Pidgin Crasher: Searching for Minimised Crashing GUI Event Sequences

Haitao Dan, Mark Harman, Jens Krinke, Lingbo Li, Alexandru Marginean and Fan Wu

CREST, Department of Computer Science,
University College London, Malet Place, London, WC1E 6BT, UK.

Abstract. We present a search based testing system that automatically explores the space of all possible GUI event interleavings. Search guides our system to novel crashing sequences using Levenshtein distance and minimises the resulting fault-revealing UI sequences in a post-processing hill climb. We report on the application of our system to the SSBSE 2014 challenge program, *Pidgin*. Overall, our *Pidgin Crasher* found 20 different events that caused 2 distinct kinds of bugs, while the event sequences that caused them were reduced by 84% on average using our minimisation post processor.

1 Introduction

Graphical User Interface (GUI) programs react to non-deterministic event sequences, with the engineering consequence that the functionalities provided by the software may be invoked in unexpected ways, possibly leading to faults. In addition, software engineers need to protect the functionalities from complex user inputs including malicious attacks. This challenges testers to find unusual test sequences that may expose critical defects before they are experienced by users or exploited.

Current GUI testing primarily relies on manual and record-playback techniques [9, 12]. Even with a record-playback tool, GUI testing remains a time consuming procedure, as it is human-centric: testers need to manually search for interesting test sequences. Another problem is that the quality of the tests depends on the testers' experience and understanding of the program. The nature of GUI programs requires testers to explore an exponential number of interleavings of test sequences. This is usually impossible, so testers rely upon assumptions about the way the software will be used to constrain the test sequences that need to be explored.

In this paper we focus on automated search-based GUI testing for the SSBSE Challenge program *Pidgin*. *Pidgin* is a popular instant messaging program [11]. It is developed as an open framework, for which others can develop plugins to enrich its functionalities. Of course, such an open and pluggable architecture may also introduce security vulnerabilities, because plugins could be embedded with malware and exploited by attackers.

We introduce and present a system testing tool, *Pidgin crasher*, that is embedded in *Pidgin* and complements existing human-centric GUI testing. *Pidgin crasher* is a GUI testing tool in which search-based algorithms are applied in both on-the-fly test generation and in a post-processing test reduction phase. In order to generate effective crash sequences to reveal potential bugs, *Pidgin crasher* selects the next valid event to send by maximizing the Levenshtein distances to all previously discovered crashing sequences. This selection technique generates shorter sequences that reveal more bugs in *Pidgin* compared to a random sequence generation process.

In order to generate test sequences for *Pidgin*, we use a combination of a *Greedy* search and a simple hill climbing post processing phase. The *Greedy* phase generates incrementally longer test sequences, guided by the measurement of Levenshtein distance to previously encountered sequences. The hill climbing phase is a cleanup operation, similar to those used in Genetic Improvement to reduce the edit sequence [8]. It seeks to minimise the length of the crashing test sequences found in the *Greedy* phase. The overall approach promotes diversity among the set of crashing sequences found by our approach.

We compare our Greedy search with a random approach and a 'tabu random' (that is forbidden to revisit previously encountered sequences, but is not guided by Levenshtein distance). We call this tabu random approach *Blocked* since it is blocked from considering previously encountered sequences.

In our experiments, we run *Pidgin crasher* 1005 times on each of the three different configurations: *Random*, *Blocked* and *Greedy*. In every execution we found at least one crashing UI sequence. On average, crash sequences generated by our greedy approach are shorter than those generated by the blocked approach (17.5 vs 58.4 UI events).

Overall, we identified 20 different crashing points triggered by 12 different UI events. Further analysis shows that these crashes are caused by two different types of bugs which we term *Type-1* and *Type-2*. *Type-1* faults are those caused by failure to check for NULL pointers passed as actual parameters, while *Type-2* are those faults caused by *Pidgin* requiring the existence of some non-existent resource (e.g. a window or widget). Finally, in our crash sequence reduction experiments, we found that the hill climber reduces crashing sequences with an average reduction factor of 4.88-7.50 (84% on average).

Related work: To apply the search-based testing, a UI model representing the behaviour of the application under test is usually used to initialise the original tests. Much of the previous work focusses on automating GUI testing [1–4, 7, 10] with a model (manually generated or automatically synthesised), which is used to guide the test generation to follow common user patterns. The closest related work is the EXSYST approach [5]. Like EXSYST we use search-based techniques to find input sequences. However, we target crashing behaviour, whereas EXSYST targets coverage. Furthermore, EXSYST is guided by a state-based model, whereas *Pidgin crasher* does not require a model. In the way that new test sequences are derived from previous crash sequences, our approach is also similar to the concept of test regeneration proposed by Yoo and Harman [13].

2 Test Generation, Execution and Reduction

Pidgin crasher simply targets crashing behaviour, so it does not require a test oracle [6]. It is designed as an automatic testing plugin for *Pidgin* based on GTK+, which is a multi-platform toolkit for creating GUIs used by *Pidgin*. We are using low-level APIs so that the framework not only works on *Pidgin*, but can also be easily adapted to other GTK+ based programs.

On-the-fly GUI testing: *Pidgin crasher* conducts testing of GUI programs at the system level. We implement three different on-the-fly UI sequence generation approaches: *Blocked* and *Greedy* search to compare with *Random* search.

In the *Blocked* approach (Algorithm 1), *Pidgin crasher* repeats a procedure of randomly selecting a GTK+ widget (a UI element such as a menu or button from the GTK+ framework) and sending a random but valid signal[1] to *Pidgin* until a crash is observed. The whole process is dynamic, meaning the number of windows varies over time and the window-selecting step adapts to the changing number of windows. In order to avoid previously discovered crashing points, a block list is loaded at the beginning of the random search process. The algorithm records the crashing sequence by writing every emitted signal into a log file.

```
LoadBlockList();
victim = SelectTopWindow() ;
repeat
    Randomly keep victim or execute victim = SelectTopWindow() ;
    target = SelectWidget(victim) ;
    sig = SelectSignal(target) ;
    if not IsBlocked(sig) then
        WriteCrashSequence(sig);
        SendSignalByName(target, sig, ...);
    end
until a crash ;
```

Algorithm 1: Random search with block list

The *Greedy* search approach, on the other hand, uses the previously generated sequences to guide the selection of new signals. More specifically, suppose we have a set of signals, $\mathcal{S}$, and a set of crashing sequences previously generated $\mathcal{P} = \{S_1, ...S_n\}$ and each sequence S_i is an ordered string of signals $S_i = s_{i_1} s_{i_2} ...$, $s_{i_j} \in \mathcal{S}$. When $S_c = s_1 ... s_k$ is the current sequence of signals we have sent so far (but for which we have yet to encounter a crash), we select the next signal s_{k+1} by computing the furthest Levenshtein distance between the current sequence after the signal is appended and all previous sequence in $\mathcal{P}$. More formally,

$$\forall_{s_i \in \mathcal{S}} : M(\mathcal{P}, s_1 ... s_k s_i) \leq M(\mathcal{P}, s_1 ... s_k s_{k+1})$$

where $M(\mathcal{P}, S) = \min_{S_i \in \mathcal{P}} \{D(S_i, S)\}$ and where $D(x, y)$ is the Levenshtein distance between x and y.

[1] *Events* from the X server are turned into GTK-specific *signals* by GTK.

3 Experiments and Results

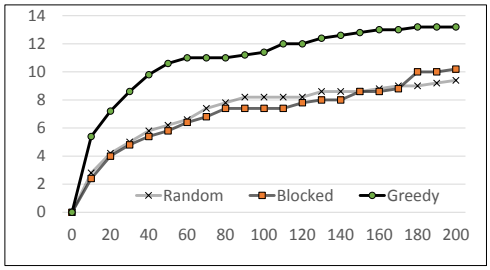
In this paper, we answer the following Research Questions (RQs):

RQ1 How effectively can *Pidgin crasher* find potential bugs?
RQ2 What are the coverage of crash points, convergence and redundancy of the sequences found by each of the three versions of *Pidgin crasher*?
RQ3 What are the kinds of faults found by *Pidgin crasher*?

We use *Pidgin crasher* to generate crashing sequences in each of its three different modes: *Random*, *Blocked*, *Greedy* search. In the *Random* mode, the next signal to send is randomly selected from all available signals, while the other two are those approaches described in Section 2.

Pidgin crasher is continuously invoked to produce 201 crashing sequences in each mode. We repeat the procedure 5 times, so *Pidgin crasher* is run for a total of 3015 ($201 \times 5 \times 3$) times. The sequences so-produced are minimised by our Hill Climbing process to remove redundancy. In the experiments, we repetitively send signals to trigger different functionalities via the special API call `g_signal_emit_by_name(GtkObject *object, const gchar *name, ...)` to which we pass the selected widget in `object`, the selected signal in `name`, and all arguments in the variable argument list are passed NULL. All experiments were run on Ubuntu 13.04 with debug versions of GTK+ 2.24.17 and Glib 2.38.0.

Answer to RQ1: According to the top-right table in Figure 1, the average lengths of the crashing sequences generated by the *Random*, *Blocked* and *Greedy* modes are 14.5, 58.4 and 17.5, respectively. In the same order, the maximum lengths are 131, 673 and 135. All three modes can find the shortest possible sequence with length 1 (Column *Min*). In the last column, it is shown that, on average, *Greedy* mode spends more time to generate 201 crashing sequences due to the calculations of Levenshtein distance. In summary, *Pidgin crasher* can effectively crash *Pidgin* in all three modes.
Answer to RQ2 (Coverage): The bottom table in Figure 1 lists the crashing points found by all runs of *Pidgin crasher*. Of the 9 columns in this table, columns 6, 7 and 8 report the number of times each crash point is discovered. In the *Random* approach, 11 out of 20 crashing points are covered, whereas *blocked* covers 13 and *greedy* covers 19. So both the *Blocked* and the *Greedy* search have a better coverage than the *Random* approach, while the *Greedy* search achieves the highest overall coverage, finding all but one of the crashing points found by all approaches.
Answer to RQ2 (Convergence): Figure 1 top-left shows the growth of the number of different crashing points found (average of 5 runs). Even though both *Blocked* and *Greedy* search find more crashing points than the *Random* approach, the *Greedy* search clearly converges more quickly than the *Blocked* approach.
Answer to RQ2 (Redundancy): In order to compare the redundancy of the crashing sequences generated by these approaches, we use the simple Hill Climbing to remove any irrelevant signals from the sequences. The results show that the crashing sequences from *Random*, *Blocked* and *Greedy* search can be



	Avg	Min	Max	Factor	Time (Sec.)
Random	14.5	1	131	4.88	1356
Blocked	58.4	1	673	7.50	2120
Greedy	17.5	1	135	5.91	4650

Crashed Function	Widget	Signal	Crash Location	#Crash			Type
				Rnd	Blk	Grd	
add_room_to_blist_cb	GtkLabel	move-cursor	gtkroomlist.c:250	1	1	3	2
gtk_editable_insert_text	GtkEntry	insert-at-cursor	gtkeditable.c:170	0	2	13	1
gtk_label_activate_link	GtkLabel	activate-link	gtklabel.c:5838	45	116	206	1
gtk_menu_set_child_property	GtkMenu	move-scroll	gtkmenu.c:926	0	1	0	1
gtk_notebook_real_switch_page	GtkNotebook	switch-page	gtknotebook.c:6142	20	39	52	1
gtk_path_bar_scroll_down	GtkMenu	move-scroll	gtkpathbar.c:803	0	0	1	2
gtk_path_bar_scroll_down	GtkButton	clicked	gtkpathbar.c:803	0	0	1	2
gtk_real_menu_item_toggle_size_request	GtkMenuItem	toggle-size-request	gtkmenuitem.c:1452	811	681	435	1
gtk_tree_model_get_valist	GtkTreeView	row-activated	gtktreemodel.c:1470	5	12	11	2
join_button_cb	GtkMenuItem	activate	gtkroomlist.c:265	0	0	1	2
join_button_cb	GtkButton	clicked	gtkroomlist.c:265	0	0	1	2
location_button_toggled_cb	GtkToggleButton	toggled	gtkfilechooserdefault.c:4662	0	0	1	1
menu_add_pounce_cb	GtkMenuItem	activate	gtkconv.c:1169	4	14	41	2
menu_add_pounce_cb	GtkMenuItem	activate-item	gtkconv.c:1169	6	18	51	2
menu_invite_cb	GtkMenuItem	activate	gtkconv.c:1250	12	7	46	2
menu_invite_cb	GtkMenuItem	activate-item	gtkconv.c:1250	12	5	49	2
purple_blist_node_get_type	GtkTreeView	row-collapsed	blist.c	0	0	1	1
purple_blist_node_set_bool	GtkTreeView	row-collapsed	blist.c	0	0	3	2
regenerate_options_items	GtkMenuItem	activate-item	gtkconv.c:3343	49	52	43	2
regenerate_options_items	GtkMenuItem	activate	gtkconv.c:3343	40	57	46	2

Fig. 1: **Experimental Results** – The upper lefthand subfigure shows convergence of the three approaches. The upper righthand figure reports summary statistics for the average, minimum and maximum sequence length and the average execution time produced by each of the three approaches and, in the fourth column, it reports the reduction in sequence length produced by the post-processing hill climb. The lower, larger, table reports the numbers, types and locations of faults found by each approach.

reduced by a factor of 4.88, 7.50 and 5.91 respectively. The *Blocked* approach generated the longest sequences with the highest redundancy (i.e. the greatest potential for minimisation).

Answer to RQ3: We inspected *Pidgin* to understand the reason for each crash. As a result, we manually categorised all crashing points into two types that reflect two difference classes of reason why *Pidgin* crashes at these points. These two 'types' of fault are reported in the the last column of the lower (larger) table in Figure 1.

A *Type-1* crash happens in the call-back function directly uses a NULL-pointer from the passed arguments to access memory without checking to ensure it is non-NULL. *Type-2* crashes also happen in call-back functions that makes an invalid assumption about the resources available in the current state. For example, function `menu_add_pounce_cb` opens a conversation window using a pointer fetched from function X which may return NULL. As there is no NULL check in the call-back function, X is assumed to always return a valid pointer, which, however, is violated in some scenarios, where the resource is simply unavailable.

4 Conclusions and Actionable Findings

Using *Pidgin crasher*, we identified two types of bug found caused by 20 different UI signals. According to our findings, we suggest that *Pidgin* return values from any function that may return NULL-pointers should be checked, and that GTK+ signal-emitting APIs that take variable argument lists such as `g_signal_emit_by_name` should be deprecated.

References

1. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Reverse Engineering Finite State Machines from Rich Internet Applications. *15th Working Conference on Reverse Engineering*, October 2008.
2. Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato. A toolset for GUI testing of Android applications. In *28th IEEE International Conference on Software Maintenance (ICSM)*, September 2012.
3. Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *27th IEEE/ACM International Conference on Automated Software Engineering*, 2012.
4. Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification and Reliability*, 16(1), March 2006.
5. Florian Gross, Gordon Fraser, and Andreas Zeller. EXSYST: Search-based GUI testing. In *34th International Conference on Software Engineering*, June 2012.
6. Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield, 2013.
7. Casper S. Jensen, Mukul R. Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis*, 2013.
8. William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEVC)*, 2014. To appear.
9. A.M. Memon. GUI testing: pitfalls and process. *Computer*, 35(8), August 2002.
10. A.M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10), October 2005.
11. Pidgin, the universal chat client. http://www.pidgin.im/, Accessed in 2014.
12. Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, June 2013.
13. S Yoo and M Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3), May 2012.