# GUI-Mimic, a cross-platform recorder and fuzzer of Graphical User Interface

Vincent Raulin, Pierre-François Gimenez, Yufei Han, Valérie Viet Triem Tong, Léopold Ouairy

# GUI-Mimic, a cross-platform recorder and fuzzer of Graphical User Interface

Vincent Raulin Pierre-François Gimenez Yufei Han
Valérie Viet Triem Tong Léopold Ouairy
CIDRE, CentraleSupelec, Inria, Univ Rennes, CNRS, IRISA, Rennes

**Abstract**

In program analysis, a fuzzing toolset is needed to automatically trigger software operations in a natural while efficient way. Especially in dynamic analysis of malware, such a toolset can help execute the suspicious files to unveil their malicious payloads hidden by other benign-looking behaviors. In the fields of software testing, this tool is necessary for triggering and testing the programmed functionalities. Nevertheless, there has not yet been an easy-to-use tool that works on Windows for the purpose of generating activity through the Graphical User Interface (GUI). To meet this requirement, in our work, we develop GUI-Mimic. It is designed to integrate some useful features for stimulating different types of software – mouse and keyboard recording, random mouse and keyboard inputs, editing, trimming, randomization, transformations – to deliver an easy-to-deploy GUI fuzzer over different Operating Systems.

*Keywords:* GUI, fuzzing, testing, recording, macro, randomization, human interaction

## 1 Introduction

Monkey testing is a software testing technique that feeds random inputs and actions into a target software to verify whether its behavior matches its specification. It is usually implemented for automated unit and reliability tests of applications. Recently, monkey testing has also been employed for dynamic analysis of malware [1, 2]. For example, Google deployed its security service Bouncer [2] back in 2012. Bouncer uses a random event generation to trigger malicious payloads of Android malware. Furthermore, in 2020, the Android Developer Package has also integrated monkey exercise tools for stress test [3]. This tool can also be used to detect abnormal behaviors of apps[4]. Besides Android, OneFuzz, a monkey testing software for Windows 10, was open-sourced in 2020 [5] by Microsoft as a stress test tool for its operating system.

However, the current monkey testing tools have their limits. In particular, most available fuzzing tools for Windows are restrained to text input from terminals or I/O files. The only tools that focus on GUI interaction are not for Windows and are often quite limited or hard to set up. On the other hand, even when looking for macro recording tools, none possess viable fuzzing capabilities or are also quite hard to use.

To solve these issues, we plan to extend the current monkey testing tools in the following perspectives:

- The system needs to interact through the GUI (and not through the command line) for simulating realistic user interaction.
- It will record a sequence of user actions and use the recorded sequences as working material (e.g., to repeat them, reorder them, add an event in response to another or apply custom transformations like time dilation).

- The recording system needs to be easy to start and stop.
- It must have randomization features since we want to fuzz the application.
- It must allow us to create usage scenarios to fuzz some components more efficiently.
- It needs to be portable to different devices and to multiple operating systems.

An extra feature that we want to achieve is deployability: indeed, we want to be able to export all of our recording, our scenarios, and transformations into files to be accessible by other users and by automated testing machines. This article presents GUI-Mimic, a toolset that implements all of these new features in order to reach our requirements of GUI input, ease to use, randomization for fuzzing and portability for our studies on dynamic analysis of malware.

## 2 Related work

We identified two kinds of tools that can only achieve some of the objectives we defined in the introduction.

**Macro recorders** are designed to record keyboard or mouse inputs and replay these actions. Most of them only allow replaying identical sequences, which is not suited for fuzzing tests since they require a lot of different sequences of actions, most generally random. Most of these recording tools like Mouse and Keyboard Recorder[6], Free Macro Recorder[7] or Mousekey Recorder[8] only do this simple record/playback task.

Nevertheless, Macro Recorder [9] has a useful feature that makes it closer to fuzzing: it allows for editing (trimming, recording, rearranging) and even some conditional structures to reorder the sequences. Unfortunately, saving a sequence requires setting many parameters for each event through dialog boxes, making the process much more complicated and time-consuming. For example, a five minutes recording can take half an hour to configure.

**Fuzzing test tools**, such as Zzuf fuzzer [10], Killerbeez [11], LibFuzzer [12], AFL [13], and FOE [14], often don't support GUI. There exist a few GUI fuzzers, though, but they have serious limitations. For example, the tools from fuzzingbook[15] are either adapted to web applications or require drivers to handle a specific application. AutoIt[16] is a scripting language dedicated to building fuzzing scenarios. It is powerful but requires extensive work to generate even simple scenarios. Some Android tools like GroddDroid[17] can systematically trigger reactions from the graphical elements. However, it is designed for Android only and benefits from Android's easy-to-access GUI (XML extensive UI description) that makes it inapplicable to Windows. Finally, Guifuzz[18] is a tool able to interact with the GUI of Windows applications in a random fashion. However, this project was abandoned at an early stage of development in 2019 and can only fuzz `Calc.exe`. One close work is *Monkey* [3] from Android, which is equipped with our expected functions but runs under Android only. Moreover, it does not allow for customizing actions/recordings.

In contrast, GUI-Mimic is a GUI-boosted fuzzing test tool that can record a user performing tasks, build scenarios with those recordings, and add a lot of randomization to reach adequate fuzzing capabilities.

## 3 Contribution

GUI-Mimic is written in Python and is freely available on `https://gitlab.inria.fr/vraulin/GUI-Mimic`.

One can use GUI-Mimic in a few steps: recording sequences of events, editing those sequences, creating a scenario, and finally playing back those scenarios. During the recording, the user records their interactions with a targeted software. When creating a sequence, it

is best to focus on one kind of interaction with the target software, like starting it, saving a file, etc. These sequences are then used as building blocks to create a scenario that will interact with the entire software. By creating chaining rules for all the recorded sequences, the user can explore complex interactions in a relevant way (for example, not saving a file before creating one). It is also possible to manually add events to create random or organized inputs from the mouse or the keyboard. Different transformations can also be applied to sequences of events. Eventually, the project can be exported for later editing, playing one of the scenarios, or deploying to other machines.

Besides, GUI Mimic is lightweight. All its default operations have a linear time and space complexity. Only custom transformations could get higher complexity. In our experiments, GUI Mimic didn't use more than 50MB of RAM while recording or playing.

In the following, we explain in more details how GUI Mimic works with the sequences of events. To illustrate its inner working, we will take the example of the creation of fuzzing scenarios for `Notepad.exe`.

### 3.1 Sequence of events

The sequences of events handled by GUI-Mimic are based on atomic events that are indivisible hardware events, such as a mouse movement or a released key on the keyboard. More precisely, we use six atomic events: Keyboard press, Keyboard release, Mouse move, Mouse press, Mouse release, and Mouse scroll. Besides, each atomic event includes some parameters, such as time of occurrence since the last event, coordinates of mouse events, key for keyboards events.

During the recording, the atomic events are stored in a list. Once the recording is done, these atomic events are summarized into more complex events. For example, a MousePress event followed by a MouseRelease event is merged with it into a MouseClick event, and several KeyboardPress and KeyboardRelease are merged into a KeyboardInput event. Besides, mouse movement creates a lot of atomics events MouseMove that each describes an infinitesimal movement. Therefore, a succession of MouseMove atomic events are transformed into a MouseStart and a MouseStop event that only records the movement's start and end points and not the path. Once this transformation is completed, the list of atomic events is replaced by a list that contains complex events (MouseClick, KeyboardInput, etc.) and atomic events that could not be merged into a complex one. The latter can happen, for example, when a MouseRelease event does not follow a MousePress event. Notice that this transformation is not reversible (for example, the exact path of the mouse is lost) and can be disabled. This transformation serves two goals: first, it helps keep the sequence short; second, it allows us to manipulate the sequence easily, as explained in the following.

After a recorded sequence has been saved, it can be modified by adding random events, removing some events from a sequence with filters (like removing all presses on Ctrl), or changing their timestamps (for example, to make the hotkeys combinations be pressed faster).

### 3.2 Recording and Playback

The following subsections describe the use of GUI-Mimic with its interactive prompt. GUI-Mimic can also be used with its graphical interface.

The recording system is made to be straightforward: the user presses a keyboard shortcut to start and stop recording events (this shortcut can be customized). Once the recording is finished, the user can extract the derived event sequence and trim it. To start recording for our Notepad example, once GUI-Mimic's interactive prompt has started, the user just needs to press a hotkey combination (Ctrl_left+Ctrl_right by default) to start recording. Now, everything the user does is recorded until they press these same hotkeys again. The user might want to record themselves starting the app through Windows' startup menu by typing "Notepad" for their first sequence. Once the window opens, they can resize it and then stop

recording by pressing their hotkeys again. Since there is not much to do in the starting sequence, the user can just extract it for refinement by typing:

```
starting = extract()
```

We instrument both the recording and the playback using a Python module called "Pynput". It allows for complete control of the inputs and outputs of the mouse and the keyboard. Once recorded, a sequence can be extracted and put in a "user_guide". This structure is an editing workstation. It saves a group of recordings (usually for the same application) and allows some editing for the whole application. The user can create one and register their first sequence by typing:

```
Notepad = user_guide()
Notepad.start_1 = starting
```

Imagine the user realizes that they should have clicked on the text zone in case their next action is to type text. They can add this task by typing:

```
click = MouseClick(2000000000, 100000000, 900, 700) # time to wait (2sec),
    duration of click (0.1sec), x and y positions on screen
Notepad.start_1.append(click)
```

In practice, the user probably want to record themselves making a specific task on the application, such as starting the application, closing it, opening a certain file, executing a certain functionality of the app. In the case of Notepad, the user would record some more sequences: *start*_2, *start*_3 to open the app in different ways, *save*_1, *save*_2 to save the file in two different ways, *open*_1, *open*_2, *open*_3, *close*_1, *close*_2, *new*_1, *new*_2... all of which are accessible through $Notepad. < name >$. Then the user will be able to create rules to replay these bits of recordings in a certain order, like a scenario detailing how the application can be used. It can be for example : start the application, then play a randomly chosen use case, then save the file under this name, then close the app or create a new file. The user can create multiple scenarios, all of which can contain some random actions.

For notepad, the user will create two scenarios:

```
Notepad.scenario_1 = user_scenario("start_1", "write_2", "save_2", "close_1")
Notepad.scenario_2 = user_scenario("start.*", "write.*", "save.*", "new.*",
    "write.*", "save.*", "open.*", "close.*")
```

In the first scenario, the user starts Notepad.exe, writes some text, saves the file, and closes the app with specific sequences. Scenario 2 is a bit more complex. The ".∗" means that the system will choose a random sequence in the user_guide that matches the corresponding regular expression. For example, "$start.*$" would choose randomly between *start*_1, *start*_2 and *start*_3.

After that, the user can reload a project, modify these sequences again, create new ones, create new scenarios, remove some, and play those scenarios. In the user's case, saving and reloading the project in the file *Notepad.pyt* (default extension) would look like this:

```
save("Notepad.pyt", Notepad)
Notepad = load("Notepad.pyt")
```

And for playback, the user can use these commands to execute these scenarios:

```
Notepad.simulate("scenario_1") # runs the first scenario
Notepad.simulate("scenario_1", "scenario_2") # runs the first or second scenario
sequence = Notepad.simulate() # runs a random scenario
```

Note that simulate will furthermore return the actual event sequence that has been played for analysis. It is important for the reproducibility of experiments, as there may be random variation in what is being played back. This feature is the focus of the next subsection.

### 3.3 Generation of realistic events

A major objective of our tool is to make the generated events as realistic as possible. To achieve that goal, a handful of random-based perturbation functions are included. They can be applied both during editing and during playback. This built-in transformation includes:

**Time dilation** To simulate different users typing and mouse speeds, the user can apply a global time dilation, like increasing the duration by 20%. The user can also add some noise to the timestamps to make it more random. It is especially useful on generated events, like keyboard text inputs. Such events have a fixed time interval between key presses.

GUI-Mimic is very precise in its playback, meaning that the difference between theoretical timestamps and the actually played timestamps is small, thanks to the Pynput library. Although not tested with real-time applications, it is far enough accurate for interfaces that are intended to be used by humans.

In the case of the starting sequence the user recorded for Notepad, the user may want to add some time noise. In particular, they may want the duration of the events to change slightly at each run, to avoid repetition, and so that the mouse move does not last exactly 2 seconds:

```
Notepad.start_1.schedule_transform(time_noise(variance = 0.3)) # Variance of 30%
    of added or removed time
```

This transform applies a Gaussian deformation to the waiting time of each event. For example, it could change an event of $2sec$ wait into $2 \times 1.25sec$ wait with 1.25 being the sample from the Gaussian distribution. The function $schedule\_transform$ registers this transformation to be performed at the time of playback (so different noise can be obtained in different playback of the same sequence). If, instead, the user wants their transformation to be done immediately, they should use:

```
Notepad._start_1 = Notepad.start_1.apply_transform(time_noise(0.3))
```

$apply\_transform$ will immediately apply this transformation and return a new event sequence. It is better to use $schedule\_transform$ in case of randomizing transformations since it will create a different version at each playback.

The function $schedule\_transform$ can store multiple transformations and apply them in the same order at execution.

**Customized transformations on the sequences** Our tool allows the user to apply user-customized transformations to the sequences. For example, the user may want to apply a time dilation on each event depending on its type. By describing their transformation depending on the target event, the user can achieve that goal.

For example, the user pressed the Ctrl key but only on the left side of their keyboard. They would prefer to use the right side. To do so, they should type the commands:

```
### Inside of a python file
def random_side(event):
  e = event.copy()
  if isinstance(event, (KeyboardPress, KeyboardRelease)) and event.key ==
      Key.ctrl_l:
    e.key = Key.ctrl_r
```

```
    return e

### After loading the function
Notepad.apply_transform(random_side)
```

Note that *apply_transform* and *schedule_transform* can also be used on *user_guide*, in which case the transformations will be applied on all the registered sequences.

**Generation of the new events without recording them** To goal of this feature is to give more fuzzing capabilities. The most useful one is to generate some keyboard input. It will generate a sequence of keyboard press and keyboard release events with fixed time intervals to type what the user wants to be written during execution. It also works with hotkeys, and a user-defined function can provide the string at execution

The user might have wondered what are the sequences "$write.*$" that we used in our scenarios defined in the previous subsections. We are going to declare them with generated events. In this case, we just need a single event in our sequence to generate a random text input from the keyboard:

```
Notepad.write_1 = user_sequence() # Empty sequence
Notepad.write_1.append(KeyboardInput(5, "This is a text file.", 6)) # after 5
    seconds, start typing this text, at a speed of 6 characters per second.

Notepad.write_2 = user_sequence()
Notepad.write_2.append(KeyboardInput(2, random_text, 8)) # where random text is a
    function that returns some random text when called
```

Note that in Section 3.2, we used "$MouseClick$", which is also a generative event.

**Randomization of mouse paths** In a summarized sequence, we don't store every atomic mouse move. Instead, we can use the detailed information of when and where the mouse starts and stops moving to create an adequate curved path. Our current model uses Bézier curves of degree 2. Their intermediate control point is placed upon a random discus distribution from a circle centered between the starting and ending points. This process ensures that the mouse path seems conducted by real human users. It also enables the adding of randomness into the mouse moving paths.

# 4 Conclusion

Our new fuzzing toolset is developed to cover a specific case of fuzzing which most other fuzzing tools failed to achieve. It focuses on GUI interaction, more precisely through both mouse and keyboard. It is designed to interact with apps mimicking what a real human user would do. At the same time, the toolset is easy to configure the testing scenarios. Its editing tools provide extensive coverage of fuzzing tests for a given app and make its use convenient. It is also equipped with randomization and data generation tools to increase its test effectiveness. There are still some features to be added, like integrating some computer vision tools to detect buttons on the screen and generating meaningful events in response. Finally, we plan to apply this tool to monitor and capture suspicious system calls for the dynamic analysis of malware.

# References

[1] Lucky Onwuzurike, Mario Almeida, Enrico Mariconti, Jeremy Blackburn, Gianluca Stringhini, and Emiliano De Cristofaro. A family of droids-android malware detection

via behavioral modeling: Static vs dynamic analysis. In 2018 16th Annual Conference on Privacy, Security and Trust (PST), pages 1–10, 2018.

[2] H.Lockheimer. Android and security, 2020.

[3] Application exerciser monkey. https://developer.android.com/studio/test/monkey.

[4] Hayyan Hasan, Behrouz Tork Ladani, and Bahman Zamani. Enhancing monkey to trigger malicious payloads in android malware. In 2020 17th International ISC Conference on Information Security and Cryptology (ISCISC), pages 65–72, 2020.

[5] J.Campbell and Mike Walker. New project onefuzz framework, 2020.

[6] http://www.robot soft.com. Mouse and keyboard recorder. https://www.robot-soft.com/mouse-keyboard-recorder.html.

[7] Free macro recorder. https://www.mjtnet.com/simple-macro-recorder.htm.

[8] Mousekey recorder. https://mousekeyrecorder.net/.

[9] Bartels Media. Macro recorder. https://www.macrorecorder.com.

[10] Sam Hocevar. Zzuf fuzzer. http://caca.zoy.org/wiki/zzuf, 2015.

[11] https://github.com/anon8675309. Killerbeez. https://github.com/grimm-co/killerbeez, 2018.

[12] Libfuzzer. https://llvm.org/docs/LibFuzzer.html.

[13] Michał Zalewski. American fuzzing lop. https://lcamtuf.coredump.cx/afl/, 2013.

[14] Software Engineering Institute. Foe. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=515311, 2013.

[15] The Fuzzing Book. Fuzzingbook. https://www.fuzzingbook.org/html/GUIFuzzer.html.

[16] Autolt. https://www.autoitscript.com/site/.

[17] Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, Jean-François Lalande, and Valérie Viet Triem Tong. Grodddroid. http://kharon.gforge.inria.fr/grodddroid.html, 2015.

[18] https://github.com/gamozolabs. Guifuzz. https://github.com/gamozolabs/guifuzz, 2019.