false otherwise
begin
    return Replay(chead , noReplay, $\mathcal{F}$);

Replay($c$Dialog cnext , List<ItemLogged> noReplay, $\mathcal{F}$)
Returns: true if $\mathcal{F}$ is reproduced; false otherwise
begin
    $w$Dialog $w_i$ ⟵ cnext.whead;
    while $w_i \neq null$ do
        if $w_i \notin noReplay$ then
            foreach $\epsilon_i \in w_i.\mathcal{E} \wedge \epsilon_i \notin noReplay$ do
                replay $\epsilon_i.ev$;
                if $\mathcal{F}$ was reproduced then
                    return true;
                if $\epsilon_i.next \neq$null then
                    if Replay($\epsilon_i$.next,noReplay) =true
                    then
                        return true;
        $w_i$ ⟵ $w_i$.next;
    return false;
```

4.2.1 Record

The user executes the application normally, while the execution is monitored and the GUI input recorded, according to the model formalized in Algorithm 1, illustrated in Fig. 4 and explained below.

- Users may interact with a single widget, triggering one or more events in a row and we refer to such interaction as a *dialog*. More specifically, when the user triggers one or more events in a row within a single widget, we say that it is a *widget dialog*. Furthermore, a continuous set of *widget dialogs* within the same container, from the moment it was opened until it was closed, is called a *container dialog*.
- The recording procedure starts with the instance *chead* of a data structure representing the main window, that we call *container dialog* or cDialog.
- If the user performs one or more actions on a specific widget w_i inside the current container, then we create a new instance of the data structure *widget di-*



ev1, ev2, ev3 open a new container each and each belong to a different widget and container.

ev4, ev5, target the same widget, whereas ev6, ev7 target a different widget each

Finally, ev8 targets a different widget in the previous container, and failure $\mathcal{F}$ is induced.

Fig. 4. An illustrative example of the proposed GUI-structured recording, applied to the example of Fig. 1.

alog or wDialog, which represents the widget, and place those events inside that wDialog. The previous wDialog (if any) of this cDialog, is linked to this new wDialog, thereby forming a linked list within the current cDialog.

- Additionally, every new event is logged in a new instance of the ϵLogged data structure. If the current event triggers the opening of a new container (e.g. a new window is opened), then field *next* of the respective ϵLogged instance, points to the newly created cDialog instance that represents the newly opened container.
- Asynchronous events are recorded in the same order as they occur, to assure total ordering of events.

4.2.2 Replay

The proposed replaying procedure is also exposed in Algorithm 1 and complies with the following definition:

Definition 1. The function Replay($\mathcal{I}^*$,$\mathcal{I}^{**}$,$\mathcal{F}$) replays $\mathcal{I}^* \setminus \mathcal{I}^{**}$, requiring that $\mathcal{I}^{**} \subseteq \mathcal{I}^*$. It returns the boolean value *true* if $\mathcal{F}$ is observed or *false* otherwise.

The replaying procedure takes the first recorded container dialog and iterates through the data structure, reproducing each recorded event in the same order as it was recorded. The Replay function is able to test any sublist of the recorded input $\mathcal{I}^u$, as the *noReplay* parameter indicates which events and/or widgets should be ignored. Before presenting the proposed algorithm, consider the following notation.

Notation. Given an input $\mathcal{I}^*$, a list of events $\mathcal{I}^* = \{e_1, e_2, \ldots, e_{|\mathcal{I}^*|}\}$, we denote with $\mathcal{E}^*$ the list of ϵLogged $\mathcal{E}^* = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_{|\mathcal{E}^*|}\}$ such that $\epsilon_1.ev = e_1$, $\epsilon_2.ev = e_2$, ..., $\epsilon_{|\mathcal{E}^*|}.ev = e_{|\mathcal{I}^*|}$. We denote as $\epsilon_i.next$ the pointer to the container dialog corresponding to the container that is opened by the event stored at $\epsilon_i.ev$ (as exposed in the pseudocode). Furthermore, we denote with $\mathcal{W}^* = \{w_1, w_2, \ldots, w_{|\mathcal{W}|}\}$ the list of wDialog such that we obtain $\mathcal{E}^*$ by concatenating $w_i.\mathcal{E} : \forall w_i \in \mathcal{W}^*$. We also denote the function $\textsc{Replay}(\mathcal{E}^*, \mathcal{E}^{**}, \mathcal{F})$ analogously to the function $\textsc{Replay}(\mathcal{I}^*, \mathcal{I}^{**}, \mathcal{F})$ of Definition 1.

4.3 The GUImin Algorithm

This section presents a new input minimization algorithm called GUImin that acknowledges the structure of the GUI, thereby avoiding infeasible tests and consequently optimizing the minimization process. The record and replay procedures presented in the previous section play a relevant role in GUImin: the recorder captures $\mathcal{I}^u$ and $\mathcal{F}$, whereas the replayer tests sublists of $\mathcal{I}^u$. We start by introducing the goals of GUImin in terms of minimization capabilities. Then, before formalizing GUImin, we present an algorithm called LazyMin, which is fundamental for the minimization procedure of GUImin.

4.3.1 Minimization goals

Ideally, we would want to guarantee that the $\mathcal{F}$-inducing sublist found $\mathcal{I}^a$ is the smallest possible $\mathcal{F}$-inducing one, also known as a *Global Minimum* [16]:

Definition 2. $\mathcal{I}^a$ is a global minimum if $\mathcal{I}^a \subset \mathcal{I}^u \wedge$ $\textsc{Replay}(\mathcal{I}^a, \emptyset, \mathcal{F}) = true$ and $\forall \mathcal{I}^i \subset \mathcal{I}^u$ such that $|\mathcal{I}^i| < |\mathcal{I}^a|$ it must be that $\textsc{Replay}(\mathcal{I}^i, \emptyset, \mathcal{F}) = false$.

Unfortunately, to the extent of our knowledge, the only minimization algorithms that guarantee finding a global minimum resort to exhaustively testing all possible sublists of $\mathcal{I}^u$ to check whether they are $\mathcal{F}$-inducing. Unless $\mathcal{I}^u$ is very small, the cost imposed by such algorithms is prohibitive. On the other hand, *ddmin* guarantees that a 1-minimal input is achieved:

Definition 3. $\mathcal{I}^a$ is 1-minimal if $\textsc{Replay}(\mathcal{I}^a, \emptyset, \mathcal{F}) = true$ and $\forall e_i \in \mathcal{I}^a$ it must be that $\textsc{Replay}(\mathcal{I}^a, e_i, \mathcal{F}) = false$.

To guarantee that the minimized sublist $\mathcal{I}^a$ is 1-minimal, one has to test $\textsc{Replay}(\mathcal{I}^a, e_i, \mathcal{F})$, for each and every event e_i in $\mathcal{I}^a$ to assure that $\mathcal{F}$ is not reproduced. This kind of guarantee makes sense in scenarios in which there are no dependencies amongst the inputs in $\mathcal{I}^a$. As already explained, this not the case for graphical-input and would result in attempting infeasible tests. Knowing that subtracting an event e_i from $\mathcal{I}^a$ may result in an infeasible sublist, we are not particularly interested in achieving a 1-minimal sublist. Instead we are more interested in learning whether $\mathcal{F}$ is reproduced if we remove a single dialog from $\mathcal{I}^a$, instead of a single event. Therefore we introduce a new definition on minimality:

Definition 4. An input $\mathcal{I}^a$ is 1-dialog-minimal iff
$\textsc{Replay}(\mathcal{E}^a, \emptyset, \mathcal{F}) = true \quad \wedge$
$\forall w_i \in \mathcal{W}^a, \textsc{Replay}(\mathcal{E}^a, w_i.\mathcal{E}, \mathcal{F}) = false \quad \wedge$
$\forall \epsilon_i \in \mathcal{E}^a : \epsilon_i.next \neq \emptyset, \textsc{Replay}(\mathcal{E}^a, \epsilon_i, \mathcal{F}) = false$

As we will demonstrate in this paper, the proposed algorithm GUImin guarantees the achievement of a 1-dialog-minimal sublist of $\mathcal{I}^u$, as in Definition 4.

4.3.2 LazyMin

Algorithm 2: Pseudo code defining a minimization algorithm called LazyMin.

```
Shared:
cDialog chead;//the first recorded container

LazyMin(List<ItemLogged> L)
void
begin
    int $n \longleftarrow |L|$;
    List<ItemLogged> L';
    do
        $n \longleftarrow \frac{n}{2} > 1 \;\; ? \;\; \frac{n}{2} : 1$
        L'⟵ΔMin (L,n);
        if L'$\neq \emptyset$ then
            L ⟵ L'
        else
            L ⟵greedy∇Min(L,n);
    while $n > 1$;

ΔMin(List<ItemLogged> L , int n)
Returns: List<ItemLogged>
begin
    List<List<ItemLogged>> L'⟵ split L in sublists of
    size n;
    foreach $l_i \in$ L' do
        if Replay($L \setminus l_i$) = true then
            Trim($L \setminus l_i$);
            return $l_i$;
    return $\emptyset$;

greedy∇Min(List<ItemLogged> L , int n)
Returns: List<ItemLogged>
begin
    List<List<ItemLogged>> L'⟵ split L in sublists of
    size n;
    foreach $l_i \in$ L' do
        if Replay($l_i$) = true then
            Trim($l_i$);
            $L \longleftarrow L \setminus l_i$;
    return L;

Trim(List<ItemLogged> trimList)
void
begin
    Trim'(chead , $l_i$);

Trim'(cDialog cnext,List<ItemLogged> trimList)
void
begin
    wDialog $w_i \longleftarrow$ cnext.whead;
    while $w_i \neq$ null do
        if $w_i \in trimlist$ then
            $w_{i-1}.next \longleftarrow w_{i+1}$;
        else
            foreach $e_i \in w_i.E$ do
                if $e_i \in trimList$ then
                    delete $e_i$ from $w_i$.E;
                    if $w_i.E = \emptyset$ then
                        $w_{i-1}.next \longleftarrow w_{i+1}$
                else if $e_i.next \neq null$ then
                    Trim'($e_i$.next,trimList);
        $w_i \longleftarrow w_i$.next;
```

The LazyMin algorithm, exposed in Algorithm 2, adopts the same binary search strategy of *ddmin* that attempts to achieve an $\mathcal{F}$-inducing sublist in logarithmic time. To do so it resorts to the ΔMin algorithm, also exposed in Algorithm 2, that is equivalent to the one used by *ddmin*: the list L is split into sublists of size n, which are tested for $\mathcal{F}$ one at a time. When a sublist l_i of L induces $\mathcal{F}$, the function Trim deletes $L \setminus l_i$ permanently and LazyMin resumes with l_i. LazyMin continues decreasing (exponentially) the size n of the lists, to test finer grained lists and keeps invoking ΔMin as long as it continues returning smaller $\mathcal{F}$-inducing lists.

When ΔMin fails to return a smaller $\mathcal{F}$-inducing list, then LazyMin resorts to greedy∇Min (a ∇test is the complement of a Δtest) to test the sublists' complements. If the complement of a sublist l_i induces $\mathcal{F}$, then the function Trim deletes l_i and LazyMin resumes with $L \setminus l_i$. The main difference between LazyMin and *ddmin* lies in greedy∇Min: *i*) it tests all complements of granularity n in one shot; *ii*) the testing of complements is never repeated. The reason for choosing LazyMin over *ddmin* lies in the worst-case time complexity. Considering, that we are not aiming at a 1-minimal sublist, it is not necessary to incur in the worst case of $|L|^2 + 3|L|$. Instead, LazyMin has a worst case of $4|L|$ tests performed, which consists of failing to reproduce $\mathcal{F}$ in all tests $(2 + 4 + \ldots + 2|L| = 4|L|)$. LazyMin on its own does not provide any guarantees on minimality, as it is meant to be used by the GUImin algorithm, which is presented next.

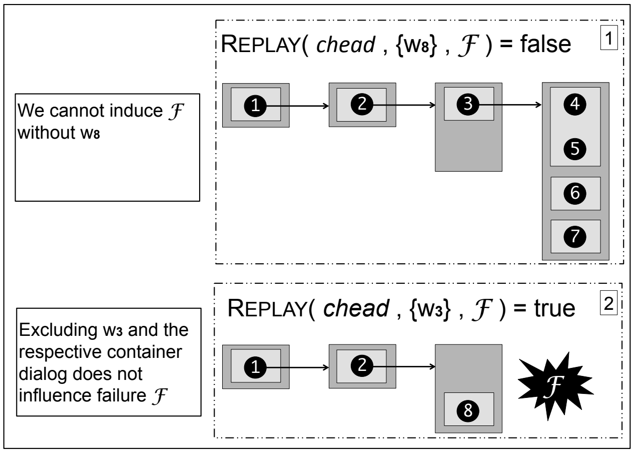

REPLAY(chead , {w8} , $\mathcal{F}$) = false
1
We cannot induce $\mathcal{F}$ without w8
1
2
3
4
5
6
7
REPLAY(chead , {w3} , $\mathcal{F}$) = true
2
Excluding w3 and the respective container dialog does not influence failure $\mathcal{F}$
1
2
8
$\mathcal{F}$

Fig. 5. The two possible tests of WMIN in the third container of the example of Fig. 1. The second test reproduces $\mathcal{F}$ while excluding the widget dialog w_3. By excluding w_3, its subsequent container is also excluded, thereby minimizing events {3,4,5,6,7}.

4.3.3 GUImin

The GUIMIN algorithm is detailed in Algorithm 3. It works in three phases: *i*) minimize widget dialogs using the $\mathcal{W}$MIN procedure; *ii*) minimize the events within the remaining widget dialogs using the $\mathcal{WE}$MIN procedure; *iii*) guarantee that the minimized input is 1-dialog-minimal as in Definition 4 (lines 3 to 8 of Algorithm 3).

GUIMIN starts by minimizing widget dialogs because it allows for trimming larger amount of dialogs and events at a time: GUI events are triggered within widget dialogs, including events that trigger the opening of container dialogs (which in their turn contain more dialogs and events). Furthermore, the procedures $\mathcal{W}$MIN, $\mathcal{WE}$MIN and $\mathcal{C}$MIN, invoke LAZYMIN one level at a time, using the function GETDIALOGS, which also trims larger amounts of input in fewer attempts. This is illustrated in Fig. 5: as soon as $\mathcal{W}$MIN tests the input without the widget dialog w_3, it excludes *all* inputs unnecessary to reproduce $\mathcal{F}$.

In the next step, GUIMIN invokes $\mathcal{WE}$MIN to minimize events within the remaining dialogs. This is particularly useful to minimize the content of dialogs that are most likely required to reproduce the failure. For example, if a container dialog contains a form, whose submission induces $\mathcal{F}$, and one/some of the text fields must be filled to enable the submission button, then $\mathcal{W}$MIN cannot trim those text fields from the widget dialogs, but $\mathcal{WE}$MIN may be capable of removing large parts of their content (e.g. the "name" text field can be left with a single character).

Algorithm 3: Pseudo code defining the GUIMIN algorithm.

```
   Shared:
   cDialog chead;//the first recorded container

   GUImin()
   void
   begin
1      $\mathcal{W}$Min();
2      $\mathcal{WE}$Min();
3      do
4          int count ⟵ count events;
5          $\mathcal{C}$Min();
6          $\mathcal{W}$Min();
7          int count' ⟵ count events;
8      while count' < count;

   $\mathcal{W}$Min()
   void
   begin
9      List<ItemLogged> $L \longleftarrow \emptyset$;
10     List<cDialog> $C \longleftarrow chead$;
11     do
12         $L \longleftarrow \forall w_i \in C$;
13         LazyMin($L$); //minimize widgets;
14         $C \longleftarrow$GetDialogs($C$);
15     while $C \neq \emptyset$;

   $\mathcal{WE}$Min()
   void
   begin
16     List<ItemLogged> $L \longleftarrow \emptyset$;
17     List<cDialog> $C \longleftarrow chead$;
18     do
19         foreach $w_i \in C$ do
20             $L \longleftarrow w_i.E$;
21             LazyMin($L$); //minimize events in widgets;
22         $C \longleftarrow$GetDialogs($C$);
23     while $C \neq \emptyset$;

   $\mathcal{C}$Min()
   void
   begin
24     List<ItemLogged> $L \longleftarrow \emptyset$;
25     List<cDialog> $C \longleftarrow chead$;
26     do
27         $L \longleftarrow \forall e_i \in C : e_i.next \neq null$;
28         LazyMin($L$); //minimize containers;
29         $C \longleftarrow$GetDialogs($C$);
30     while $C \neq \emptyset$;


   GetDialogs(List<cDialog> $C$)
   Returns: List<cDialog>
   begin
31     List<cDialog> $C' \longleftarrow \emptyset$
32     foreach $c_i \in C$ do
33         wDialog $w_i = c_i.w$head;
34         while $w_i \neq null$ do
35             foreach $e_i \in w_i.E$ do
36                 if $e_i.next \neq null$ then
37                     add $e_i.next$ to $C'$;
38             $w_i \longleftarrow w_i$.next;
39     return $C'$;
```

Finally, $\mathcal{C}$MIN and $\mathcal{W}$MIN are re-invoked in a loop, to assure that $\mathcal{I}^a$ is 1-dialog-minimal. The idea is that if the input size, after the invocation of $\mathcal{C}$MIN and $\mathcal{W}$MIN, is exactly the same as before, then it must be that by removing one (no matter which) dialog from $\mathcal{I}^a$ disables

$\mathcal{I}^a$ from reproducing $\mathcal{F}$, thereby taking us to the proposition exposed next.

Proposition 1. The GUImin algorithm always achieves a 1-dialog-minimal sublist $\mathcal{I}^a$, such that $\mathcal{I}^a \subseteq \mathcal{I}^u$, as in Definition 4.

In the best case scenario, GUImin performs as stated by the following proposition:

Proposition 2. The best case time complexity of GUImin is $log_2(|\mathcal{W}^u|) + log_2(|w_i.\mathcal{E}|) + 1$.

In the worst case, GUImin has a $\mathcal{O}(n^2)$ performance:

Proposition 3. The worst case time complexity of GUImin is $4|\mathcal{I}^u|^2 + 12|\mathcal{I}^u|$.

The proofs of Propositions 1, 2 and 3 are exposed in appendices A, B and C respectively.

4.4 Implementation and Interface

This section provides further insight on the proposed system. We start by describing the overall architecture of the proposed system as well as some implementation details in Sec. 4.4.1. Then, in Sec. 4.4.2, we explain how users interact with the proposed system and specifically how users can verify the content of the error reports before authorizing their transmission.

4.4.1 Architecture and Main Components

The overall architecture is depicted in Fig. 6. We implemented a prototype of our work in Java for Java programs with graphical user interfaces[1]. We provide a description of its main components.

Recorder. While the user executes the application normally (step 1), the recorder is responsible for logging the user input to the GUI (step 2), and for providing it to the input minimizer if and when a failure occurs (step 3). The recorder was implemented using the *EventListener* interface of Java. The recorder includes an event mask that can be configured to monitor only the types of event and widget that the developer considers to be relevant [30]. By default it records all. It is important to note that an over-restrictive event mask may exclude events that are relevant for reproducing $\mathcal{F}$. In a pre-deployment stage, the developer is responsible for configuring the event mask properly according to the nature of the target application.

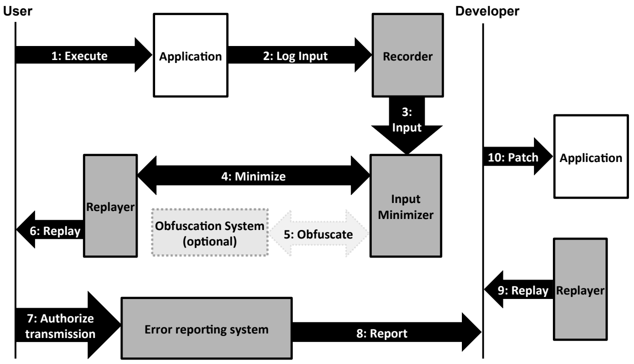


Fig. 6. Architecture of the proposed system.

Replayer. The replayer starts by running the application and dispatches each event in the respective widget, as specified in algorithm 1. As already mentioned, the replayer is used for three different jobs: *i*) for GUImin to test the sublists of $\mathcal{I}^u$ for $\mathcal{F}$; *ii*) for the user to visualize and verify the minimized input $\mathcal{I}^a$ before authorizing (or not) the transmission of the error report; *iii*) for the developers to visualize $\mathcal{I}^a$ just as the user did, while searching for the causes of $\mathcal{F}$.

Input Minimizer. The input minimizer receives the event list $\mathcal{I}^u$ logged by the recorder and runs GUImin (step 4), while resorting to the replayer to test each sublist. Once a 1-dialog minimal input $\mathcal{I}^a$ is achieved, optionally, one may choose to use one of the obfuscation systems presented in Sec. 3 to obfuscate textual-based input that GUImin is not able to minimize (step 5). In Sec. 5 we evaluate a practical example in which there are advantages in integrating GUImin with input obfuscation systems.

Error Reporting System. If and when the user authorizes the transmission of the error report (step 7), the latter is sent to the maintenance team (step 8). Then the maintenance staff may investigate the error (step 9) and potentially patch the application (step 10).

4.4.2 Interface

User Interface. The user can execute the application normally, without interacting with the proposed system, until a failure occurs. When a failure $\mathcal{F}$ occurs, the

[1] The prototype implementation of GUImin is available at https://github.com/jgmatos/GUImin/

recorded log is automatically moved to a folder dedicated for logs that are not yet minimized. *The minimization procedure is meant to be executed in the background during idle periods of the user's machine*, so that the user is not required to wait for it to finish. When the minimization procedure finishes, the minimized event list $\mathcal{I}^a$ is recorded into a replayable file, which is moved to a folder dedicated for logs that have been minimized.

Our system automatically opens a small dialog once the minimization is finished. If the timing is not good for the user, she may choose to verify the minimized logs later. If the user accepts to verify the log(s), she is presented with another window displaying a selectable list of logs, from which she can select a log and, with a single click, she may: *i*) decide to replay the selected log; *ii*) either report or discard the selected log; *iii*) close the window.

Privacy-wise error report verification. Quantifying the achieved privacy can be very subjective. The minimized input contains information that may be considered sensitive (or not) according to different interpretations of its meaning. Amongst the different possible interpretations, the only one that we consider relevant is the user's. For this reason, the user's participation, after the minimization process finishes, is paramount, because she must ultimately decide whether she is comfortable with the information to be sent.

The only information sent in the error report is $\mathcal{I}^a$. We believe that a graphical demonstration of $\mathcal{I}^a$, in a step-by-step fashion, is arguably better than a textual description of the input and, furthermore, we believe it to be the best way to show the user *exactly* what is sent in the error report. Once $\mathcal{I}^a$ becomes available the system recommends that the user replay it and if the user accepts to verify it:

- The system launches the above-described replayer component;
- The replayer launches the application and its main window becomes visible to the user;
- The replayer reproduces each event of $\mathcal{I}^a$ in the GUI in the correct order, while the user can visualize each event being triggered;
- The replayer introduces a small delay between each event being reproduced, so that it becomes very easy for the user to follow;
- When the failure is reproduced, the procedure terminates.

Figure 7 illustrates the replaying of the minimized input for the example of Fig. 1. Once the user visualized each event in the error report, she can decide if there is any reason for concern. The user is asked for permission to transmit the error report to the maintenance team and, of course, she may accept or reject the request. In the latter case, the error report and respective log are deleted.

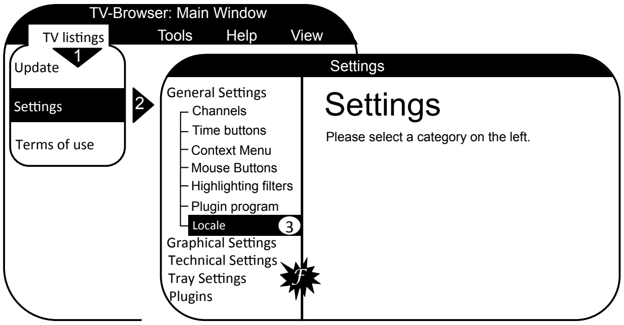


Fig. 7. Replaying the minimized input of the example in Fig. 1: The user visualizes, step-by-step, the triggering of the three events required to induce $\mathcal{F}$.

Developer Interface. The developer is notified once error reports are received, in which case she is presented with the list of error reports received. The developer may select any of the available logs and replay the respective failure-inducing (and minimized) input, exactly the same way the user did before authorizing the transmission of the error report. It is also worth mentioning that replaying $\mathcal{I}^a$ instead of $\mathcal{I}^u$, also has the potential of substantially facilitating the search for the causes of the failure. In fact, input minimization was originally created with this purpose [16]. Finally, after fixing the software error, the developer archives the error report.

5 Evaluation

In this section we describe our experiments which address the following research questions:

- In practice, how far is the 1-dialog-minimal input achieved by GUIMIN from the global minimum?
- Is GUIMIN applicable to real applications containing real (reported) bugs?
- Is it worth adopting a minimization algorithm that takes into account the structure of the input?
- How many tests are required to produce a 1-dialog-minimal solution as in Definition 4?
- Does GUIMIN minimize sensitive information and, does sensitive information remain in the minimized input? In line with the latter, can GUIMIN be complemented with previous obfuscation systems?

5.1 Test Subjects

This evaluation uses six different applications, selected because they are used to manage user sensitive private information, due to their high popularity and availability of real bugs. Our testbed includes two text editors, *jEdit* [31] (2366 classes, 115 kLOC) and *Lexi* [32] (156 classes, 6901 LOC) and two mail clients, *Columba* [33] (3356 classes, 108 kLOC) and *Pooka* [34] (854 classes, 48 kLOC), with real reported bugs [6, 35–37]. Our testbed also includes the *TV-Browser* [26] tv guide (2491 classes, 110 kLOC) of the motivational example of this paper and *JKeyring* (3 classes,1225 LOC), a password manager created and maintained by us. In its earlier phases *JKeyring* contained a bug triggered by the use of special characters when adding a new entry to the key chain.

5.2 Configuration

The experimental platform used in this study is a machine running the MacOS X Lion operating system, with a 2.5 GHz Intel Core i5 processor and 4 GB of memory.

Event Filtering. As mentioned in Sec. 4, the developers are responsible for configuring the event mask of the recorder. Thus, for this evaluation, we configured it to filter several types of events that do not trigger any application behavior. *We analyzed each of the benchmarks in this evaluation to determine which types of event do not trigger any reaction in the respective GUIs.* The reason for this is that we do not want to pollute the measurements that we are about to present, with events that cannot possibly influence the triggering of a failure. For example, the subjects described in Sec. 5.1, do not react to events such as MouseMoved and, this type of event fills the logs with millions of entries, even in short executions. The recording process of the execution scenarios considered in this evaluation includes solely events that contribute to the flow of the execution.

Log Size. The applications were used normally, while monitored by the recorder, before the respective bugs were triggered. For each of the six above-mentioned subjects, the recorder logged approximately 1000 events. The events logged are mostly MouseClicked and KeyTyped events. Then we used GUImin to minimize the content of the logs, in order to obtain a 1-dialog-minimal input, as in Definition 4.

Repeated tests. If a test is equivalent to a previously conducted one (the events to be replayed are the same), it cannot possibly help the minimization process. For example, if we split the input in two sublists of equal size, the tests of ΔMin are the same as the tests of greedy∇Min. Thus we uniquely identify each sequence of events to be tested, to make sure that the same test is not performed twice.

5.3 Results

In this section we present the results of the experiments conducted. We start by assessing the minimality of GUImin for the above-described subjects. Then, we overview the performance of GUImin and compare it against *ddmin*. Finally, we provide a quantitative and a qualitative evaluation of the outputs of GUImin, in the following sections.

5.3.1 Minimization Quality

In our experiments, we analyzed how far the *1-dialog-minimal* input achieved by GUImin is from the optimal solution, that is, the *global minimum*. To do so, we manually analyzed the bugs described in Sec. 5.1, at the source code level, to understand how they are triggered. Then, we calculated what the smallest possible list of events is that can induce the respective failures. This enabled us to conclude that, in all of our experiments, the *1-dialog-minimal* input achieved by GUImin coincided with the *global minimum*. However, it is important to note that GUImin guarantees a *1-dialog-minimal* input and not a *global minimum* input.

5.3.2 Scalability

Figure 8 presents the elapsed time of GUImin for the scenarios specified in Sections. 5.1 and 5.2. The GUImin algorithm took at most 45 minutes to complete. Furthermore, except for the *TV-Browser* subject, GUImin took under 25 minutes to complete.

These results highlight the main difference between minimizing graphical-based input and textual-based input. To test a sequence of graphical input events, we have to trigger each event at a time, in the correct order. Furthermore, the target widget may require a non-negligible time interval to become visible (e.g. if it belongs to a different window). This is a different and slower paradigm than minimizing textual based input, and for this reason, GUImin requires minutes to complete, instead of seconds. Nevertheless, as mentioned in

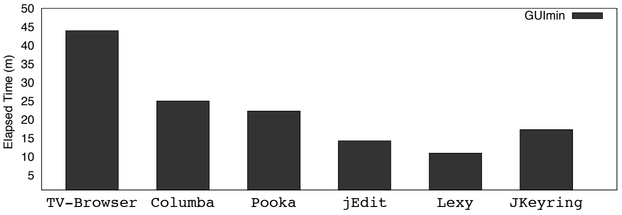


Fig. 8. Time (in minutes) required by GUIMIN to minimize the logs of 1000 events, of each benchmark presented in Sec. 5.1.

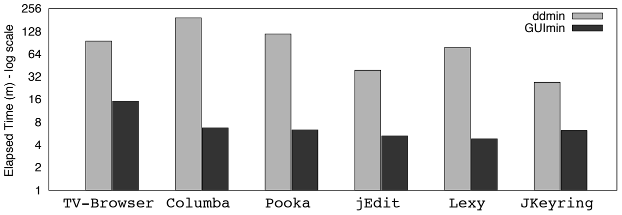


Fig. 9. Comparing the elapsed time (in minutes) of GUImin against ddmin, for each benchmark presented in Sec. 5.1 and for smaller log sizes (100 events). The y-axis is in log scale.

Sec. 4, users are not required to wait for GUIMIN to finish because GUIMIN is meant to run during idle periods. Therefore we consider these elapsed time results to be reasonable.

The *TV-Browser* experiment was by far the most time consuming. The graphical user interface of *TV-Browser* has a lot fewer text areas than the other five subjects. Therefore, in the *TV-Browser's* experiment, $\mathcal{I}^u$ is mostly composed of mouse events. As a consequence, the recorder does not frequently group recorded events, meaning that there are more widget dialogs with fewer events, instead of fewer containing more logged events. This also explains why the *jEdit* and *Lexi* took the least amount of time. Knowing that these two subjects are text editors, the widget dialog representing the main text input area contains most recorded events. Larger clustering of events logged within widgets, means less widget dialogs (whereas the opposite is to have one logged event for each widget dialog). Intuitively, if $|\mathcal{W}^u|$ gets smaller, $\mathcal{W}$MIN works faster. This particularity is analyzed with more detail further in Sec. 5.3.3.

We also performed a second experiment, on a smaller scale, to compare GUIMIN against *ddmin*, for graphical-based input. The goal was to validate our claim that *ddmin* is likely not to scale in typical scenarios. We performed this experiment for the same subjects but with a substantially smaller input: approximately 100 events. This is because *ddmin* requires an unreasonable amount of time to minimize inputs composed by 1000 events. The results are shown in Fig. 9. Note that the y-axis is in log scale. The completion time of *ddmin* is at least 1 order of magnitude greater than GUIMIN's. For example, in the Columba test, *ddmin* required more than 3 hours to complete. Although the minimization process is supposed to take place during idle periods of the user's machine, the completion time required by *ddmin* may well exceed those periods if a larger amount of input is recorded. The fact that *ddmin* required hours to minimize such small logs, suggests that *ddmin* is not suitable to minimize the usual large logs, composed of thousands of events.

5.3.3 Quantitative evaluation

Figure 10 depicts, for all six benchmarks, the input size reduction obtained as the sequence of tests progresses, in order to demonstrate GUIMIN's potential to minimize error reports.

TV-Browser. This program required 78 tests to achieve a *1-dialog-minimal* input. The majority of the minimized input was trimmed in the first 10 tests. Once the first invocation of $\mathcal{W}$MIN finishes, there are no tests for $\mathcal{W}\mathcal{E}$MIN to perform. The vertical line represents the beginning of the loop at lines 3 to 8 of Algorithm 3. The *TV-Browser* was the subject that required more tests and consequently was the subject for which GUIMIN took more time. This is an example of a test case in which $|\mathcal{W}^u|$ is closer to $|\mathcal{I}^u|$. Thus, a larger and more complex data structure is generated by the recording procedure (see Algorithm 1).

Email Clients Executing an email client is likely to generate a significant amount of *key typed* events, unlike TV-Browser. Typically, a balance between mouse operations and key events is expected. Mouse operations are used to browse the application and the recipients, whereas key events are used to compose messages.

Fewer tests are required to achieve a *1-dialog-minimal* input for these email applications. The *Key-Typed* events that are used to compose the message content are grouped in the corresponding text area widget dialogs. Thus $\mathcal{W}$MIN has less widget dialogs to minimize. We can observe this by comparing it with the plot of TV-Browser: the last phase of the minimization, highlighted by the vertical line in the plots, starts a lot earlier for the mail clients.

In the Pooka subject, no tests are performed in the last phase. Hence, the vertical line in the plot of Pooka coincides with the last test performed. This is because,

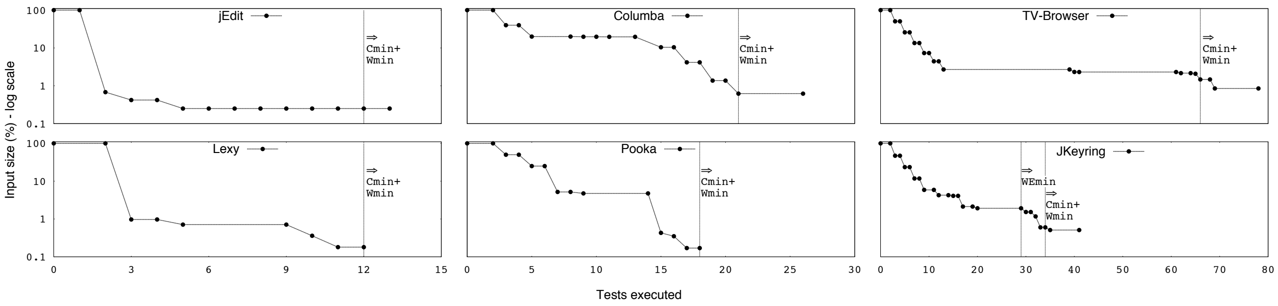


Fig. 10. Plots showing the file size reduction with each test performed by GUIMIN, on all 6 benchmarks. The y axis is in log scale. The x axis shows three scales: 0-15 for the leftmost plots; 0-30 for the middle plots; 0-80 for the rightmost plots. The points in each plot, represent the tests in which the input size was reduced.

as mentioned earlier, our implementation of GUIMIN keeps track of all tests already performed, which was the case for the last phase of this experiment.

Text Editors These subjects are examples of scenarios in which $|\mathcal{W}^u|$ is considerably lower than $|\mathcal{I}^u|$. Thus they are more likely to be less demanding to $\mathcal{W}$MIN. Given that the bugs in these subjects are graphical based, GUIMIN is able to complete its procedure resorting to fewer tests, as we can see in the jEdit and Lexi plots of Fig. 10. In both cases, a single test minimized 99% of the input. Like in the case of *Pooka*, few or no tests were executed in the last phase, for the same reasons.

$\mathcal{F}$-inducing text area The *JKeyring* subject is important in this evaluation, not only because of the potentially sensitive input it manages, but also because it is the only subject in our evaluation in which $\mathcal{F}$ is triggered by the content of a text area. Consequently, $\mathcal{WE}$MIN is required to minimize such content.

5.3.4 Qualitative Evaluation

The previous section shows that GUIMIN is capable of minimizing up to 99% of the user input. However, it is important to evaluate whether the minimized information was indeed private information and, most importantly, whether $\mathcal{I}^a$ contains sensitive information.

We manually analyzed our 6 benchmarks to access the information in the recorded logs that could be considered sensitive and we found the following types of sensitive information:

- *Identity Information (II)*;
- *Credential Information (CI)*;
- *Location Information (LI)*;
- *Behavioral Information (BI)*;
- *Confidential Data (CD).*

	II	CI	LI	PBI	CD
jEdit	21	0	12	0	1137
Lexi	21	0	12	0	1091
Columba	269	15	252	34	767
Pooka	62	15	36	14	891
TV-Browser	0	0	8	213	0
Jkeyring	271	676	0	0	0

Table 1. How many bytes of each category are found in the logs of the user input $\mathcal{I}^u$, for each benchmark.

	II	CI	LI	PBI	CD
jEdit	0	0	0	0	0
Lexi	0	0	0	0	0
Columba	205	0	204	0	0
Pooka	0	0	0	0	0
TV-Browser	0	0	0	0	0
Jkeyring	0	1	0	0	0

Table 2. How many bytes of each category are found in the logs of the user input $\mathcal{I}^a$, for each benchmark.

Before Minimization. Table 1 shows the number of instances of each type of information found within the recorded logs of our subjects.

- The jEdit and Lexi applications were used to edit a confidential (*CD*) text (1137 and 1091 bytes, respectively), signed with the author's name (*II*) and affiliation (*LI*).
- It is normal for mail clients to be used to exchange messages and to include features such as address books, calendars and mail server authentication procedures, amongst others. The messages exchanged using Columba and Pooka totaled 767 and 891 bytes of confidential data, respectively. Behavioral information (*BI*) was also found in the messages and in the interaction between the user and the calendar. Furthermore, the messages were also signed with the author's name (*II*) and affiliation (*LI*) and the address book of Columba loaded contacts from a csv file containing 205 bytes of identifiable information and 204 bytes of location information. Finally, the mail server authentication dialogs discloses credentials of the user (*CI*).

- TV-Browser asks the user for her location (LI) and also for information such as favorite movies, shows, actors amongst others. The execution also involved the subscription of channels, browsing TV shows schedules and more, which totaled 213 bytes of behavioral information (BI).
- Finally, JKeyring manages user credentials, namely <username , password> pairs. This test resulted in 271 bytes of *Identity Information (II)* and 676 bytes of *Credential Information (CI).*

After minimization. Table 2 clarifies whether the minimized input $\mathcal{I}^a$ still contains such types of information. As mentioned in Sec. 5.3.1, GUIMIN achieves the *global minimum* in these six test subjects, meaning that $\mathcal{I}^a$ includes *only* the steps-to-reproduce $\mathcal{F}$. In the test subjects of jEdit, Lexi, Pooka and TV-Browser, the steps-to-reproduce are completely innocuous, therefore GUIMIN was able to minimize 100% of the sensitive information in the logs. Unfortunately, Columba and Jkeyring require sensitive information to reproduce $\mathcal{F}$ and for this reason GUIMIN was unable to minimize every byte of sensitive information, leaving 30.59% of sensitive information in the log of Columba and less than 1% in the log of JKeyring.

- In the case of Columba, the bug is located in the contacts import wizard: the user browses the file system and selects a *csv* file containing her contacts. The bug is triggered if that file is malformed. Knowing that the *csv* file is external and is not graphical-based input, GUIMIN is not used to minimize its content. The csv file contains identity and location information, such as name, email and street address.
- The JKeyring application attempts to estimate the quality of the password inserted by the user, to inform her whether the password is strong. To do that, JKeyring iterates the password and checks if the decimal reference of each character is contained in pre-determined ranges (e.g. if $48 \leq c \leq 57$ then c is a digit), thereby counting the occurrences of each type of character (uppercase/lowercase, digits, letters, special characters) in the password. In earlier versions of JKeyring: *i*) an uncaught exception is thrown if a character is not contained in any of the pre-determined ranges; *ii*) the ranges were too restrictive at first, causing JKeyring to throw an exception when certain characters (that should be valid) are inserted.

By inserting one "out of range" character: *i*) all dialogs between the user and JKeyring are minimized by $\mathcal{W}$MIN except for the last; *ii*) $\mathcal{WE}$MIN minimizes every character except for the $\mathcal{F}$-inducing one. This means that $\mathcal{I}^a$ reveals one character of the password.

In both of these cases, GUIMIN cannot be used to minimize the remaining sensitive information. However, our system allows for the integration with existing obfuscation systems.

5.4 Integrating with Obfuscation Systems

We integrated GUIMIN with an obfuscation system, to test whether they can complement each other: GUIMIN minimizes plenty of sensitive graphical-based information that obfuscation systems do not (see Sec. 3), and obfuscation systems are capable of obfuscating the remaining input that may still include some sensitive information. The Columba and JKeyring subjects are particularly interesting for this part of the evaluation, considering that GUIMIN does not minimize the csv failure-inducing file of Columba, nor the failure-inducing character left in a text field of JKeyring. On the other hand, the state-of-the-art obfuscation systems [5–8, 10] have the potential to obfuscate those inputs by replacing them with alternative failure-inducing inputs.

Amongst all the systems described in Sec. 3 we chose our previous work REAP [10]. To perform symbolic execution on graphical-based applications we resorted to *JPF-AWT* [38]. We created a module in our system that automatically creates, after the minimization process, the script file required to execute *JPF-AWT*. Given this, we were able to integrate REAP with our system. We configured REAP with the parameters that provided the best results, privacy-wise, in the respective paper, that is the *REAP-BAG* algorithm within an unbounded search [10].

After GUIMIN minimized most sensitive information in $\mathcal{I}^u$, REAP was invoked to obfuscate: *i*) the content of the csv file; *ii*) the remaining character of the last password inserted by the user. After the obfuscation, there was no sensitive information left disclosed in the logs. More specifically, there was no similarity (a.k.a. residue [6]) between the original comma-separated values of the file and the values used to replace them and the remaining character of the JKeyring test was successfully replaced by an alternative $\mathcal{F}$-inducing character.

According to the leakage metric proposed by the work of Castro et. al. [5], the amount of bits revealed was less than 1%, for the Columba test case and less than 5% for the JKeyring test.

6 Discussion: Privacy and Utility

The proposed system guarantees that a 1-dialog-minimal input is found but, as mentioned in the previous section, it does not quantitatively evaluate what was minimized and what was left in the error report. However, the proposed system ensures that the user can verify, in a step-by-step fashion, each input that is being sent in the report, thereby visualizing exactly what a person with access to the error report will visualize. We believe that this enables the user to perform her own qualitative evaluation of the information included in the error report and consequently to decide whether her information is safe. Note that *the only data in the error report* is the replayable input $\mathcal{I}^a$.

One important question is whether the proposed system trades utility for privacy:

- By further informing the user on what information is being sent in the error report, we may dissuade her from submitting the error report, thereby compromising the system's utility and consequently the software's maintenance. However, we argue that utility should not be achieved at the cost of privacy loss.
- On the other hand, by further informing the user on what information is in the error report, we may show/assure the user that her data is safe, thereby persuading her to authorize the transmission of the error report, thus increasing system utility.
- Given an $\mathcal{F}$-inducing input, the maintenance team's job is easier than if it had not been provided with the steps to reproduce $\mathcal{F}$ as well as the minimized input $\mathcal{I}^a$ (instead of $\mathcal{I}^u$).

Similarly to all previous privacy enhancing error reporting systems, the proposed system regards only user input. Although there may exist other sources of sensitive information besides user input (e.g. GPS coordinates or IP address). The system would have to be extended to handle the programming calls that provide this information to the applications.

Furthermore, in order to guarantee the reproduction of the observed failure for all possible scenarios, one has to log all sources of non-determinism besides the input of the user. However, since we are solely concerned with privacy, other sources of non-determinism besides user input are generally not privacy sensitive and were not considered.

7 Conclusions and Future Work

This work proposed a new privacy-enhancing system that addresses the privacy concerns of error reporting systems. State-of-the-art systems do not cope with graphical input. Consequently, a substantial amount of private information may get included in the error reports sent to the maintenance teams.

The proposed system minimizes the disclosure of input in error reports of GUI-based applications. Previous input minimization algorithms are likely to underperform when minimizing graphical-based input, as they disregard the structure of the graphical user interface. We presented a new input minimization algorithm, optimized for applications with graphical user interfaces. The results of the conducted experimental study suggest that GUIMIN is suitable to minimize input in GUI-based applications. We also implemented integration capabilities that allow for input obfuscation systems to be employed alongside our system. The proposed system is scalable and the user is not required to wait for the procedure to complete, as this system is meant to be used during idle periods.

In the future we will explore the application of the proposed system in mobile devices, as their interfaces are purely graphical. Furthermore, we could even envisage a user interface for our system where pending error reports and the corresponding user input can be shown to the user and she can determine whether to let the input minimization continue or whether the privacy obtained so far is enough for the error report to be sent. Our performance results below support this approach since the biggest privacy gains are obtained in the initial phase of the minimization process.

8 Acknowledgements

We thank our colleagues from the distributed systems group for their valuable help, especially Rodrigo Rodrigues, Nuno Santos and Nuno Machado. We also thank James Clause for providing us with the Columba test case for our evaluation. We are also greatful for the help of Kerry Parker. Furthermore, this work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT), via the projects UID/CEC/50021/2013.

References

[1] Nat. Inst. of Standards and Tech., Software Errors Cost U.S. Economy $59.5 Billion Annually. NIST News Release http://www.nist.gov/director/planning/upload/report02-3.pdf. 2002.

[2] Zhivich, M.; Cunningham, R. The Real Cost of Software Errors. IEEE Security & Privacy. 2009; pp 87–90.

[3] Cambridge University, Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year http://www.prweb.com/releases/2013/1/prweb10298185.htm. 2013.

[4] McLaughlin, L. Automated bug tracking: the promise and the pitfalls. IEEE Software. 2004; pp 100 – 103.

[5] Castro, M.; Costa, M.; Martin, J.-P. Better Bug Reporting with Better Privacy. Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA, 2008; pp 319–328.

[6] Clause, J.; Orso, A. Camouflage: Automated Anonymization of Field Data. Proceedings of the 33rd International Conference on Software Engineering. New York, NY, USA, 2011; pp 21–30.

[7] Wang, R.; Wang, X.; Li, Z. Panalyst: Privacy-aware Remote Error Analysis on Commodity Software. Proceedings of the 17th Conference on Security Symposium. Berkeley, CA, USA, 2008; pp 291–306.

[8] Andrica, S.; Candea, G. Mitigating Anonymity Challenges in Automated Testing and Debugging Systems. Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13). San Jose, CA, 2013; pp 259–264.

[9] Louro, P.; Garcia, J.; Romano, P. MultiPathPrivacy: Enhanced Privacy in Fault Replication. Dependable Computing Conference (EDCC), 2012 Ninth European. 2012; pp 203–211.

[10] Matos, J.; Garcia, J.; Romano, P. *Programming Languages and Systems*; Lecture Notes in Computer Science; Springer Berlin Heidelberg, 2014; Vol. 8410; pp 453–472.

[11] Snelting, G. Combining Slicing and Constraint Solving for Validation of Measurement Software. Proceedings of the Third International Symposium on Static Analysis. London, UK, UK, 1996; pp 332–348.

[12] De Moura, L.; Bjørner, N. Z3: An Efficient SMT Solver. Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg, 2008; pp 337–340.

[13] Dutertre, B.; de Moura, L. *The Yices SMT solver*; 2006.

[14] Barrett, C.; Tinelli, C. CVC3. Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). 2007; pp 298–302, Berlin, Germany.

[15] Prud'homme, C.; Fages, J.-G.; Lorca, X. Choco3 Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2014.

[16] Zeller, A.; Hildebrandt, R. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering. 2002; pp 183–200.

[17] Park, S.; Zhou, Y.; Xiong, W.; Yin, Z.; Kaushik, R.; Lee, K. H.; Lu, S. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. New York, NY, USA, 2009; pp 177–192.

[18] Altekar, G.; Stoica, I. ODR: Output-deterministic Replay for Multicore Debugging. Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. New York, NY, USA, 2009; pp 193–206.

[19] Machado, N.; Romano, P.; Rodrigues, L. Lightweight cooperative logging for fault replication in concurrent programs. Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. 2012; pp 1–12.

[20] Huang, J.; Zhang, C.; Dolby, J. CLAP: Recording Local Executions to Reproduce Concurrency Failures. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, 2013; pp 141–152.

[21] Broadwell, P.; Harren, M.; Sastry, N. Scrash: A System for Generating Secure Crash Information. Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12. Berkeley, CA, USA, 2003.

[22] Gupta, N.; He, H.; Zhang, X.; Gupta, R. Locating Faulty Code Using Failure-inducing Chops. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. New York, NY, USA, 2005; pp 263–272.

[23] Shakya, K.; Xie, T.; Li, N.; Lei, Y.; Kacker, R.; Kuhn, R. Isolating Failure-Inducing Combinations in Combinatorial Testing Using Test Augmentation and Classification. ICST. 2012.

[24] Artho, C. Iterative delta debugging. International Journal on Software Tools for Technology Transfer. 2011; pp 223–246.

[25] Yu, K.; Lin, M.; Chen, J.; Zhang, X. Practical Isolation of Failure-inducing Changes for Debugging Regression Faults. ASE. 2012.

[26] TV-Browser, http://www.tvbrowser.org/.

[27] TV-Browser, http://hilfe.tvbrowser.org/viewtopic.php?f=14&t=11689&hilit=reproduce.

[28] Cook, S. A. The Complexity of Theorem-proving Procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing. New York, NY, USA, 1971; pp 151–158.

[29] Wellnomics, An Analysis of Computer Use Across 95 Organisations in Europe, North America and Australasia. 2007; http://www.wellnomics.com/assets/Uploads/WorkPace/News/Wellnomics-white-paper-Comparison-of-Computer-Use-across-different-Countries.pdf.

[30] Matos, J.; Coracao, N.; Garcia, J. Record and Replay GUI-Based Applications with Less Overhead. IEEE International Symposium on Software Reliability Engineering Workshops. 2014; pp 353–358.

[31] jEdit, http://www.jedit.org.

[32] Lexi, http://sourceforge.net/projects/lexi/.

[33] Columba, http://sourceforge.net/projects/columba/.

[34] Suberic, http://www.suberic.net/pooka/.

[35] jEdit, http://sourceforge.net/p/jedit/bugs/3776.

[36] Lexi, http://sourceforge.net/p/lexi/bugs/13/.

[37] Lexi, http://sourceforge.net/p/pooka/bugs/33/.

[38] Mehlitz, P.; Tkachuk, O.; Ujma, M. JPF-AWT: Model checking GUI applications. Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on. 2011; pp 584–587.

Appendices

A Proof of Proposition 1

Proof. The last two invocations of GUIMin are to $\mathcal{C}$Min and $\mathcal{W}$Min. These last two procedures end (each) with an invocation to greedy∇Min with $n = 1$, performing Replay$(\mathcal{E}^a, w_i.\mathcal{E}, \mathcal{F}), \forall w_i \in \mathcal{W}^a$ and Replay$(\mathcal{E}^a, \epsilon_i, \mathcal{F}), \forall \epsilon_i \in \mathcal{E}^a : \epsilon_i.next \neq \emptyset$, respectively. GUIMin finishes when *count'*$\geq$ *count*, which is only true when:

$$\text{Replay}(\mathcal{E}^a, \epsilon_i, \mathcal{F}) = false, \forall \epsilon_i \in \mathcal{E}^a : \epsilon_i.next \neq \emptyset \quad (1)$$

$$\text{Replay}(\mathcal{E}^a, w_i.\mathcal{E}, \mathcal{F}) = false, \forall w_i \in \mathcal{W}^a \quad (2)$$

which follows Definition 4. □

B Proof of proposition 2

Proof. The LazyMin has a best case scenario of

$$log_2(|L|) \quad (3)$$

in which L is the list passed as a parameter. The best case scenario for GUIMin is when the first invocation to $\mathcal{W}$Min reproduces $\mathcal{F}$ in all tests performed, thus totaling:

$$log_2(|\mathcal{W}^u|) \quad (4)$$

tests performed, which leaves one widget dialog w_i inside one container dialog. Then $\mathcal{W}\mathcal{E}$Min is invoked to minimize the logged events held by w_i and is successful in every test:

$$log_2(|w_i.\mathcal{E}|) \quad (5)$$

The final verification invokes $\mathcal{C}$Min that has no tests to perform and $\mathcal{W}$Min again, performing one successful test, totaling:

$$log_2(|\mathcal{W}^u|) + log_2(|w_i.\mathcal{E}|) + 1 \quad (6)$$

□

C Proof of Proposition 3

Remark. For a matter of simplicity, we denote with $\mathcal{C}^*$ the set of all container triggering events in $\mathcal{I}^*$. Each recorded widget $w_i \in \mathcal{W}^u$ contains at least one recorded event $\epsilon_i \in \mathcal{E}^u$ and each recorded event is contained by only one widget $w_i \in \mathcal{W}^u$. Analogously, each container opened by each $\epsilon_i \in C^u$ contains at least one recorded widget $w_i \in \mathcal{W}^u$. Therefore, the following equation is always valid:

$$|\mathcal{I}^u| = |\mathcal{E}^u| \geq |\mathcal{W}^u| \geq |\mathcal{C}^u| \quad (7)$$

Proof. The worst possible cost of invoking any of the algorithms $\mathcal{C}$Min, $\mathcal{W}$Min and $\mathcal{W}\mathcal{E}$Min is consistent with the scenario in which every test fails to reproduce $\mathcal{F}$, thereby totaling the amount of tests performed to:

$$2 + 4 + 8 + \ldots + 2|L| = 4|L| \quad (8)$$

where L is the list passed as a parameter to LazyMin. Therefore, the cost of lines 2 and 3 in Algorithm 3 is at most:

$$4|\mathcal{W}^u| + \sum_{i \leq |\mathcal{W}^u|}^{i=1} 4|w_i.\mathcal{E}| \quad (9)$$

Then, the proposed algorithm attempts to guarantee 1-Dialog-minimality. The worst possible scenario for this phase is when $\mathcal{C}$Min and $\mathcal{W}$Min are unsuccessful to minimize a single dialog except for the very last ϵ_i complement test of $\mathcal{W}$Min, such that $\epsilon_i.next \neq \emptyset$: Replay$(\mathcal{E}^a, \epsilon_i, \mathcal{F}) = true$. Then, such scenario repeats itself until we are left with only one widget dialog. If we use Equation 7 to upper bound $|\mathcal{C}^u|$ to $|\mathcal{W}^u|$, we get:

$$4|\mathcal{W}^u| + \sum_{i \leq |\mathcal{W}^u|}^{i=1} 4|w_i.\mathcal{E}| + \sum_{i < |\mathcal{W}^u|}^{i=0} 8(|\mathcal{W}^u| - i) \quad (10)$$

As already mentioned, we get $\mathcal{E}^*$ by concatenating $w_i.\mathcal{E} : \forall w_i \in \mathcal{W}^*$, thus we can simplify to:

$$4|\mathcal{W}^u| + 4|\mathcal{E}^u| + \sum_{i < |\mathcal{W}^u|}^{i=0} 8(|\mathcal{W}^u| - i) \quad (11)$$

Again we use Equation 7 to upper bound $|\mathcal{W}^u|$ to $|\mathcal{I}^u|$:

$$4|\mathcal{I}^u| + 4|\mathcal{I}^u| + \sum_{i < |\mathcal{I}^u|}^{i=0} 8(|\mathcal{I}^u| - i) \quad (12)$$

Finally, the rightmost component of Equation 12 is a triangular number multiplied by 8, thus:

$$4|\mathcal{I}^u| + 4|\mathcal{I}^u| + 8\frac{|\mathcal{I}^u|(|\mathcal{I}^u| + 1)}{2} \quad (13)$$

that we can simplify to:

$$4|\mathcal{I}^u|^2 + 12|\mathcal{I}^u| \quad (14)$$

□