GUIFuzz++: Unleashing Grey-box Fuzzing on Desktop Graphical User Interfacing Applications

Dillon Otto
University of Utah

Tanner Rowlett
University of Utah

Stefan Nagy
University of Utah

***Abstract*—Desktop applications represent one of today's largest software ecosystems, accounting for over 96% of workplace computing and supporting essential operations across critical sectors such as healthcare, commerce, industry, and government. Though modern software is increasingly being vetted through *fuzzing*—an automated testing technique for large-scale bug discovery—a major component of desktop applications remains universally under-vetted: the *Graphical User Interface* (GUI). Existing desktop-based fuzzers like AFL++ and libFuzzer are limited to *non*-GUI interfaces (e.g., file- or buffer-based inputs), rendering them wholly incompatible with GUIs. Conversely, mobile app GUI fuzzers like Android's Monkey and iOS's XCMonkey rely on platform-specific SDKs and event-handling, rendering them fundamentally unportable to the broader, more complex landscape of desktop software. For these reasons, desktop GUI code remains largely under-tested, burdening users with numerous GUI-induced errors that should, in principle, be just as discoverable as any other well-fuzzed class of software bugs.**

This paper introduces GUIFuzz++: the first general-purpose fuzzer for desktop GUI software. Unlike desktop fuzzers that randomly mutate file- or buffer-based inputs, GUIFuzz++ exclusively targets GUI *interactions*—clicks, scrolls, key presses, window navigation, and more—to uncover complex event sequences triggering GUI-induced program errors. Central to our approach is a novel GUI *Interaction Interpreter*: a middle-layer translating fuzzer-generated random inputs into distinct GUI operations, enabling successful non-GUI fuzzers like AFL++ to be easily ported to testing GUIs. Beyond supporting today's most popular GUI development frameworks like QT, GTK, and Xorg, we introduce a suite of enhancements capitalizing on ubiquitous *Software Accessibility Technologies*, significantly boosting GUI fuzzing precision as well as GUI bug-finding effectiveness.

We integrate GUIFuzz++ as a prototype atop state-of-the-art GUI-agnostic fuzzer AFL++, and perform a large-scale ablation study of its fundamental components and enhancements. In an evaluation across 12 popular, real-world GUI applications, GUIFuzz++ uncovers 23 previously-unknown GUI-induced bugs—with 14 thus far confirmed or fixed by developers.

I. Introduction

Desktop software—applications deployed on personal computers or workstations—play an ever-growing role in our modern digital age, representing **over 96%** of workplace computing today [4]. As desktop software spans important domains like healthcare, commerce, industry, and government, its *correctness* directly impacts many of society's most critical services. Unfortunately, program bugs remain a significant challenge across today's desktop computing ecosystems (e.g., Linux, macOS, and Windows), burdening users with outright failures, and developers with costly remediation efforts. In the race to proactively thwart bugs *before* they emerge post-deployment, developers are increasingly turning to *fuzzing*: an automated testing technique that scrutinizes software by generating massive amounts of randomly-mutated test cases.

Fuzzers are uniquely engineered to target specific software *interfaces*: the channels by which user input is passed into a program—and ultimately triggers its bugs. For example, popular fuzzer AFL [54] and its successor, AFL++ [10], both focus on *file*-based interfaces, mutating on-disk files and subsequently re-executing the target program on each to uncover its aberrant runtime behaviors. Others, such as libFuzzer [44] and Nyx [42], instead target *memory*-based interfaces, such as API functions that consume buffered data. As nearly all desktop software fuzzers are merely derivations of these few "mother" fuzzers [37]—commonly AFL++ and libFuzzer—they subsequently target the very same interfaces. Yet, **one crucial interface remains universally under-tested across today's ever-growing desktop software ecosystems**: the *Graphical* User Interface (GUI).

Given their prevalence among desktop software, GUIs unsurprisingly are responsible for many program bugs. Public issue trackers reveal numerous crashes stemming from unhandled GUI-induced edge cases, plaguing applications as simple as calculators [19] to those as complex as image editors [7], 3-D modeling tools [16], and web browsers [2]. Unfortunately, existing desktop-based fuzzers like AFL++ are isolated to *non*-GUI interfaces, with zero direct support for GUIs [10]. While mobile SDKs offer built-in GUI testing (e.g., Android's Monkey [24], iOS's XCMonkey [22]), the diverse landscape of GUI software in commodity *desktop* OSes—coupled with incompatible system-level event handling between these platforms—impedes direct porting of mobile GUI fuzzers to desktop ecosystems. Consequently, desktop GUI fuzzing currently remains limited to cost-prohibitive commercial offerings (e.g., Ranorex [18], Squish [17]) or one-off, target-specific fuzzers (e.g., GUIFuzz [9] for calc.exe), **leaving an untold number of GUI-induced bugs hidden among today's critical desktop GUI software ecosystems**.

To overcome these challenges and unleash large-scale GUI testing on desktop applications, this paper introduces GUIFuzz++: the first general-purpose grey-box fuzzer for desktop GUI software. Unlike typical file- or buffer-mutating fuzzers, GUIFuzz++ systematically explores GUIs by mutating its *interactions*—clicks, scrolls, key-presses, window navigation, and more—facilitating discovery of complex GUI-induced errors in diverse desktop applications. Central to GUIFuzz++ is a novel *Interaction Interpreter*: a middle-layer for translating fuzzers' randomly-generated test cases into distinct

GUI operations, enabling conventional file- or buffer-mutating fuzzers to be *repurposed* for desktop GUI fuzzing. We further bolster GUIFUZZ++ by harnessing widely-available *Software Accessibility Technologies* [13], significantly enhancing GUI fuzzing precision and bug-finding effectiveness.

We implement GUIFUZZ++ atop today's leading non-GUI grey-box fuzzer, AFL++ [10], and evaluate its efficacy across a diverse corpus of 12 real-world desktop applications on Linux spanning popular GUI development frameworks such as QT [6], GTK [12], and Xorg [15]. We empirically evaluate GUIFUZZ++'s contributions and enhancements through a series of ablation studies, showing how its combined components create an effective platform for discovering GUI bugs in desktop software. Notably, GUIFUZZ++ reveals **23 previously-unknown GUI-induced bugs** across 11 desktop applications, of which 14 are so far confirmed or fixed by their developers.

Through the following contributions, this paper introduces **the first general approach for uncovering GUI-induced bugs in today's vast ecosystems of desktop GUI software**:

- We examine the challenges of extending fuzzing to GUI-based applications on desktop platforms such as Linux, macOS, and Windows. We survey existing state-of-the-art fuzzing approaches, and weigh their shortcomings with respect to enabling systematic desktop app GUI fuzzing.
- We leverage our insights to design GUIFUZZ++: the first general-purpose grey-box fuzzer for desktop GUI software. We detail how GUIFUZZ++'s design facilitates practical and far-reaching desktop GUI fuzzing, maintaining high precision toward effective GUI bug discovery.
- We evaluate GUIFUZZ++'s capabilities through a series of ablation studies across 12 popular Linux GUI applications spanning various software domains. We show that GUIFUZZ++ enables effective GUI bug discovery, culminating in the identification of 23 new GUI-induced crashes, of which 14 are so far confirmed or fixed.
- We release GUIFUZZ++ in addition to all of our evaluation artifacts and benchmarks at the following URL: https://github.com/FuturesLab/GUIFuzzPlusPlus.

II. Background, Related Work, and Motivation

This section introduces the fundamental topics related to GUIFUZZ++: software GUIs, GUI-induced bugs, and the challenges of fuzzing desktop GUI software.

A. GUIs: Graphical User Interfaces

Graphical User Interfaces (GUIs) are among the most prevalent features in modern software, enabling complex applications to offer rich and intuitive user interactions: clicking, dragging-and-dropping, scrolling, menu navigation, and much more. Today's GUI development market, valued at $885 million, is projected to surpass $2 billion by 2031 [47]. To craft these interfaces, developers typically utilize dedicated GUI development frameworks. Common examples include cross-platform libraries like QT [6] and GTK [12], Android's Jetpack [26], and Apple's UIkit [23].

Program	Crash Type	Brief Description	Bug ID
Glaxnimate [3]	Abort	Text object properties	#408
KolourPaint [30]	Abort	Double undo in new window	#457915
LabPlot [31]	Abort	Fitting function data	#372834
LibreCAD [36]	Segfault	Right click with move/copy	#235
MATE-calc [38]	Segfault	"Not" on long hex value	#114
PlotJuggler [8]	Segfault	Apply filter on curve	#603
Umbrello [33]	Segfault	Cancelling seq diagram class	#443580

TABLE I: Examples of known GUI-triggered bugs in desktop software.

Most GUI development frameworks employ a similar multi-threaded architecture: a dedicated "main" thread updates the user interface and dispatches GUI-issued events to the application's back-end, while one or more "worker" threads process the application's back-end operations. This separation is key to ensuring that the interface remains responsive, even as complex operations are handled in the background.

However, this design also brings unique challenges: thread coordination can introduce subtle concurrency bugs [35], while complex GUI component lifecycles can trigger temporal memory errors—both often surfacing only under specific interaction sequences (Table I). Moreover, the inherent complexity of GUIs, ranging from diverse user interactions and nested sub-menus to transient pop-up screens and other application-specific bottlenecks, poses major challenges to proactive bug discovery. To address this, a substantial body of research has emerged targeting GUI-induced errors in *mobile* app ecosystems such as Android and iOS [24], [22]. Yet, while these approaches have achieved great success in uncovering GUI bugs within mobile apps, today's ever-growing *desktop* software ecosystems spanning Linux, macOS, and Windows remain completely overlooked, **with *no* comparable solutions for discovering their GUI-related software defects**.

B. Why Fuzzers Fail on Desktop GUI Software

Among today's most proven approaches for software bug discovery is *fuzzing*: an automated software testing technique that uncovers bugs by generating and mutating massive amounts of test cases. Despite the variety of fuzzing techniques available currently [10], [44], desktop applications—accounting for over 96% of workplace computing needs today [4]—lack any practical fuzzing solutions for uncovering GUI-induced bugs. In the following, we survey contemporary fuzzing solutions, assessing their key shortcomings with respect to supporting GUI fuzzing for desktop-based software ecosystems.

Desktop Application Fuzzers: Popular application fuzzers like AFL [54], AFL++ [10], and honggFuzz [50] all target *file* interfaces, mutating test cases as on-disk files that are subsequently each fed to the program under test. Others such as libFuzzer [44] and Nyx [42] instead target *memory*-based interfaces, mutating in-memory data that is ultimately read by API functions, respectively. Despite their proven success, these mainstream grey-box fuzzers—the foundational frameworks for most modern fuzzers [37]—lack *any* support

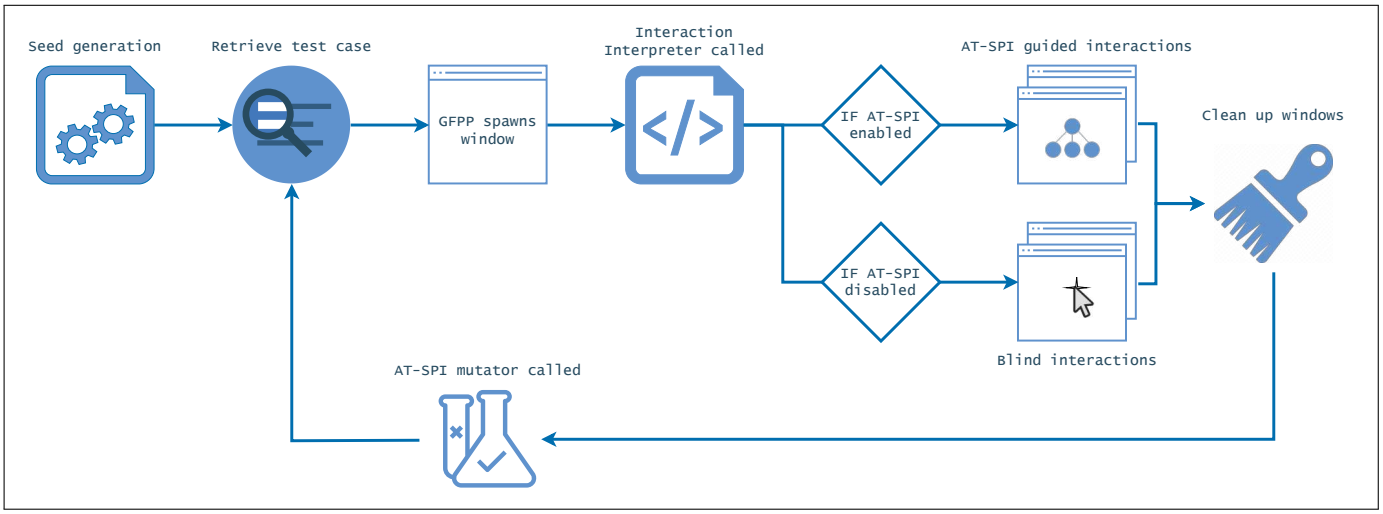

Fig. 1: High-level visualization of GUIFuzz++'s core GUI fuzzing workflow.

for GUI testing, instead concentrating on traditional file- and memory-based program interfaces. Though recent advancements attempt to bypass GUIs via automated program slicing (e.g., Winnie [28]), these methods merely redirect testing to typical file- or buffer-based interfaces, **leaving bugs caused by GUI *interactions* still undiscoverable**.

Desktop Environment Fuzzing: EnvFuzz [39], a recent grey-box fuzzing approach, instead mutates desktop applications' *environment*-level interfaces such as configuration files, fonts, themes, and sockets. While EnvFuzz has indeed been applied to desktop GUI applications (e.g., calculators [20]), it inherits conventional desktop fuzzing's [10] limitation of supporting only *data*-level interfaces, leaving it unable to explore GUI interactions whatsoever. This limitation is further reflected in **EnvFuzz's failure to uncover any genuine *GUI*-induced bugs** across its tested GUI applications [39].

Mobile App GUI Fuzzers: Although GUI testing continues to see adoption in mobile ecosystems, fundamental differences between underlying GUI frameworks, event-handling models, and application architectures leave mobile app GUI fuzzers unusable on desktop software. Mobile platforms typically offer well-defined UI lifecycles and standardized GUI development APIs, and consequently, mobile GUI fuzzers remain tightly coupled with platform-specific SDKs (e.g., Android's Monkey [24], iOS's XCMonkey [22]). In contrast, desktop environments are highly heterogeneous, with diverse GUI frameworks (e.g., Qt [6], GTK [12]) and interaction paradigms (e.g., first-party vs. third-party windows) that lack centralized control mechanisms. Moreover, desktop applications often rely on complex, multi-window workflows as well as non-touch-based inputs, further complicating automated testing. For these reasons, **mobile app fuzzers are neither currently used—nor practically adaptable—for testing *desktop*-based GUIs**.

Motivation: the need for a desktop-based GUI fuzzer. As today's fuzzers are universally unable to test desktop GUI software, we aim to bridge the gaps between widely-successful fuzzing platforms and desktop GUI targets. We envision a world where desktop application GUIs are fuzzable **just as any other software interface**, and thus design a fuzzer to meet these capabilities: **GUIFuzz++**.

III. GUIFuzz++: Challenges and Solutions

To bridge the long-standing gap between conventional fuzzing and GUI-based software, we introduce GUIFuzz++: the first system to extend general-purpose desktop fuzzing platforms to support today's diverse and complex desktop GUI application ecosystems. In the following, we outline the fundamental challenges that motivate GUIFuzz++'s core design (Figure 1), along with our corresponding solutions aimed at enabling effective and scalable GUI fuzzing.

A. Challenge 1: Making Desktop Fuzzers Interact with GUIs

Unlike prior GUI testing tools which are tied to specific targets [9] or platforms [24], GUIFuzz++ aims for *breadth* in supporting a wide range of desktop GUI software and ecosystems. To achieve this, we draw inspiration from mainstream desktop-based fuzzers like AFL++ [10] which, while incompatible with GUIs, are by far today's most far-reaching and ubiquitous desktop software fuzzing tools in practice [45]. Our goal, thus, is extending these general-purpose fuzzers to desktop GUIs—with the *least* modification necessary—via a novel mechanism for directly translating their random inputs into concrete GUI events: our GUI *Interaction Interpreter*.

Interpreting Fuzzer Inputs as GUI Events: Contemporary desktop fuzzers such as AFL++ [10] and libFuzzer [44] operate at the *byte* level, generating a continuously-growing corpus of random, string-based inputs. Using widely-available GUI automation APIs [49], we thus see an opportunity to reinterpret these inputs as sequences of GUI operations, analogous to

Op Structure	Description of GUI Interaction
`00 FF FF`	**Close** currently-active window, ignoring the last two operands.
`01 CC FF`	**Input** the key press corresponding to the extended `ASCII` encoding of primary operand `CC`, ignoring the second operand. ***Ex:*** `01 7F FF` → input extended `ASCII` key press "`DEL`".
`02 XX YY`	**Click** the location (`X%`, `Y%`) relative to the current window's dimensions, offset from its bottom-left coordinate (`0`, `0`). ***Ex:*** `02 A0 1B` → click relative position (`62.5%`, `10.5%`).
`03 XX YY`	**Drag** the cursor from its *current* position to the *new* position (`X%`, `Y%`) relative to the current window's dimensions, offset from its bottom-left coordinate (`0`, `0`). ***Ex:*** `03 00 80` → drag to relative position (`0%`, `50%`).
`NN AA BB`	**All higher opcodes** (i.e., `04`–`FF`): normalize the opcode via (`NN % 4`), reinterpreting the transformed opcode accordingly. ***Ex:*** `B2 2C 9F` → reinterpret as *click* operation `02 2C 9F`.

TABLE II: Overview of GUIFuzz++'s core GUI operation grammar.

the randomized "monkey"-style GUI interaction testing that has historically proven effective in mobile GUI fuzzing [24]. To enable this, we formalize fundamental GUI actions via a minimal grammar of *three-byte* instructions, shown in Table II.

Upon receiving an input from the fuzzer, GUIFuzz++ invokes its Interaction Interpreter—a component fully independent of the fuzzer—parsing the input bytes according to our GUI event grammar and dispatching corresponding actions. To keep our grammar compact, our instruction set (Table II) defines four core opcodes (i.e., `00`–`03`), with any higher opcodes normalized via modulo and mapped back into this defined range. For instance, a three-byte sequence beginning with opcode `06` is normalized to `02`, and thus interpreted as a *click* event. This simple normalization enables seamless integration with the random byte sequences produced by existing fuzzers, requiring no changes to their input generation logic.

Facilitating GUI Interaction: Central to our approach is leveraging existing GUI automation and introspection capabilities. Because many Table II instructions require window-relative positioning, GUIFuzz++ first queries the target application's window dimensions using native windowing APIs provided by the host OS (e.g., x11-utils [14] on Linux). It then executes the parsed GUI actions using cross-platform automation libraries (e.g., PyAutoGUI [49]), which expose generic primitives for mouse, keyboard, and window management. These readily available APIs allow GUIFuzz++ to issue platform-agnostic GUI interactions, including higher-level actions such as spawning or closing windows, without the need for any application-specific instrumentation.

Breadth of GUI Interactions: Because GUIFuzz++ aims to avoid the overhead of program-specific static analysis or tailoring, it operates using a minimal yet expressive set of core events: window-closing, key presses, clicks, and drags. These primitives form a functional superset capable of emulating a wide range of GUI *element-specific* interactions. For example, scrolling through a menu can be accomplished entirely via a drag operation, allowing GUIFuzz++ to explore substantial interface behavior using just these basic inputs. Moreover, our Interaction Interpreter is designed for easy extensibility with *new* opcodes—and, as we demonstrate in § III-C—it readily supports the integration of more targeted, element-specific mutators toward higher-precision GUI exploration.

Solution 1: GUIFuzz++ introduces a *GUI Interaction Interpreter* that enables conventional, GUI-agnostic desktop-based fuzzers like AFL++ to be fully repurposed for GUI fuzzing—without any need for costly reengineering.

B. Challenge 2: Handling Desktop-specific Window Obstacles

Unlike mobile platforms, desktop GUI fuzzing faces significantly more obstacles from the proliferation of unwanted *third-* and *first-party* windows, which disrupt fuzzing workflows and pollute the interaction space. On mobile OSes, apps are typically sandboxed with strict lifecycle control [1], where only one app is active in the foreground at a time, and popups or overlays are generally constrained by platform-level guidelines and permission models. In contrast, desktop environments allow *multiple* overlapping windows from *different* processes (e.g., update dialogs, crash reporters, or unrelated apps). Even within a single program, modal dialogs, system alerts, and nested windows may appear unpredictably. These extraneous windows can intercept input, obscure the target interface, or cause unintended side effects during fuzzing. In the following, we detail GUIFuzz++'s mechanisms for mitigating unwanted third- and first-party windows to ensure interactions remain focused on the *intended* application GUI.

Tackling Third-party Window Interference: To ensure that GUI fuzzing remains confined to the intended application window, GUIFuzz++ records the target process's PID, and initiates GUI interactions only when the currently displayed window *matches* that PID. Once a test case completes its sequence of interactions, GUIFuzz++ sends an interrupt signal (`SIGINT`) to terminate the window, returning control to the fuzzer to begin the next iteration. If the target or OS spawns unwanted third-party windows (e.g., an update dialog or the web browser), GUIFuzz++ collects their PIDs and similarly terminates each via `SIGINT`. Nearly all of GUIFuzz++'s window management logic resides within the GUI Interaction Interpreter, with only two additional lines of code added to AFL++ to capture the target application's PID.

Tackling First-party Window Interference: While third-party interference accounts for the majority of window-related disruptions, we also observe several cases where *first*-party windows—those spawned by the fuzzed application itself—can impede fuzzing. Unlike third-party windows, these originate from the target process, and thus cannot be filtered out using our PID-based third-party window filtering. The most common example involves file browser dialogs, which are often triggered by GUI actions such as clicking a SAVE or OPEN button. These dialogs pose a particularly dangerous risk: if the fuzzer inadvertently interacts with them, it may initiate unintended operations on the host file system.

To mitigate this, we extend our use of GUI introspection APIs (§ III-A) to heuristically detect and suppress such di-

alogs. Specifically, we scan for window *titles* containing common file-related keywords like LOAD, SAVE, and FILE, allowing us to identify and preemptively close or bypass most file-browser windows before they interfere with fuzzing execution. As with our third-party window filtering, this mechanism resides entirely within GUIFUZZ++'s Interaction Interpreter, requiring *no* additional customization to the fuzzer itself.

Solution 2: GUIFUZZ++ mitigates unwanted window interference with minimal changes to the underlying fuzzer, leveraging its core window introspection APIs to find and suppress signs of disruptive or extraneous GUI activity.

C. Challenge 3: Maintaining Precise Desktop GUI Interaction

A key challenge in GUI fuzzing is the disconnect between screen coordinates and interactive elements, making arbitrary interaction likely to *miss* actionable GUI components. While mobile platforms aid fuzzers [24], [22] with built-in GUI introspection APIs (e.g., Android's AccessibilityService [25]), desktop OSes *lack* such centralized mechanisms for introspecting GUI elements, leaving GUIFUZZ++'s interactions unlikely to drive meaningful fuzzing progress. To overcome this, we introduce a suite of enhancements leveraging emerging desktop-based *Software Accessibility Technologies* [13], providing GUIFUZZ++ with a powerful means of directly targeting GUI components toward more effective fuzzing.

Achieving GUI Introspection via AT-SPI: To meet the needs of assistive devices such as screen readers, magnifiers, and braille displays, recent years have seen the widespread adoption of the Assistive Technology Service Provider Interface (AT-SPI) [13]—the primary accessibility framework for Linux desktop environments, with emerging support extending to platforms like macOS. At a high level, AT-SPI exposes a hierarchical view of an application's GUI elements (e.g., buttons, menus, and text fields), enabling accessibility tools to support meaningful, non-visual navigation. With built-in support for popular GUI toolkits such as GTK and Qt, AT-SPI offers a robust foundation for external tools to inspect and reason about interface structure—making it a natural fit for introspection-driven enhancements within GUIFUZZ++.

Leveraging AT-SPI in GUI Fuzzing: To improve GUIFUZZ++'s GUI-interaction precision, we leverage AT-SPI's built-in recognition of standard GUI elements [13]. Accordingly, we extend our GUI Interaction Interpreter with 11 new operators targeting the fundamental classes of interactable GUI elements (Table III): toggleables, selections, movables, as well as general push buttons and user-controllable text fields. Continuing from Table II, we assign each of these 11 new instructions a unique opcode, with normalization similarly applied via modulo to bring any fuzzer-generated higher-opcode instructions (e.g., `17`) within GUIFUZZ++'s full expanded instruction opcode range (i.e., `00–14`).

Although GUIFUZZ++'s core clicking operation (Table II) relies on window-relative positioning to find *where* to click, AT-SPI exposes a deterministic tree of all currently-visible GUI elements, enabling *operand-guided* targeting for GUIFUZZ++'s enhanced interactions as well. Namely, each element type in Table III is accessed via a *click*, with the

Category	Op Structure	Element Type	Visual Example
General	`04 AA BB`	Pushable Button	Submit
	`05 AA BB`	Text Entry Field	Find ...
Toggleables	`06 AA BB`	Checkbox Button	Apply?
	`07 AA BB`	On/Off Button	On Off
Selections	`08 AA BB`	Radio Button	Use Option 1 / Use Option 2
	`09 AA BB`	Spinner Button	Width: 4px
	`10 AA BB`	Table Cell Button	+/- % ÷
	`11 AA BB`	Drop-down Item	Option 1 / Option 2 / Option 3
	`12 AA BB`	Combination Box	Search: "width" / Option 1: Height / Option 2: Width
Movable	`13 AA BB`	Scrollable Field	Field 2: value 2 / Field 3: value 3 / Field 4: value 4
	`14 AA BB`	Sliding Selection	0 1 2 3 4 5 6 7

TABLE III: GUIFUZZ++'s AT-SPI enhanced interactions. As in Table II, every operation is mapped to a *three*-byte structure: a single opcode, followed by two operands `AA` and `BB` that reference the GUI element's *position* within the type-specific element list that is exposed by AT-SPI and updated alongside the GUI. Similarly, higher opcodes are *normalized* to range `00–14` via `(NN % 15)`. For example, sequence `58 00 04` is normalized to `13 00 04`, and thus interpreted as an interaction on the *fourth*-indexed *scrollable field* element.

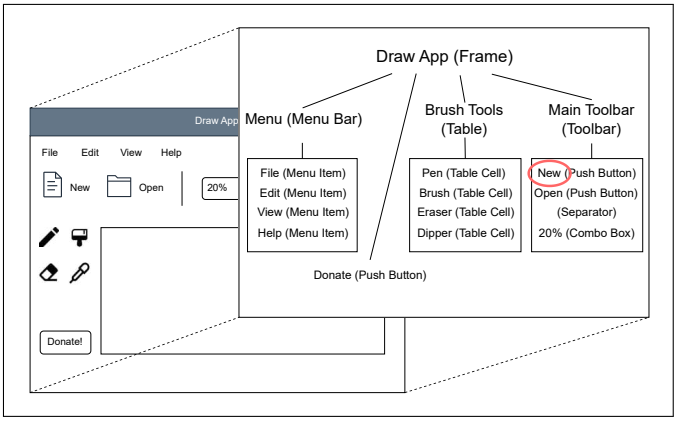

Fig. 2: Example visualization of an AT-SPI [13] dynamically-generated GUI element tree for a simple drawing app. In order to use the AT-SPI tree, GUIFUZZ++ flattens it into a list of each type of element (e.g., *push* button and *menu* item from Table III). Different operators select different types of elements, and the operands are subsequently used to index into the list.

instruction's final two operands used to index into a type-specific *list* of matching elements (Figure 2). GUIFuzz++ constructs these lists dynamically—re-polling AT-SPI's tree of GUI elements after each dispatched GUI interaction—and flattening all same-type nodes from the AT-SPI tree, enabling fast, deterministic element selection. If the resulting index exceeds the list's bounds, GUIFuzz++ wraps it via modulo to ensure a valid target. This design allows GUIFuzz++ to precisely fetch specific GUI elements even as the interface evolves at runtime, offering far more fruitful GUI fuzzing compared to blind, on-screen pixel clicking.

Solution 3: GUIFuzz++ overcomes GUI interaction imprecision through a suite of AT-SPI–assisted operations, enabling *targeted, element-aware* interactions that significantly improve the effectiveness of desktop GUI fuzzing.

IV. Implementation

We implement GUIFuzz++ atop state-of-the-art grey-box fuzzer AFL++ [10] v4.21c, enabling GUIFuzz++'s inheriting of AFL++'s rich ecosystem of fuzzing enhancements. Below, we detail the technical integration of GUIFuzz++'s core components within the AFL++ platform.

A. Fuzzing Process Execution

Like all AFL-based fuzzers, GUIFuzz++ resets the target process for new test cases via *forkserver*-based process cloning [53], enabling higher fuzzing throughput than slower from-scratch process creation [46]. Beyond executing the target, we configure AFL++ to additionally launch our GUI Interaction Interpreter (§ III-A), which we implement via Python's PyAutoGUI framework [49]. All other fuzzer execution steps—code coverage collection, crash recognition, and inter-process communication—are left as-is in AFL++'s core, underscoring GUIFuzz++'s lightweight design. In total, our changes to AFL++'s core span just **eight** lines of code.

Importantly, GUIFuzz++ supports any AFL-compatible bug oracle (e.g., flagging error-revealing process signals like `SIGFPE`) or sanitizer (e.g., AddressSanitizer [43]), following the same compile-time instrumentation and target preparation steps as conventional non-GUI fuzzing workflows [10], [44].

B. Test Case Mutation and Trimming

Since AFL++'s [10] in-house mutators modify test cases at bit- and byte-level, they will overwhelmingly *break* the three-byte structure of our GUI operations (§ III-A), leading to invalid interactions—and fruitless fuzzing. To address this, we implement a GUI-*aware* mutator, ensuring that mutations (e.g., modifications, insertions, and splices) occur strictly on *well-formed* GUI interactions. We further extend this to test case *trimming*, ensuring that incremental deletions similarly preserve GUI operation structures. As GUIFuzz++'s mutation and trimming are both implemented via AFL++'s Custom Mutator API [10], no changes are needed to AFL++ itself.

C. Supported Software and Desktop Platforms

While our current prototype of GUIFuzz++ targets GUI software in Linux environments, we anticipate its portability to other platforms supportive of AFL++ and PyAutoGUI [49] such as macOS, and extensible to AFL-like fuzzers on platforms *not* directly supported by AFL++, such as WinAFL [55] for Windows. We posit that the only *platform-specific* component of GUIFuzz++ is the retrieval of the active window's PID and dimensions. Fortunately, nearly all modern OSes expose APIs for this functionality via their respective windowing subsystems (e.g., x11-utils on Linux [14], PyObjC/Quartz on macOS [40], and PyWin32 on Windows [21]), enabling GUIFuzz++ to merely swap-out these components.

Although the accessibility framework driving GUIFuzz++'s higher-precision fuzzing (§ III-C), AT-SPI [13], sees best support for GTK- [12] and QT-based [6] GUIs, GUIFuzz++ remains fully functional without it. This allows grey-box GUI fuzzing to be deployed **across a wider range of targets—even in the absence of accessibility integration**.

V. Evaluation

Our evaluation of GUIFuzz++'s desktop GUI fuzzing capabilities is guided by the following fundamental questions:

Q1: How does GUIFuzz++'s grey-box fuzzing compare to traditional black-box GUI fuzzing?

Q2: To what extent does GUIFuzz++'s AT-SPI-enhanced interaction improve GUI fuzzing?

Q3: Is GUIFuzz++ effective at finding new GUI-induced bugs in desktop GUI software?

Benchmarks: Table IV shows our evaluation benchmarks. We evaluate GUIFuzz++ on 12 open-source Linux-based GUI programs spanning a variety of application domains. To assess GUIFuzz++'s support across today's diverse desktop GUI ecosystems, we include benchmarks spanning three distinct GUI development frameworks: Qt [6], GTK [12], as well as Xorg [15]. We compile all applications with AFL++'s built-in source-level compilers (e.g., `afl-clang-fast`).

Program	Description	Base GUI
Dia [27]	Graphic Design	GTK
Glaxnimate [3]	Animation	Qt
KCalc [29]	Calculator	Qt
KolourPaint [30]	Image Editor	Qt
LabPlot [31]	Data Plotting	Qt
LibreCAD [36]	3-D Modeling	Qt
MATE-calc [38]	Calculator	GTK
PlotJuggler [8]	Data Plotting	Qt
QCAD [41]	3-D Modeling	Qt
Skrooge [32]	Finance	Qt
Umbrello [33]	UML Editor	Qt
XCalc [52]	Calculator	Xorg

TABLE IV: Our desktop-based GUI fuzzing evaluation benchmarks.

Experiment Setup & Infrastructure: As there are **zero** fuzzers broadly supportive of desktop GUI software today, our evaluation seeks to understand the influence of GUIFuzz++'s

Program	C1: Grey-box w/ AT-SPI			C2: Grey-box w/o AT-SPI			C3: Black-box w/ AT-SPI			C4: Black-box w/o AT-SPI		
	Speed	CodeCov	Bugs	Speed	CodeCov	Bugs	Speed	CodeCov	Bugs	Speed	CodeCov	Bugs
Dia	2491.2	**12990.4**	0	1631.6	12074.8	**1**	**3154.0**	12101.4	0	3047.0	11364.8	0
Glaxnimate	2562.6	**34312.2**	**2**	2850.2	30537.2	0	6625.6	30890.0	0	**7374.0**	29405.3	0
KCalc	2546.8	**8013.8**	1	2914.5	7616.4	1	**7212.6**	6625.0	1	6364.6	5473.8	**2**
KolourPaint	1859.8	**5845.2**	**3**	2019.4	4880.0	1	5579.4	5615.2	2	**7595.8**	4988.8	1
LabPlot	2190.0	25170.6	**3**	2588.8	25114.0	2	4647.8	26402.6	2	**6037.0**	**27732.2**	1
LibreCAD	2001.6	**39613.6**	1	2453.3	38703.3	1	6760.2	39076.8	0	**8453.4**	36529.4	0
MATE-calc	1989.0	1196.6	0	2421.0	1349.0	0	5542.8	1353.4	0	**6490.0**	**1359.6**	**2**
PlotJuggler	2411.2	**17979.8**	1	1968.2	16342.2	1	5389.2	17083.6	1	**5697.6**	16201.0	1
QCAD	2647.0	61598.0	0	3020.2	61568.4	**1**	**7075.6**	**61734.2**	0	6928.2	61626.4	0
Skrooge	1754.8	33794.2	0	1575.0	**34052.6**	0	6040.4	33550.4	0	**7822.2**	32642.4	0
Umbrello	2279.0	**23431.0**	**5**	3243.4	17900.8	3	6854.0	20335.8	1	**8110.0**	16693.6	2
XCalc	6344.8	**441.8**	2	5560.0	441.4	2	**28078.8**	440.6	2	27887.2	439.8	2
GEOMEAN:	2431.4	**12079.5**	**1.9**	2536.9	11409.2	1.3	6572	11635.3	1.4	**7555.6**	10961	1.5

TABLE V: Per-configuration mean fuzzing speed (i.e., test case throughput), mean target code coverage (i.e., control-flow edges), and total manually-deduplicated bugs per benchmark. **Bolded** values indicate the best-performing configuration with respect to each evaluated metric per benchmark.

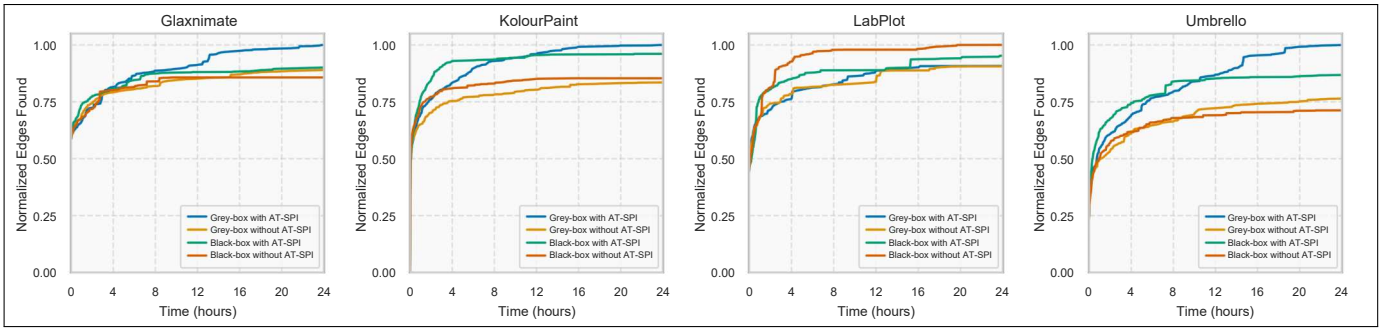

Fig. 3: Five-trial mean coverage over time for GUIFuzz++'s four configurations. We normalize coverage based on the top-ranked competitor per benchmark.

key constituent parts. Accordingly, we set up GUIFuzz++ in four fundamental configurations: **(C1)** grey-box mode with AT-SPI, **(C2)** grey-box mode without AT-SPI, **(C3)** black-box mode with AT-SPI, and **(C4)** black-box mode without AT-SPI. We seed all fuzzing campaigns with a single sequence of 33 randomly-generated GUI interactions. Following the fuzzing evaluation standard established by Klees et al. [34], we fuzz each benchmark for five 24-hour trials per configuration. We deploy all fuzzing experiments within 20 KVM-based Ubuntu virtual machines, run atop a 24-core Ubuntu 22.04 workstation with 64G RAM and an Intel i9-12900K CPU.

Post-processing Results: We evaluate all GUIFuzz++ configurations on the following metrics: **speed** via test case throughput, **depth** via code coverage, and true GUI-induced **bugs** uncovered. We measure code coverage as control-flow *edges* via AFL++'s built-in `afl-showmap` tool [10], and further plot mean normalized coverage in Figure 3. For bugs, we perform manual analysis to deduplicate fuzzer-found crashes into their per-configuration unique bugs. We calculate and report all means as geometric means.

A. Q1: Grey-box vs. Black-box Desktop GUI Fuzzing

As GUI fuzzing has historically relied heavily on black-box testing [24]—eschewing any target-level feedback about test case significance (e.g., code coverage)—we evaluate GUIFuzz++'s performance in both traditional black-box and more modern grey-box (i.e., coverage-guided [10]) fuzzing modes. Since GUIFuzz++'s underlying fuzzer, AFL++, is inherently *not* a black-box fuzzer, we modify it to emulate black-box behavior by continuing to save all coverage-increasing test cases on disk, but refraining from actually *using* that coverage information during the fuzzing process. This strategy enables us to accurately measure code coverage even in the absence of coverage-guided exploration. Beyond code coverage, we also compare both modes' execution speeds (measured by test-case throughput) and their total fuzzer-reported unique crashes. Table V summarizes these results side-by-side per benchmark.

Outcomes: As shown in Table V, GUIFuzz++'s black-box configurations (C3 and C4) consistently achieve the highest test case throughput across all benchmarks, with grey-box configurations (C1 and C2) never surpassing them in speed. However, this performance advantage does not translate to effectiveness: black-box fuzzing yields lower code coverage—outperformed in 9 of 12 benchmarks—and fewer bugs, with grey-box mode discovering more issues in 7 of 12 applications.

Interestingly, we observe that the few cases where black-box mode excels in bug discovery involve simpler applications with more constrained interaction spaces: namely, the MATE-calc [38] and KCalc [29] calculators. These results suggest that black-box fuzzing's speed is best suited for lightweight GUIs, whereas grey-box fuzzing is better-equipped to navigate complex, bug-prone paths in larger applications. Overall, by supporting both modes, **GUIFuzz++ enables flexible,**

target-tailored fuzzing strategies that adapt to the structure and complexity of today's diverse desktop GUI applications.

> **Q1:** GUIFuzz++ adapts to target GUI complexity, facilitating black-box speed for simple apps, and grey-box precision for deeper bug discovery in more complex ones.

B. Q2: Impact of GUIFuzz++*'s Enhanced GUI Interaction*

To determine whether GUIFuzz++'s AT-SPI-enhanced GUI interaction (§ III-C) improves desktop GUI fuzzing effectiveness, we further evaluate GUIFuzz++ both with *and* without AT-SPI enabled. As in § V-A, the test case throughput, code coverage, and fuzzer-reported unique crashes per configuration–benchmark pairing are shown in Table V.

Outcomes: As Table V shows, GUIFuzz++'s AT-SPI-assisted configurations (C1 and C3) face significant runtime overhead—being outperformed in throughput by their AT-SPI-agnostic counterparts (C2 and C4) on 8 of 12 benchmarks. We posit this is expected, as AT-SPI integration incurs additional costs from both rendering the accessibility tree as well as invoking GUIFuzz++'s targeted GUI-aware mutators.

Despite this cost, AT-SPI-assisted fuzzing proves valuable: it achieves higher code coverage in 9 of 12 applications—and in many cases, seeing consistently-higher coverage throughout fuzzing (Figure 3)—indicating that richer GUI introspection enables exploration of deeper, more complex interaction paths that are otherwise missed by GUI-agnostic modes. This benefit extends to bug discovery as well: when paired with grey-box guidance, AT-SPI-enhanced fuzzing finds the most bugs on four benchmarks, surpassing the next-best configuration, grey-box without AT-SPI (C2), which leads on only two. Altogether, these results demonstrate that while AT-SPI slows execution, **it greatly boosts both code coverage and bug-finding in interface-rich applications**—solidifying GUIFuzz++'s effectiveness in GUI-focused fuzzing campaigns.

> **Q2:** GUIFuzz++'s AT-SPI integration trades raw speed for deeper exploration—unearthing more bugs and behaviors invisible to GUI-agnostic desktop GUI fuzzing.

C. Q3: Discovery of GUI-induced Bugs in Desktop Apps

Lastly, we manually deduplicate all fuzzer-reported unique crashes per each of GUIFuzz++'s evaluated configurations, obtaining the final set of real-world bugs uncovered by GUIFuzz++. We follow standard practice in fuzzing literature for crash deduplication [34], employing AddressSanitizer-based [43] stack-trace bucketing. Table VI lists all GUI-induced bugs found in our evaluation, alongside their revealing GUIFuzz++ configuration and current reporting status.

Outcomes: As shown in Table VI, GUIFuzz++ uncovers a total of 25 GUI-induced bugs across 11 real-world desktop GUI applications, with **23 being previously-unreported** GUI-induced errors. Figure 4 breaks down the distribution of unique bugs across different fuzzing configurations, highlighting that each configuration surfaces distinct issues—and underscoring the importance of GUIFuzz++'s support for tailoring fuzzing strategies to the characteristics of the target application.

ID	Program	Bug Type	Brief Desc.	New	Status
01	Dia	Bad Free	Color area (transient)	✔	🔒
02	Glaxnimate	Segfault	Improper closing	✔	⌛
03	Glaxnimate	Segfault	Invalid cut/pastes	✔	⌛
04	KCalc	Invalid Ptr	Inserting open parent	✔	👍
05	KCalc	Segfault	Left bit shift overflow		
06	KolourPaint	Heap UAF	Specific tools with undo	✔	👍
07	KolourPaint	Segfault	Buggy bug report menu	✔	⌛
08	KolourPaint	Segfault	Shortcut settings dropdowns	✔	⌛
09	KolourPaint	Segfault	Print preview zooming	✔	⌛
10	LabPlot	Invalid Ptr	Invalid column insert	✔	🔒
11	LabPlot	Heap UAF	Pinning spreadsheets	✔	🔒
12	LabPlot	Heap UAF	Pinning matrices	✔	🔒
13	LibreCAD	Heap UAF	Invalid plugin usage	✔	🔒
14	LibreCAD	Heap UAF	Consecutive points	✔	🔒
15	MATE-calc	Bad Free	Invalid square roots		
16	MATE-calc	Bad Free	Empty inverse trig functions	✔	🔒
17	PlotJuggler	Segfault	Quickly close button docker	✔	⌛
18	QCAD	Segfault	Tool use in multiple sheets	✔	⌛
19	Umbrello	Segfault	Birds eye after discard	✔	🔒
20	Umbrello	Heap UAF	Multiple sequence diagrams	✔	🔒
21	Umbrello	Heap UAF	Undo after discard	✔	🔒
22	Umbrello	Segfault	Print Preview after discard	✔	🔒
23	Umbrello	Segfault	Cut on empty diagram	✔	🔒
24	XCalc	FPE	Invalid modulus	✔	⌛
25	XCalc	FPE	Invalid modulus	✔	⌛

TABLE VI: All GUI-induced bugs uncovered by GUIFuzz++, alongside a brief description of their bug-triggering semantics and reporting statuses (🔒 = fixed by developers, 👍 = confirmed and waiting fixing, ⌛ = pending). For brevity, we classify bugs #04 and #10 as "Invalid Pointer" errors, and follow AddressSanitizer's [43] crash taxonomy for all remaining bugs.

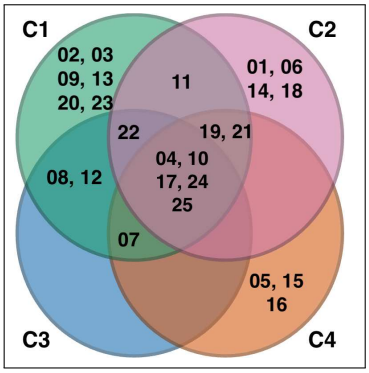

Fig. 4: Per-configuration GUI bugs found by GUIFuzz++. Bug IDs correspond to those listed in Table VI (**C1** = Grey-box w/ AT-SPI, **C2** = Grey-box w/o AT-SPI, **C3** = Black-box w/ AT-SPI, **C4** = Black-box w/o AT-SPI).

In the remainder of this section, we present several representative case studies that showcase the depth and diversity of bugs exposed by GUIFuzz++.

Case Study 1: LibreCAD (bug #13). GUIFuzz++ uncovered a heap user-after-free in LibreCAD [36] triggered by the following high-level operations: selecting the "Same Properties" plugin from the `Plugins` menu, placing an angle on the canvas using the `Angle` tool, and then beginning a two-point line with the `Two Points` tool, spanning a total of eight distinct GUI interactions (Figure 5).

This bug was confirmed and patched by developers on

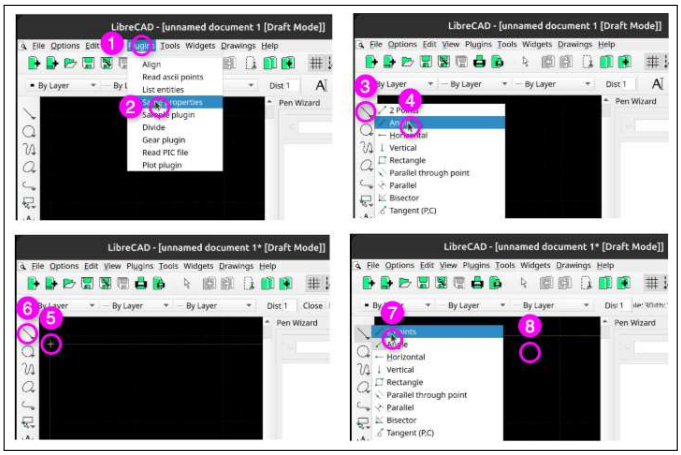

Fig. 5: LibreCAD crash from an *eight*-sequence interaction: (1) `Plugins` (click), (2) `Same Properties` (click), (3) `Lines Menu` (click), (4) `Select Angle` (click), (5) `Place Angle` (drag), (6) `Lines Menu` (click), (7) `Select Two Points` (click), and (8) `Place Two Points` (drag).

the same day it was reported. The root cause lies in the "Same Properties" plugin, which attempts to copy attributes from one diagram entity to another. A call to the plugin's `finish()` method triggers a segfault, as it prematurely deallocates objects still in use by the partially drawn two-point line. Because this line is incomplete, the plugin encounters invalid state during cleanup. Notably, this bug was only exposed under GUIFuzz++'s AT-SPI-enhanced configuration. This is likely due to the need for precise sequencing of submenu interactions—such as selecting transient plugin entries—which are rarely triggered through random UI exploration alone.

Case Study 2: LabPlot (bug #11). GUIFuzz++ finds another use-after-free in LabPlot [31] triggered by the following four-sequence interaction (Figure 6): clicking `Pin Active Tab`, opening the `Spreadsheet` menu, and selecting column #2 in the data browser. This bug, confirmed by the LabPlot developers one week after reporting, stems from incorrect usage of the Qt [6] docking API. When a dock widget is pinned, it is removed from its parent `QDockArea`. If this removal leaves the dock area empty, the area itself is destroyed. Later, when interacting with the spreadsheet column, LabPlot attempts to access this now-destroyed dock area, resulting in a segmentation fault. This issue was uncovered only in GUIFuzz++'s grey-box configuration, suggesting that coverage-guided exploration was necessary to navigate the specific conditions leading to the crash.

Case Study 3: Umbrello (bugs #19, #21, #22). GUIFuzz++ revealed a series of Umbrello [33] crashes during fuzzing, triggered by different UI actions following diagram deletion. These crashes stem from a missing null check: when a user discards changes, Umbrello deletes the active diagram but fails to update or reset internal state, leaving no valid diagram selected. Triggering sequences include when the user opens a new diagram and makes a change (e.g., adds a line or class), then clicks `New` and chooses to discard the current changes, and finally attempts to invoke actions such as `Print Preview` (e.g., Figure 7), `Bird's Eye View`, `Undo`, or interacts with the `Command History`.

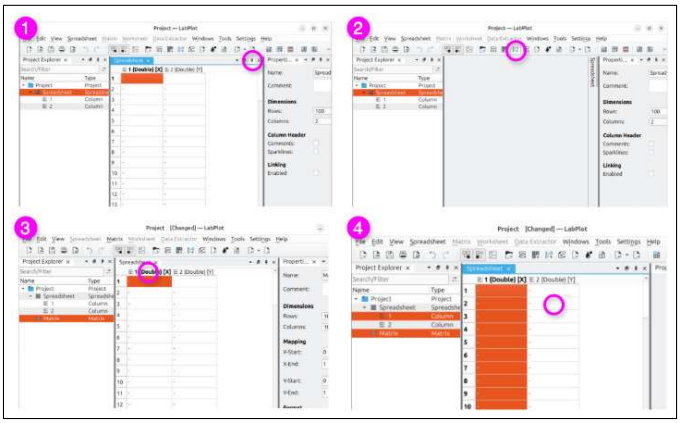

Fig. 6: LabPlot crash stemming from a *four*-sequence interaction: (1) `Pin` (click), (2) `Matrix` (click), (3) `Column 1` (click), then (4) `Cell 2,2` (click).

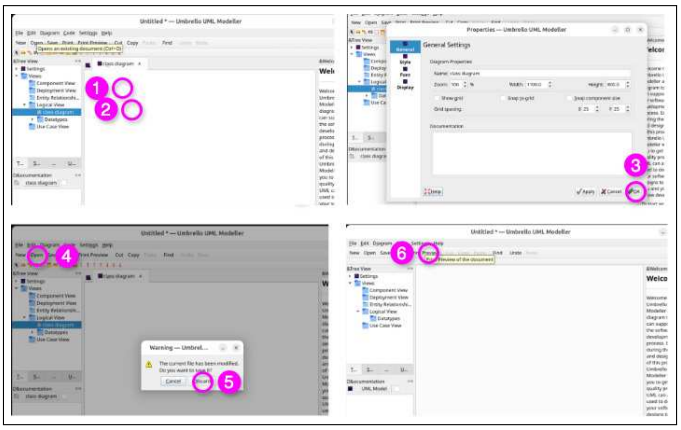

Fig. 7: Umbrello crash with root cause stemming from a *six*-sequence GUI interaction: (1) `Canvas` (click), (2) `Canvas` (click), (3) `Ok` (click), (4) `Open` (click), (5) `Discard` (click), (6) `Print Preview` (click).

Each of these operations assumes a valid diagram is present, triggering distinct `SIGSEGV` faults when dereferencing the null pointer. The exact crash varies by action, revealing a broader flaw in Umbrello's state management following diagram disposal. While these bugs were discovered across all configurations, a greater number surfaced during grey-box runs (Table V), suggesting that coverage-guided exploration is more effective at uncovering deeper, state-dependent bugs.

Developer Responses: While several bugs remain under review at the time of writing (Table VI, many were acknowledged and confirmed by developers within days to a week of reporting—underscoring their reproducibility and practical significance. Collectively, these case studies showcase GUIFuzz++'s effectiveness at **surfacing real, high-impact GUI bugs in desktop applications**, many of which stem from complex GUI interaction sequences that GUIFuzz++'s supported fuzzing modes are uniquely equipped to explore.

Q3: GUIFuzz++'s structured GUI interactions and application introspection enables its discovery of non-trivial GUI bugs—broadening the reach of fuzzing into the rich, event-driven behaviors of modern desktop applications.

VI. Discussion & Threats to Validity

In the following, we weigh several limitations of our prototype implementation of GUIFuzz++.

Improving Runtime Throughput: While conventional grey-box application fuzzing has seen numerous advancements for increasing speed toward accelerating bug discovery [46], [56], GUI fuzzing is inescapably much slower due to GUI-specific bottlenecks—namely, the high resource cost from *rendering* the GUI itself. Though some application-specific runtime bottlenecks seem mitigable (e.g., on-launch splash screens), we posit there is non-trivial engineering needed to build a generalizable way of (1) pinpointing and (2) successfully cutting-out these components without breaking the overall application. As such, we leave exploring potential optimizations for desktop GUI fuzzing to future work.

Supporting Other Interactions: Our current prototype's click-oriented GUI interactions (e.g., Table III) currently focus on left-clicks, though extending GUIFuzz++ to right-clicks is straightforward. However in practice, we observe right-click menus largely expose redundant functionality (e.g., `Cut`, `Copy`, `Paste`) that is already accessible through left-click–navigable menus (e.g., `Edit` → `Cut`), so omitting right-clicking does not meaningfully impact most GUI fuzzing. Likewise, while GUIFuzz++'s key-pressing interactions currently issue just single-key presses, extending support to multi-key combinations requires only minimal modification. We plan to explore these in future extensions to GUIFuzz++.

Generality to Other Platforms: Although GUIFuzz++'s current prototype remains Linux-specific, many of its components already are cross-platform—or are easily swapped-out with their platform-specific counterparts—as we discuss in § IV-C. Importantly, GUIFuzz++ specifically targets desktop-based OSes, and intentionally avoids mobile OSes (e.g., Android, iOS) or tablet ones (e.g., iPadOS), as we expect that these platforms' already-mature GUI fuzzing tools [22], [24] remain a better fit for their respective GUI app ecosystems. While we do plan to explore GUIFuzz++'s porting to other desktop platforms like macOS and Windows, we leave this engineering and requisite re-evaluation to future work.

Generality to Other Applications: To mitigate risk of bias and overfitting, we evaluate GUIFuzz++ across a wide range of targets (Table IV) spanning diverse types of GUI applications: calculators, multimedia, financial, data analysis, and computer modeling. Additionally, these applications are built atop three of today's most popular GUI development frameworks—Qt [6], GTK [12], and Xorg [15]—further underscoring the generality of GUIFuzz++'s fundamental approach across diverse ecosystems of desktop GUI software.

Since GUIFuzz++ builds on existing fuzzing frameworks [10], it can be applied to closed-source GUI binaries as well (e.g., Tesla's Qt-based infotainment software [51]), provided that the base fuzzer and underlying program instrumentation supports them. However, we expect certain GUIFuzz++ modes (e.g., grey-box mode, AT-SPI [13]) likely require additional adaptation to work on closed-source targets.

Further, while we foresee many opportunities in extending GUIFuzz++ to other GUI-based application domains beyond our present evaluation set—such as video games, web browsers, or an aircraft avionics software—many domain-specific orthogonal challenges need solving first. For example, web browsers are large, multi-threaded codebases, yet multi-threading often deteriorates the stability of general grey-box fuzzing [5]. Similarly, video games likely require many new game-tailored bug oracles for non-crashing bugs (e.g., glitching-into Super Mario's "Minus World" [11], [48]). Others, such as avionics software, call for porting from their niche platforms (e.g., PowerPC) to more tool-friendly ones like Linux. Nonetheless, with solutions to these obstacles, we anticipate that GUIFuzz++ and future follow-on solutions will be ready to fuzz these and other important GUI domains.

VII. Conclusion

In this work, we introduce GUIFuzz++, a general-purpose framework that bridges the longstanding disconnect between modern grey-box fuzzing and desktop GUI software. While traditional fuzzers operate primarily over byte-level or spatial input domains, GUIFuzz++ formalizes GUI interactions as operand-driven instructions, systematically translating random bytes into actionable, logic-exercising GUI events. Through a minimal yet expressive core instruction set and integration with accessibility frameworks such as AT-SPI, GUIFuzz++ supports both pixel-based and element-aware interaction modes, along with built-in mechanisms to isolate fuzzing to the intended application context. Altogether, GUIFuzz++ enables mainstream fuzzers like AFL++ to be seamlessly extended to GUI fuzzing—with negligible intervention—providing **the first scalable platform for effectively uncovering GUI-induced defects in today's rapidly-evolving desktop software ecosystems**, as demonstrated by the **23 previously-unknown bugs** it has discovered thus far.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under Award No. 2419798, and by the Defense Advanced Research Projects Agency (DARPA) under Award No. FA8750-24-2-0002, Subaward No. GR105409-SUB00001384. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or DARPA.

References

[1] Android Developers. The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle, 2025.

[2] Audacity Community. [Bug] Audacity freezes when opening large files. https://github.com/audacity/audacity/issues/6678, 2023.

[3] Mattia Basaglia. Glaxnimate, 2025. URL: https://glaxnimate.mattbas.org/.

[4] Beyond Identity. BYOD: Exploring the Evolution of Work Device Practices in a New Remote-Forward Era, 2021. URL: https://www.beyondidentity.com/resource/byod-exploring-the-evolution-of-work-device-practices-in-a-new-remote-forward-era-survey.

[5] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *USENIX Security Symposium*, USENIX, 2020.

[6] The Qt Company. Qt Framework. https://www.qt.io/, 2024.

[7] Darktable Community. [Bug] Darktable crashes when importing files. https://github.com/darktable-org/darktable/issues/6150, 2023.

[8] Davide Faconti. PlotJuggler, 2025. URL: https://plotjuggler.io/.

[9] Brandon Falk. GUIFuzz: A GUI fuzzing application set up to fuzz calc.exe right now. https://github.com/gamozolabs/guifuzz, 2024.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[11] Llewellyn Forward, Io Limmer, Joseph Hallett, and Dan Page. Super Mario in the Pernicious Kingdoms: Classifying Glitches in Old Games. In *ACM/IEEE International Workshop on Games and Software Engineering*, GAS, 2024.

[12] GNOME Foundation. GTK: The GTK Project. https://www.gtk.org/, 2024.

[13] GNOME Foundation. AT-SPI2 Reference Manual. https://docs.gtk.org/atspi2/, 2025.

[14] X.Org Foundation. x11-utils: X11 X Window System utility programs. https://www.x.org/releases/current/doc/man/man1/x11-utils.1.xhtml, 2024.

[15] X.Org Foundation. X.Org Server. https://www.x.org/, 2024.

[16] FreeCAD Community. TechDraw dimension extension lines - extension bug. https://forum.freecad.org/viewtopic.php?style=4&t=66713, 2021.

[17] Froglogic GmbH. Squish Automated GUI Testing. https://www.froglogic.com/squish/, 2024.

[18] Ranorex GmbH. Ranorex Studio: GUI Test Automation. https://www.ranorex.com/, 2024.

[19] GNOME Community. [Bug] GNOME Calculator crashes on launch. https://gitlab.gnome.org/GNOME/gnome-calculator/-/issues/282, 2023.

[20] GNOME Foundation. GNOME Calculator. https://gitlab.gnome.org/GNOME/gnome-calculator, 2025.

[21] Mark Hammond. PyWin32: Python extensions for Windows. https://github.com/mhammond/pywin32, 2024.

[22] Apple Inc. XCMonkey: iOS Fuzzing Tool. https://developer.apple.com/documentation/xctest, 2024.

[23] Apple Inc. UIKit Framework. https://developer.apple.com/documentation/uikit, 2025. Accessed: 2025-04-20.

[24] Google Inc. Monkey Testing Tool for Android. https://developer.android.com/studio/test/monkey, 2024.

[25] Google Inc. AccessibilityService — Android Developers. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService, 2025.

[26] Google Inc. Android Jetpack. https://developer.android.com/jetpack, 2025.

[27] Inkscape Project. Inkscape, 2025. URL: https://inkscape.org/.

[28] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Network and Distributed System Security Symposium*, NDSS, 2021.

[29] KDE Community. KCalc, 2025. URL: https://apps.kde.org/kcalc/.

[30] KDE Community. KolourPaint, 2025. URL: https://apps.kde.org/kolourpaint/.

[31] KDE Community. LabPlot, 2025. URL: https://labplot.org/.

[32] KDE Community. Skrooge, 2025. URL: https://skrooge.org/.

[33] KDE Community. Umbrello UML Modeller, 2025. URL: https://uml.sourceforge.io/.

[34] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[35] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7), 1993.

[36] LibreCAD Community. LibreCAD, 2025. URL: https://librecad.org/.

[37] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11), 2021.

[38] MATE Desktop Environment. MATE Calculator, 2025. URL: https://wiki.mate-desktop.org/mate-desktop/applications/mate-calc/.

[39] Ruijie Meng, Gregory J Duck, and Abhik Roychoudhury. Program Environment Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2024.

[40] Ronald Oussoren. PyObjC: Python bindings for macOS Cocoa and Quartz APIs. https://pyobjc.readthedocs.io/en/latest/, 2024.

[41] RibbonSoft GmbH. QCAD, 2025. URL: https://www.qcad.org/en/.

[42] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems*, EuroSys, 2022.

[43] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, ATC, 2012.

[44] Kosta Serebryany. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *IEEE Cybersecurity Development Conference*, SecDev, 2016.

[45] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, USENIX, 2017.

[46] Leo Stone, Rishi Ranjan, Stefan Nagy, and Matthew Hicks. No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting. In *USENIX Security Symposium*, USENIX, 2023.

[47] Straits Research. Graphical User Interface (GUI) Design Software Market Size, Share and Forecast to 2031, 2023. URL: https://straitsresearch.com/report/graphical-user-interface-design-software-market.

[48] Super Mario Wiki Contributors. Minus World, 2025. URL: https://www.mariowiki.com/Minus_World.

[49] Al Sweigart. PyAutoGUI Documentation. https://pyautogui.readthedocs.org, 2025.

[50] Robert Swiecki. Honggfuzz, 2018. URL: http://honggfuzz.com.

[51] Haohuang Wen and Zhiqiang Lin. Egg hunt in Tesla infotainment: A first look at reverse engineering of Qt binaries. In *USENIX Security Symposium*, USENIX, 2023.

[52] X.Org Foundation. XCalc, 2025. URL: https://www.x.org/archive/X11R7.6/doc/man/man1/xcalc.1.xhtml.

[53] Michal Zalewski. Fuzzing random programs without execve(). http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014.

[54] Michal Zalewski. American fuzzy lop, 2017. URL: http://lcamtuf.coredump.cx/afl/.

[55] Google Project Zero and Ivan Fratric. WinAFL: A fork of AFL for Windows. https://github.com/googleprojectzero/winafl, 2024.

[56] Yunhang Zhang, Chengbin Pang, Stefan Nagy, Xun Chen, and Jun Xu. Profile-guided System Optimizations for Accelerated Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2023.