WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning

Jinho Jung, Stephen Tong, Hong Hu†, Jungwon Lim, Yonghwi Jin, Taesoo Kim
Georgia Institute of Technology †Pennsylvania State University

***Abstract*—Fuzzing is an emerging technique to automatically validate programs and uncover bugs. It has been widely used to test many programs and has found thousands of security vulnerabilities. However, existing fuzzing efforts are mainly centered around Unix-like systems, as Windows imposes unique challenges for fuzzing: a closed-source ecosystem, the heavy use of graphical interfaces and the lack of fast process cloning machinery.**

In this paper, we propose two solutions to address the challenges Windows fuzzing faces. Our system, WINNIE, first tries to synthesize a harness for the application, a simple program that directly invokes target functions, based on sample executions. It then tests the harness, instead of the original complicated program, using an efficient implementation of `fork` on Windows. Using these techniques, WINNIE can bypass irrelevant GUI code to test logic deep within the application. We used WINNIE to fuzz 59 closed-source Windows binaries, and it successfully generated valid fuzzing harnesses for all of them. In our evaluation, WINNIE can support 2.2× more programs than existing Windows fuzzers could, and identified 3.9× more program states and achieved 26.6× faster execution. In total, WINNIE found 61 unique bugs in 32 Windows binaries.

I. INTRODUCTION

Fuzzing is an emerging software-testing technique for automatically validating program functionalities and uncovering security vulnerabilities [42]. It randomly mutates program inputs to generate a large corpus and feeds each input to the program. It monitors the execution for abnormal behaviors, like crashing, hanging, or failing security checks [56]. Recent fuzzing efforts have found thousands of vulnerabilities in open-source projects [12, 28, 52, 62]. There are continuous efforts to make fuzzing faster [4, 9, 53] and smarter [60, 65, 67].

However, existing fuzzing techniques are mainly applied to Unix-like OSes, and few of them work as well on Windows platforms. Unfortunately, Windows applications are not free from bugs. Recent report shows that in the past 12 years, 70% of all security vulnerabilities on Windows systems are memory safety issues [43]. In fact, due to the dominance of Windows operating system, its applications remain the most lucrative targets for malicious attackers [10, 17, 18, 48]. To bring popular fuzzing techniques to the Windows platform, we investigate common applications and state-of-the-art fuzzers, and identify three challenges of fuzzing applications on Windows: a predominance of graphical applications, a closed-source ecosystem (e.g., third-party or legacy libraries), and the lack of fast cloning machinery like `fork` on Unix-like OSes.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24334
www.ndss-symposium.org

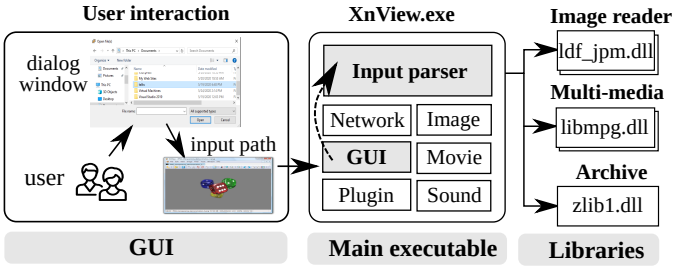

Fig. 1: Architecture of XnView on Windows. The program accepts the user input via the GUI. The main executable parses the received path and dynamically loads the library to process the input. A fuzzing harness bypasses the GUI to reach the functionality we wish to test.

Windows applications heavily rely on GUIs (graphical user interfaces) to interact with end-users, which poses a major obstacle to fuzzing. As shown in Figure 1, XnView [64] requires the user to provide a file through the graphical dialog window. When the user specifies the file path, the main executable parses the file, determines which library to delegate to, and dynamically loads the necessary library to handle the input. Although some efforts try to automate the user interaction [7], the execution speed is much slower than ordinary fuzzing. For example, fuzzing GUI applications with AutoIt yields only around three executions per second [20], whereas Linux fuzzing often achieves speeds of more than 1,000 executions per second. Speed is crucial for effective fuzzing, and this slow-down renders fuzzing GUI application impractical.

The general way to overcome the troublesome GUI is to write a simple program, called a *harness*, to assist with fuzzing. A harness replaces the GUI with a CLI (command-line interface), prepares the execution context such as arguments and memory, and invokes the target functions directly. In this way, we can test the target program without any user interaction. For example, with a harness that receives the input path from the command line and loads the `decoder` library, we can test XnView without worrying about the dialog window. Recent work has even explored generating harnesses automatically for open-source programs [8, 35].

Nevertheless, Windows fuzzing still relies largely on human effort to create effective harnesses because most Windows programs are closed-source, commercial-off-the-shelf (COTS) software [3, 21, 44, 55, 61]. Existing automatic harness synthesis methods require to access the source code, and thus cannot handle closed-source programs easily [8, 35]. Without the source code, we have little knowledge of the program's internals, like the locations of interesting functions and their prototypes. Since manual analysis is error-prone and unscalable to a large number of programs, we need a new method to generate fuzzing harnesses directly from the *binary*.

Finally, Windows lacks the fast cloning machinery (e.g., `fork` syscall) that greatly aids fuzzing on Unix-like OSes. Linux fuzzers like AFL place a fork-server before the target function, and subsequent executions reuse the pre-initialized state by forking. The fork-server makes AFL run 1.5×–2× faster on Linux [70]. `fork` also improves the stability of testing as each child process runs in its own address space, containing any side-effects, like crashes or hangs. However, the Windows kernel does not expose a clear counterpart for `fork`, nor any suitable alternatives. As a result, fuzzers have to re-execute the program from the beginning for each new input, leading to a low execution speed. Although we can write a harness to test the program in a big loop (aka., *persistent mode* [68]), testing many inputs in one process harms stability. For example, each execution may gradually pollute the global program state, eventually leading to divergence and incorrect behavior.

We propose an end-to-end system, WINNIE, to address the aforementioned challenges and make fuzzing Windows programs more practical. WINNIE contains two components: a harness generator that automatically synthesizes harnesses from the program binary alone, and an efficient Windows fork-server. To construct plausible harnesses, our harness generator combines both dynamic and static analysis. We run the target program against several inputs, collect execution traces, and identify interesting functions and libraries that are suitable for fuzzing. Then, our generator searches the execution traces to collect all function calls to candidate libraries, and extracts them to form a harness skeleton. Finally, we try to identify the relationships between different function calls and arguments to build a full harness. Meanwhile, to implement an efficient fork-server for Windows systems, we identified and analyzed undocumented Windows APIs that effectively support a Copy-on-Write `fork` operation similar to the corresponding system call on Unix systems. We established the requirements to use these APIs in a stable manner. The availability of `fork` eliminates the need for existing, crude fuzzing techniques like persistent mode. To the best of our knowledge, this is the first practical counterpart of `fork` on Windows systems for fuzzing.

We implemented WINNIE in 7.1K lines of code (LoC). We applied WINNIE on 59 executables, including Visual Studio, ACDSee, ultraISO and EndNote. Our harness generator automatically synthesized candidate harnesses from execution traces, and 95% of them could be fuzzed directly with only minor modifications (i.e., $\leq$ 10 LoC). Our improved fuzzer also achieved 26.6× faster execution and discovered 3.6× more basic blocks than WinAFL, the state-of-the-art fuzzer on Windows. By fuzzing these 59 harnesses, WINNIE successfully found 61 bugs from 32 binaries. Out of the 59 harnesses, WinAFL only supported testing 29.

In summary, we make the following contributions:

- We identified the major challenges of fuzzing closed-source Windows applications;
- We developed WINNIE, which can automatically generate harnesses for Windows binaries to bypass GUI code;
- We implemented the first efficient Windows fork-server;
- WINNIE successfully generated efficient harnesses for 59 Windows binaries and found 61 bugs in 32 binaries.

To facilitate future research, we have open-sourced WINNIE at https://github.com/sslab-gatech/winnie.

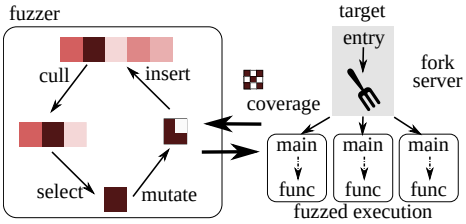

Fig. 2: Fuzzing overview. (1) The fuzzer maintains a queue of inputs. Each cycle, (2) it picks one input from the queue and (3) modifies it to generate a new input. (4) It feeds the new input into the fuzzed program and (5) records the code coverage. (6) If the execution triggers more coverage, the new input is added back into the queue.

II. BACKGROUND: WHY HARNESS GENERATION?

Fuzzing is a popular automated technique for testing software. It generates program inputs in a pseudo-random fashion and monitors program executions for abnormal behaviors (e.g., crashes, hangs or assertion violations). Since it was introduced, fuzzing has found tens of thousands of bugs [27].

Most popular fuzzers employ *greybox, feedback-guided* fuzzing. Under this paradigm, fuzzers treat programs like black boxes, but also rely on light-weight instrumentation techniques to collect useful feedback (e.g., code coverage) from each run. The feedback is used to measure how an input helps explore the program's internal states. Thus, a fuzzer can gauge how effective an input is at eliciting *interesting* behaviors from the program. Intuitively, since most bugs lie in the relatively complicated parts of code, the feedback guides the fuzzer towards promising parts of the program. This gives greybox fuzzers a decisive advantage over black-box fuzzers which blindly generate random inputs without any runtime feedback.

AFL [69], a popular Linux fuzzer, exemplifies greybox fuzzing in practice. Figure 2 depicts AFL's fuzzing process. The testing process is similar to a genetic algorithm. It proceeds iteratively, mutating and testing new inputs each round. Inputs which elicit bugs (i.e., crashes or hangs) or new code coverage from the program are selected for further testing, while other uninteresting inputs are discarded. Across many cycles, AFL learns to produce interesting inputs as it expands the code coverage map. Although simple, this strategy is surprisingly successful: several recent advanced fuzzers [4, 9, 14] follow the same high-level process. Overall, AFL-style, greybox fuzzing has proven extremely successful on Linux systems.

Although most recent research efforts focus on improving fuzzing Linux applications [4, 9, 14, 22, 39, 50, 69], Windows programs are also vulnerable to memory safety issues. Past researchers have uncovered many vulnerabilities by performing a manual audit [43]. In fact, Windows applications are especially interesting because they are commonly used on end-user systems. These systems are prime targets for malicious attackers [10, 17]. Automatic Windows testing would pave a way for researchers to look for bugs in many Windows programs while limiting manual code review. In turn, this would help secure the Windows ecosystem.

Unfortunately, no fuzzers can test Windows applications as effectively as AFL can test Linux applications. Table I compares Linux AFL with popular Windows fuzzers. WinAFL is a fork of AFL ported for Windows systems [57] and supports feedback-

Fuzzer	AFL	WinAFL	HonggFuzz	Peach	WINNIE
Feedback	✔	✔	✗	✗	✔
Forkserver	✔	✗	✗	✗	✔
Open-source	✔	✔	✔	✗	✔
Windows	✗	✔	✔	✔	✔

TABLE I: Comparison between various Windows fuzzers and Linux AFL. We compare several key features that we believe are essential to effective fuzzing. WINNIE aims to bring the ease and efficiency of the Linux fuzzing experience to Windows systems.

Program	Harness	GUI	Ratio	Program	Harness	GUI	Ratio
HWP-jpeg	117	4075	34.8×▲	Tiled	28	720	25.7×▲
Gomplayer	15	1105	73.6×▲	ezPDF	184	4397	23.8×▲
ACDSee	16	510	31.8×▲	EndNote	30	1461	23.8×▲

TABLE II: Execution times (ms) with and without GUI. GUI code dominates fuzzing execution time (35× slower on average). Thus, fuzzing harnesses are crucial to effective Windows application fuzzing. We measured GUI execution times by hooking GUI initialization code.

Attributes	Fudge	FuzzGen	Winnie
Binary	✗	✗	✔
Target OS	Linux	Linux/Android	Windows
Control-flow analysis	✔	✔	✔
Data-flow analysis	✔	✔	✗
Input analysis	Heuristic	-	Dynamic trace
Ptr / Struct analysis	Heuristic	Value-set analysis	Heuristic

TABLE III: Comparison of harness generation techniques. Most importantly, WINNIE supports closed-source applications by approximating source-level analyses. Fine-grained data-flow tracing is impractical without source code as it incurs a large overhead.

driven fuzzing. HonggFuzz supports Windows, but only for fuzzing binaries in dumb mode, i.e., without any coverage feedback [25]. Peach is another popular fuzzer with Windows support but requires users to write specifications based on their knowledge of the fuzzed program [16]. Overall, although there are several rudimentary fuzzers for Windows systems, we find that none offers fast and effortless testing in practice. In this paper, we aim to address these concerns and make Windows fuzzing truly practical. To do so, we must first examine what the major obstacles are.

A. The GUI-Based, Closed-Source Software Ecosystem

Compared to Linux programs, Windows applications have two distinguishing features: closed-source and GUI-based. First, many popular Windows applications are commercial products and thus closed-source, like Microsoft Office, Adobe Reader, Photoshop, WinRAR, and Visual Studio. As these commercial applications contain proprietary intellectual property, most of them are very unlikely to be open-sourced in the future. Second, Windows software is predominantly GUI-based. Unlike on Linux which features a rich command-line experience, essentially all of the aforementioned Windows programs are GUI applications. Due to the closed nature of the ecosystem, vendors rarely have an incentive to provide a command-line interface, as most end-users are most familiar with GUIs. In other words, the only way to interact with most programs' core functionality is through their GUI.

GUI applications pose a serious obstacle to effective fuzzing. First, GUI applications typically require user interaction to get inputs, and cannot be tested automatically without human intervention. Bypassing the GUI is nontrivial: it is slow to fully automate Windows GUIs with scripting [20]; meanwhile avoiding the user interface altogether usually requires a deep understanding of the application's codebase, as programmers often intertwine the asynchronous GUI code with the input processing code [30]. Second, GUI applications are slow to boot, wasting a lot of time on GUI initialization. Table II shows the startup times of GUI applications compared to a fully-CLI counterpart. In our experiments, GUI code often brought fuzzing speeds down from 10 or more executions per second to less than one. Naturally, fuzzing a CLI version of the application is absolutely essential. WinAFL [57] acknowledges this issue, and recommends users to create fuzzing harnesses.

B. Difficulty in Creating Windows Fuzzing Harnesses

It is a common practice to write fuzzing *harnesses* to test large, complicated software [8, 35]. In general, a harness is a relatively small program that prepares the program state for testing deeply-embedded behaviors. Unlike the original program, we can flexibly customize the harness to suit our fuzzing needs, like bypassing setup code or invoking interesting functions directly. Hence, harnesses are a common tactic for enhancing fuzzing efficacy in practice. For instance, Google OSS-Fuzz [29] built a myriad of harnesses on 263 open-source projects and found over 15,000 bugs [27].

Harnesses are especially useful when testing GUI-based Windows applications. First, we can program the harness to accept input from a command-line interface, thus avoiding user interaction. This effectively creates a dedicated CLI counterpart for the target program which existing fuzzers can easily handle. Second, using a harness avoids wasting resources on GUI initialization, focusing solely on the functionality at the heart of the program (e.g., file parsing) [3, 44, 55].

Unfortunately, Windows fuzzing faces a dilemma: due to the nature of the Windows ecosystem, effective fuzzing harnesses are simultaneously indispensable yet very difficult to create. In addition, due to the prevalence of closed-source applications, many existing harness generation solutions are inadequate [8, 35]. As a result, harness creation often requires in-depth reverse engineering by an expert, a serious human effort. In practice, this is a serious hindrance to security researchers fuzzing Windows applications.

Fudge and FuzzGen. Fudge [8] and FuzzGen [35] aim to automatically generate harnesses for open-source projects. Fudge generates harnesses by essentially extracting API call sequences from existing source code that uses a library. Meanwhile, FuzzGen relies on static analysis of source code to infer a library's API, and uses this information to generate harnesses. Table III highlights the differences between the existing solutions and WINNIE. Most crucially, Fudge and FuzzGen generally target *open-source* projects belonging to the *Linux* ecosystem, but WINNIE aims specifically to fuzz *COTS, Windows* software. Although it may seem that Linux solutions should be portable to Windows systems, the GUI-based, closed-source Windows software ecosystem brings new, unique challenges. As a result, these tools cannot be used to generate harnesses for Windows applications.

Fudge, FuzzGen, and WINNIE all employ heuristics to infer API control-flow and data-flow relationships. However, whereas Fudge and FuzzGen can rely on the availability of source code, WINNIE cannot as a large amount of API information is irrevocably destroyed during the compilation process, especially under modern optimizing compilers. Thus, although Fudge and FuzzGen's analyses are more detailed and fine-grained, they are crucially limited by their reliance on source code. This is the fundamental reason why these existing solutions are not applicable to Windows fuzzing. Hence, a new set of strategies must be developed to effectively generate fuzzing harnesses *in the absence of source code*.

III. CHALLENGES AND SOLUTIONS

WINNIE's goal is to automate the process of creating fuzzing harnesses in the absence of source code. From our experience, even *manual* harness creation is complicated and error-prone. Thus, before exploring automatic harness generation, we will first discuss several common difficulties researchers encounter when creating harnesses manually.

A. Complexity of Fuzzing Harnesses

Fuzzing harnesses must replicate all behaviors in the original program needed to reach the code that we want to test. These behaviors could be complex and thus challenging to capture in the harness. For instance, a harness may have to initialize and construct data structures and objects, open file handles, and provide callback functions. We identified four major steps to create a high-quality harness: ① target discovery; ② call-sequence recovery; ③ argument recovery; ④ control-flow and data-flow dependence reconstruction.

To illustrate these steps in action, we look into a typical fuzzing harness, shown in Figure 3. XnView is an image organizer, viewer and editor application [64]. Although the original program supports more than 500 file formats [63], our goal is to test the JPM parser, implemented in the library `ldf_jpm.dll`. Figure 3 shows the corresponding harness. First, the harness declares callback functions (lines 2-3), and initializes variables (lines 6 and 9). Second, the harness imitates the decoding logic of the original program: it opens and reads the input file (line 10), retrieves properties (lines 14-17), decodes the image (line 20), and closes it (line 23). Lastly, the harness declares the required variables (line 9) and uses them appropriately (lines 15, 17, 20 and 23). Conditional control flow based on return values is also considered to make the program exit gracefully upon failures (line 11).

❶ Target discovery. The first step of fuzzing is to identify promising targets that handle user inputs. This process can be time-consuming as, depending on the program, the input may be specified in a variety of ways, such as by filename, by file descriptor, or by file contents (whole or partial). In this example, the researcher should identify that the API `JPM_Document_Start` from `ldf_jpm.dll` library is responsible for accepting the user input through a pointer of an opened file descriptor (line 10).

❷ Call-sequence recovery. The harness must reproduce the correct order of all function calls relevant to the target library. In this example, there are total 10 API calls to be reconstructed in the full harness. Note that static analysis alone is not enough to discover all callsites. Due to the prevalence of indirect calls and jump tables, researchers must also use dynamic analysis to get the concrete values of the call targets.

```
1  // 1) Declare structures and callbacks
2  int callback1(void* a1, int a2) { ... }
3  int callback2(void* a1) { ... }
4
5  // 2) Prepare file handle
6  FILE *fp = fopen("filename", "rb");
7
8  // 3) Initialize objects, internally invoking ReadFile()
9  int *f0_a0 = (int*) calloc(4096, sizeof(int));
10 int f0_ret = JPM_Document_Start(f0_a0, &callback1, &fp);
11 if (f0_ret){ exit(0); }
12
13 // 4) Get property of the image
14 int f1_a2 = 0, int f4_a2 = 0;
15 JPM_Document_Get_Page_Property((void *)f0_a0[0], 0xA, &f1_a2);
16 ...
17 JPM_Document_Get_Page_Property((void *)f0_a0[0], 0xD, &f4_a2);
18
19 // 5) Decode the image
20 JPM_Document_Decompress_Page((void *)f0_a0[0], &callback2);
21
22 // 6) Finish the harness
23 JPM_Document_End((void *)f0_a0[0]);
```

Fig. 3: An example harness, synthesized by our harness generator. It tests the JPM parser inside the `ldf_jpm.dll` library of the application XnView. The majority of the harness was correct and usable out of the box. We describe the steps taken to create this harness in §III-A and in more detail in §IV. Low level details are omitted for brevity.

❸ Argument recovery. The harness must also pass valid arguments to each function call. Reconstructing these arguments is challenging: the argument could be a pointer to a callback function (like `&callback1` at line 10), a pointer to an integer (like `&f1_a2` at line 15), a constant (like `0xa` at line 15), or many other types. When manually constructing a harness, the researcher must examine every argument for each API call, relying on their expertise to determine what the function expects.

❹ Control-flow and data-flow dependence. It is oftentimes insufficient to simply produce a list of function calls in the right order. Moreover, libraries define implicit semantic relationships among APIs. These relationships manifest in control-flow dependencies and data-flow dependencies. For example, a conditional branch between API calls may be required for the harness to work, like the if-statement at line 11 of the example. Alternatively, one API may return or update a pointer which is used by a later API call. Unless these relationships are respected, the resulting harness will be incorrect, yielding false positives and spurious crashes. For example, the above code updates array `f0_a0` at line 10, and uses the first element in lines 15, 17, 20, and 23. In the absence of source code, this step is extremely challenging, and even the most advanced harness generator cannot guarantee correctness. Human intuition and experience can supplement auto-analysis when reverse-engineering.

B. Limitations of Existing Solutions

As Windows does not provide fast process cloning machinery (e.g., Linux's `fork`), fuzzers usually start each execution from the very beginning. Considering the long start-up time of Windows applications (see Table II), each re-execution wastes a lot of time to reinitialize the program. Existing solutions (e.g., WinAFL) resort to a technique known as *persistent mode* to

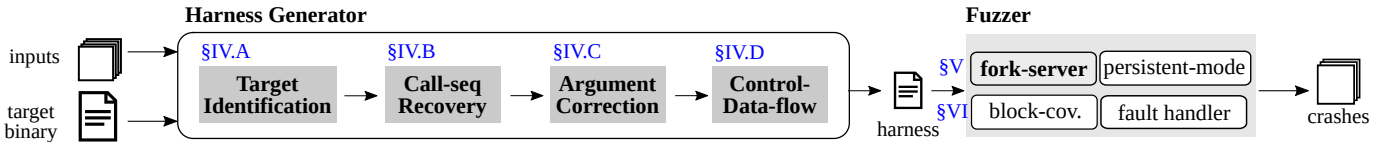

Fig. 4: Overview of WINNIE. Given the target program and a set of sample inputs, WINNIE aims to find security vulnerabilities. It uses a harness generator to synthesize simple harnesses from the execution trace, and then fuzzes harnesses efficiently with our implementation of `fork`.

overcome the re-execution overhead [68]. In persistent mode, the fuzzer repeatedly invokes the target function in a tight loop *within the same process*, without reinitializing the program each iteration. To realize the most performance gains, one generally aims to test as many inputs as possible per new process.

While persistent mode partially addresses the performance issues of Windows fuzzing, its efficacy is limited by its strict requirements on the loop body. Specifically, persistent mode expects harnesses to behave like *pure functions*, meaning that harnesses avoid any side-effects, such as leaking memory or modifying global variables. Otherwise, each execution would start from a different program state. Since the harness is repeatedly looped for thousands of iterations, even the smallest side-effects will gradually accumulate over time, finally leading to problems like memory leaks, unreproducible crashes and hangs, and unreliable coverage. For example, a program that leaks 1MB of memory per iteration will reach WinAFL's default memory limit and be terminated. We experienced such errors very often in practice, and discuss more details later in §VII-A.

Many side-effect errors from persistent mode are difficult to debug or difficult to circumvent. A common issue is that persistent mode cannot continue if the target function does not return to the caller. For example, a program can implement error handling by simply terminating the program. Because most inputs generated during fuzzing are invalid (albeit benign), this still demands constant re-execution, severely degrading performance. Another common problem is that a program will open the input file in exclusive mode (i.e., other processes cannot open the same file) without closing it. This prevents the fuzzer from updating the input file in the next iteration, breaking persistent mode. Problems like these limit the applicability and scalability of persistent mode fuzzers.

C. Our Solutions

We propose WINNIE, an end-to-end system that addresses aforementioned obstacles to effectively and efficiently fuzz Windows applications. WINNIE contains two components, a harness generator that synthesizes harnesses for closed-source Windows programs with minimal manual effort (§IV), and a fuzzer that can handle uncooperative target applications with our efficient `fork` implementation (§V). Figure 4 shows an overview of our system. Given the program binary and sample inputs, our tracer runs the program and meanwhile, collects dynamic information about the target application, including API calls, arguments and memory contents. From the trace, we identify interesting fuzzing targets that handle user input, including functions in external libraries and locations inside the main binary. For each fuzzing target, our harness generator analyzes the traces and reconstructs related API sequences as a working harness. We test the generated harnesses to confirm their robustness and effectiveness, and then launch fuzzing instances with our fork-server to find bugs. In the following sections, we will use the harness shown in Figure 3 as an example to explain the design of each component of WINNIE.

Class	Type	What to record
① **Module**	string	name, path, module
② **Call/Jump**	inter-module	thread id, caller, callee, symbols, args
	intra-module	same as above, only for main .exe
③ **Return**	inter-module	thread id, callee, caller, retval
	intra-module	same as above, only for main .exe
④ **Arg/RetVal**	constants	concrete value
	pointers	address and referenced data (recursively)

TABLE IV: Dynamic information collected by the tracer. We record detailed information about every inter-module call. We also record the same information for intra-module calls within the main binary. If the argument or return value is a pointer, we recursively dump memory around the pointed location. We then use this information to construct fuzzing harnesses (§IV).

IV. HARNESS GENERATION

To generate the harness, WINNIE followed the four steps previously outlined in §III-A. Consider XnView as an example:

❶ For *target discovery* (§IV-A), we trace XnView while opening several JPM files, and then search the traces for input-related APIs, such as `OpenFile` and `ReadFile`.

❷ For *call-sequence recovery* (§IV-B), we search the traces for function calls related to the fuzzing target. In the example, we find all the function calls related to the chosen library (lines 10, 15, 17, 20 and 23). We put the call-sequence into the harness, forming a *harness skeleton*. The skeleton is now more-or-less a simple series of API calls, which we then flesh out further.

❸ For *argument recovery* (§IV-C), we analyze the traces to deduce the prototype for each function in the call sequence. The traces contain verbose information about APIs between the main binary and libraries, like arguments and return values.

❹ Finally, we establish the *relationships* (§IV-D) among the various calls and variables presented in the harness skeleton and emit the final code after briefly *testing* (§IV-E) the candidate harness. WINNIE also points out complicated logic potentially missed by our tracer (such as the callback function at line 20) as areas for further improvement.

A. Fuzzing Target Identification

In this step, WINNIE evaluates whether the program can be fuzzed and tries to identify promising target functions. We begin by performing dynamic analysis on the target program as it processes several chosen inputs. Table IV shows a detailed list of items that the tracer captures during each execution. ① We record the name and the base address of all loaded modules. ② For each call and jump that transfers control flow between modules, our tracer records the current thread ID, the caller and

callee addresses, symbols (if available), and arguments. Without function prototype information, we conservatively treat all CPU registers and upper stack slots as potential arguments. ③ We record return values when encountering a return instruction. ④ If any of values fall into accessible memory, we conservatively treat it as a pointer and dump the referenced memory for further analysis. To capture multi-level pointer relationships (e.g., double or triple pointers), we repeat this process recursively. For pointers, we also recognize common string encodings (e.g., C strings) and record them appropriately.

Using our captured traces, we look for functions which are promising fuzzing targets. It is commonly believed that good fuzzing targets have two key features [58, 68]: the library accepts the user-provided file path as the input, and it opens the file, parses the content and closes the file. We use these two features to find candidate libraries for fuzzing. Specifically, for each function call, we check whether one of its arguments points to a file path, like `C:\my_img.jpm`. To detect user-provided paths, our harness generator accepts filenames as input. Next, we identify callers of well-known file-related APIs such as `OpenFile` and `ReadFile`. If a library has functions accepting file paths, or invokes file-related APIs, we consider it is an input-parsing library and treat it as a fuzzing candidate.

WINNIE also identifies library functions that do not open or read the file themselves, but instead accept a file descriptor or an in-memory buffer as input. To identify functions accepting input from memory, our tracer dissects pointers passed to calls and checks if the referenced memory contains any content from the input file. We also verify that the appropriate file-read APIs were called. To find functions taking file descriptors as inputs, we inspect all invocations of file-open APIs and track the opened file descriptors. Then, we check whether the library invokes file-related APIs on those file descriptors.

Our harness generator focuses primarily on the external interfaces a library exposes. On the other hand, we do not record control flow within the same module as these represent libraries' internal logic. Because invoking the API through those interfaces models the same behavior as the original program, inter-module traces are sufficient for building an accurate harness. However, we treat the main executable as a special case and record all control-flow information within it. This is because the main executable is responsible for calling out to external libraries. Thus, we also search the intra-module call-graph of the main executable for suitable fuzzing targets.

WINNIE then expands its search to within the main binary by analyzing its call-graph. Specifically, WINNIE begins at the *lowest common ancestor (LCA)* of I/O functions and the parsing library APIs we previously identified. In a directed acyclic graph, the LCA of two nodes is the deepest one that can reach both. In our case, we search for the lowest node in the main binary's callgraph that satisfies two criteria. First, it should be before the file-read operation so that our fuzzer can modify the input. Note that even if the fuzzed process has opened the input file, we still can modify it so the program uses the new content. Second, the LCA should reach locations that invoke parsing functions. Figure 5 shows an example callgraph from the program ACDSee. The function at address `0x5cce80` is the LCA as it reaches two file-related APIs (i.e., `OpenFile` and `ReadFile`) and also invokes the parsing functionality in `ide_acdstd.apl`. We also consider the LCA's ancestors (e.g., `main()`) as fallback candidates, if the immediate LCA does not yield a working harness. In cases where a working LCA is found, it often is sufficient for making an effective harness.

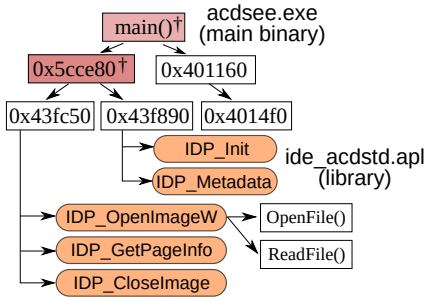

Fig. 5: A simplified call-graph of the ACDSee program. WINNIE analyzes the call-graph for fuzzing possible targets, focusing on inter-module calls and I/O functions. We look for functions that can reach both I/O functions and also the interesting ones we wish to fuzz. "†" indicates such functions, known as *LCA candidates* (§IV-A).

Our tool can also optionally use differential analysis to refine the set of candidate fuzzing targets. Given two sets of inputs, one triggering the target functionality and another not triggering, WINNIE will compare the two execution traces and locate the library functions that are specific to the target functionality. We discard the other functions which are present in both sets of traces. This feature helps deal with multi-threaded applications where only one thread performs operations related to the input file. In any case, differential analysis is optional; it only serves as an additional criteria to improve harness generation.

B. Call-sequence Recovery

Now that we have identified a candidate fuzzing target, our goal in this step is to reproduce a series of API calls which will correctly reach and trigger the functionality we wish to fuzz. We call such an API sequence a *harness skeleton*. We search the traces for function calls related to that library and copy them to the harness skeleton (lines 10, 15, 17, 20, 23 in Figure 3). We also reconstruct the functions' prototypes (e.g., argument count and types) with hybrid analysis: we combine the static analysis provided by IDA Pro [31] or Ghidra [2] with concrete information retrieved from the dynamic execution traces. Namely, we apply pointer types to arguments that were valid addresses in the traces, as the static analysis can misidentify pointer arguments as integers. Lastly, we attach auxiliary code that is required to make the harness work, like a `main` function, forward function declarations, and helper code to open or read files (line 6).

Special care must be taken to handle applications which use multiple threads. In that case, we will only consider the threads that invoke file-related APIs. This is to avoid adding irrelevant calls that harm the correctness of the harness. We encountered several programs that exhibit this behavior, such as GomPlayer, which had hundreds of irrelevant function calls in the execution trace. When the program creates multiple threads within the same library, the trace records an interleaving of many threads' function calls combined. However, since we recorded the thread IDs in our previous step, we can untangle the threads to focus on them individually. With the per-thread analysis, we can narrow the number of calls down to just seven.

C. Argument Recovery

In this step, we reconstruct the arguments that should be passed to each API call in the call sequence recovered in the previous step. WINNIE attempts to symbolize the raw argument values recorded in the traces into variables and constants. First, we identify pointer arguments. We do so empirically through differential analysis of the trace data. Specifically, the tracer runs the program with the same input twice, both times with address space layout randomization (ASLR) enabled [49]. Because ASLR randomizes memory addresses across different runs, two pointers passed to the same call site will have different, pseudo-random values that are accessible addresses both times. If this is the case, we can infer that the argument is a pointer. For pointer arguments, we use the concrete memory contents from the trace, dissecting multiple levels of pointers of necessary. Otherwise, we simply consider the value of the argument itself.

Next, we determine whether the argument is *static* or *variable*. Values which vary from execution to execution are *variable*, and we define names for variables and replace their uses with new names. Values which remain constant between runs are *static*, and we simply pass them as the constant value seen in the trace (like `0xA` and `0xD` in Figure 3).

D. Control-Flow and Data-Flow Reconstruction

WINNIE analyzes the program to reflect control-flow and data-flow dependencies in the harness. Control-flow dependencies represent how the various API calls are logically related (e.g., the if-statement on line 11 in Figure 3). To find control-flow dependencies, we apply static analysis. Specifically, WINNIE analyzes the control-flow between two API calls for paths from the return value of the invoked function to a termination condition (e.g., `return` or `exit()`). If such a path is found, WINNIE duplicates the decompiled control-flow code (e.g., `if`-statements). The current version of WINNIE avoids analyzing complex flows involving multiple assignments or variable operands in the conditional statement; we leave such cases to a human expert. This is important for accurate harness generation: neglecting control-flow dependencies causes incorrect behavior. For example, consider a harness that fails to reflect an early exit error handling condition in the original program. The program under normal execution would terminate immediately, but the harness would proceed onwards to some unpredictable program state. These kinds of mistakes lead to unreproducible crashes (i.e., false positives).

Data-flow dependencies represent the relationships among function arguments and return values. To find data-flow dependencies, WINNIE tries to connect multiple uses of the same variable between multiple call sites (e.g., `f0_a0` in Figure 3). We consider the following possible cases:

- **Simple flows from return values.** Return values of past function calls are commonly reused as arguments for later calls. We detect these cases by checking if an argument always has the same value as a past return value. We only do this for whose values exceed a certain threshold. If we connected any frequently observed values (e.g., connect return value `0` as the next argument), we may generate incorrect harnesses; this resolves many common cases where functions return object pointers.
- **Points-to relationships.** Some arguments are retrieved from memory using pointers returned by previous code. For instance, an API may return a pointer, whose pointed contents are used as an argument in a later API call. In the example harness in Figure 3, line 23 uses an argument `f0_a0` that is loaded from memory, initialized by the API `JPM_Document_Start`. When we detect these points-to relationships in the trace, we reflect them in the harness as pointer dereferences (i.e., `*p`). WINNIE also supports multi-level points-to relationships (e.g., double and triple pointers), thanks to the tracer's recursive memory dumping.
- **Aliasing.** WINNIE defines a variable if it observes one or more repeated usages. In other words, if the same non-constant value is used twice as an argument, then the two uses are considered aliases forming a single variable.

E. Harness Validation and Finalization

Although it covers most common cases, WINNIE's harness generator is not foolproof. WINNIE points out parts of the harness that is unsure about and provides suggestions to help users further improve it. ① We report distant API calls where the second API's call site is far from the first. In such cases, our API-based tracer might have missed some logic between two API calls. ② We highlight code pointer arguments to users, which could represent callback function pointers or virtual method tables. ③ We provide information about file operations as they are generally important during harness construction.

Once a fuzzing harness has been generated, we perform a few preliminary tests to evaluate its effectiveness. First, we check the harness's stability. We run the harness against several normal inputs; if the harness crashes, we immediately discard it. Second, we evaluate the harness's ability to explore program states. Specifically, we fuzz the harness for a short period and check whether the code coverage increases over time. We discard harnesses that fail to discover new coverage. Lastly, we test the execution speed of the harness. Of all stable, effective harnesses, we present the fastest ones to the user.

WINNIE's goal is to generate harnesses automatically. However, the general problem of extracting program behaviors from runtime traces without source code is very challenging so there will always be cases it cannot cover. Thus, we aim to handle most common cases to maximize WINNIE's ability to save the human researcher's time. We observe that in practice it produces good approximations of valid harnesses, and most of them can be fuzzed with only minor modifications as shown in Table VIII. We discuss our system's limitations and weaknesses in §VII-C and §VIII.

V. FAST PROCESS CLONING ON WINDOWS

Fork indeed exists on Windows systems [15], but existing work fails to provide a stable implementation. To support efficient fuzzing of Windows applications, we reverse-engineered various internal Windows APIs and services and identified a key source of instability. After overcoming these challenges, we were able to implement a practical and robust fork-server for Windows fuzzing. Specifically, our implementation of the Windows fork corrects the problems related to the CSRSS, which is a user-mode process that controls the underlying layer of the Windows environment [54]. If a process is not connected

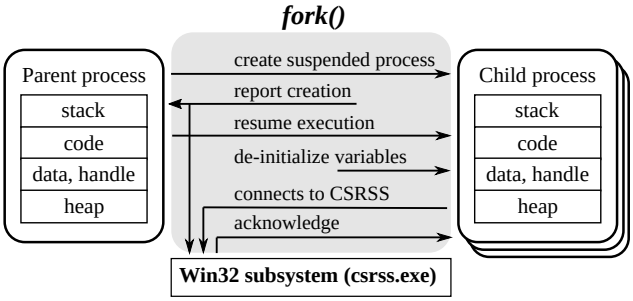

Fig. 6: Overview of fork() on Windows. We analyzed various Windows APIs and services to achieve a CoW fork() functionality suitable for fuzzing. Note that fixing up the CSRSS is essential for fuzzing COTS Windows applications: if the CSRSS is not re-initialized, the child process will crash when accessing Win32 APIs. We include a detailed technical description in appendix §X-A.

Fork()	Re-execute	Forkserver				
	CreateProcess	WINNIE	Cygwin	WSL(v1)	WSL(v2)	Linux
Supports PE files?	✔	✔	✔	✗	✗	✗
Copy-on-Write?	✗	✔	✗	✔	✔	✔
Speed (exec/sec)	91.9	310.9	72.8	442.8	405.1	4907.5

TABLE V: Comparison of `fork()` implementations. Cygwin is not CoW, and WSL does not support Windows PE binaries. WINNIE's new fork API is therefore the most suitable for Windows fuzzing.

to the CSRSS, it will crash when it tries to access Win32 APIs. Note that virtually every Windows application uses the Win32 API. Our fork correctly informs the CSRSS of newly-created child processes, as shown in Figure 6. Connecting to the CSRSS is not trivial for forked processes: for the call to succeed, we must manually *de-initialize* several undocumented variables before the child process connects. We provide a detailed technical description of our implementation in the appendix §X-A.

To the best of our knowledge, our fork implementation is the only one that can support fuzzing commercial off-the-shelf (COTS) Windows applications. Table V shows a comparison of process creation techniques on Windows and Linux. `CreateProcess` is the standard Windows API for creating new processes with a default program state, used by WinAFL. New processes must re-execute everything from the beginning, wasting a lot of time on GUI initialization code, shown in Table II. Persistent mode [68] aims to mitigate the re-execution overhead, but is impractical due to the numerous problems outlined in §III-B. Thus, our goal is to avoid re-executions altogether by introducing a `fork`-style API. Meanwhile, Cygwin's `fork` implementation is not designed for COTS Windows applications. It works by manually copying the program state after calling `CreateProcess`. It also suffers from problems related to address space layout randomization [6]. The Windows Subsystem for Linux (WSL) is designed for running Linux ELF binaries on Windows. Thus, we cannot use it for testing Windows PE binaries, even if it is faster [41]. Our fork implementation achieves a speed comparable to the WSL fork, and most importantly, supports Windows PE applications.

Verifying the Fork Implementation. We ran several test programs under our fork-server to verify its correctness. First, verified that each child process receives a correct copy of the global program state. We checked various the values of global variables in test programs before and after forking a new process. For example, we incremented a global counter in the parent process after each fork and verified that the child process received the old value. Second, to verify that the `fork` implementation is CoW (*copy-on-write*), we initialized large amounts of memory in the parent process before forking. Because the memory footprint of the parent process did not affect the time taken by `fork`, we concluded that our implementation is indeed CoW.

We also briefly measured the speed of `fork` with WinAFL's built-in test program as shown in Table V. On an Intel i7 CPU, we were able to call our `fork` 310.9 times/sec per core with a simple program, which is 4.2× faster than Cygwin's *No-CoW* fork and ∼1.3× slower than the WSL fork. Since we are not using the same fork mechanism as the one provided by the Linux kernel but instead mimicking its CoW behavior using the Windows API, the execution speed is nowhere as fast (e.g., >5,000 execs/sec). Even if Windows implementation of `fork` is slower than Linux's, the time regained from avoiding costly re-executions easily makes up for the overhead of `fork`. Moreover, the process creation machinery on Windows is slow in general: in our experiments, ordinary `CreateProcess` calls (as used by WinAFL) only reach speeds of less than 100 execs/sec. Overall, we believe that the reliability and quality of our Windows fork-server is comparable to ones used for fuzzing on Unix systems.

Idiosyncrasies of Windows Fork. Our `fork` implementation has a few nuances due to the design of the Windows operating system. First, if multiple threads exist in the parent process, only the thread calling `fork` is cloned. This could lead to deadlocks or hangs in multi-threaded applications. Linux's `fork` has the same issue. To sidestep this problem, we target deeply-nested functions that behave in a thread-safe fashion. For example, in the program UltraISO, we bypassed the GUI and fuzzed the target function directly, shown in Table VIII. Second, handle objects, the Windows equivalent of Unix file descriptors, are not inherited by the child process by default. To address this issue, we enumerate all relevant handles and manually mark them inheritable. Third, because the data structures involved in fork-related APIs differ from version to version of Windows, it is impractical to support all possible installations of Windows. Nevertheless, our fork-server supports all recent builds of Windows 10. Since Windows is very backwards-compatible, we do not see this as a significant limitation of our implementation.

Category	Component	Lines of code
Harness generator	Dynamic tracer	1.6K LoC of C++
	Synthesizer	2.0K LoC of Python
Fuzzer	Fuzzer	3.0K LoC of C++
	Fork library	0.5K LoC of C++

TABLE VI: WINNIE components and code size

VI. IMPLEMENTATION

We prototyped WINNIE with 7.1K lines of code (shown in Table VI). WINNIE supports both 32- and 64-bit Windows PE binaries. We built our fuzzer on top of WinAFL and implemented the `fork` library from the scratch. The tracer relies on Intel Pin [40] for dynamic binary instrumentation.

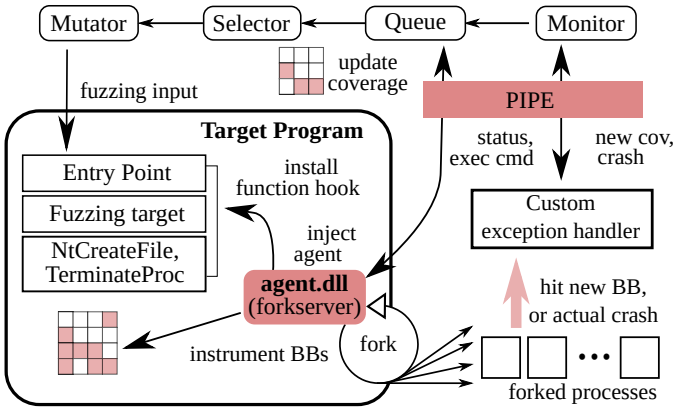

Fig. 7: Overview of WINNIE's fuzzer. We inject a fuzzing agent into the target. The injected agent spawns the fork-server, instruments basic blocks, and hooks several functions. This improves performance (§VI-A) and sidesteps various instrumentation issues (§VI-B).

A. Fuzzer Implementation

Figure 7 shows an overview of our fuzzer. We inject a fuzzing agent `agent.dll` into the target program, which cooperates with the fuzzer using a pipe for bidirectional communication. This architecture helps assuage the most uncooperative of fuzzing targets.

The fuzzing agent is injected as soon as the program loads, before any application code has begun executing. Once injected, the agent first hooks the function specified by the harness and promptly returns control to the target application. Then, the target application resumes and initializes itself. The application halts once it reaches the hooks, and the fuzzing agent spins up the fork-server. Since we spin up the fork-server only at some point deep within the program, initialization code only runs once, massively improving performance.

Our fuzzer works as follows: ❶ The fuzzing agent, which contains the fork server, is injected into the target application. The injected agent ❷ installs function hooks on the entry point and the target function, and ❸ instruments all basic blocks so it can collect code coverage. ❹ Then, the fuzzer creates forked processes. Using the pipe between the fuzzer and target processes, ❺ the agent reports program's status and ❻ the fuzzer handles coverage and crash events.

B. Reliable Instrumentation

Collecting code coverage from closed-source applications is challenging, specially for Windows applications. WinAFL uses two methods to collect code coverage: one using dynamic binary instrumentation using DynamoRIO [57], and another using hardware features through Intel PT (IPT) [59]. Unfortunately, DynamoRIO and IPT are prone to crashes and hangs. In our evaluation, WinAFL was only able to run 26 of 59 targets.

To address this issue, we discard dynamic binary instrumentation in favor of *fullspeed fuzzing* [47] to collect code coverage. Fullspeed fuzzing does not introduce any overhead except when the fuzzer discovers a new basic block. Based on boolean basic block coverage, fullspeed fuzzing only considers there to be new coverage when a new basic block is visited. To implement this, we patch all basic blocks of the tested program with an `int 3` instruction. Then, we fuzz the patched program and wait for the execution to reach a new block. When reached, the first byte of the new block is then restored so that it will no longer generate exceptions in the future. Since encountering new basic blocks is rare during fuzzing, fullspeed fuzzing has negligible overhead and can run the target application at essentially native speed. Breakpoints need only be installed *once* thanks to the fork-server: child processes inherit the same set of breakpoints as the parent. We noticed that this is an important optimization as we observe Windows applications easily contain a massive number of basic blocks (e.g., >100K).

VII. EVALUATION

We evaluated WINNIE on real-world programs to answer the following questions:

- **Applicability of WINNIE.** Can WINNIE test a large variety of Windows applications? (§VII-A)
- **Efficiency of `fork`.** How efficient is `fork` on versus other modes of fuzzing like persistent mode? (§VII-B)
- **Accuracy of harness generation.** How effectively can WINNIE create fuzzing harnesses from binaries? (§VII-C)
- **Finding new bugs.** Can WINNIE discover new program states and bugs from real world applications? (§VII-D)

Evaluation Setup. Our evaluation mainly compares WINNIE with WinAFL. Other Windows fuzzers either do not support feedback-driven fuzzing (e.g., Peach [16]), or cannot directly fuzz Windows binaries (e.g., HonggFuzz [25]). We configured WinAFL to use basic-block coverage as feedback and used persistent-mode to maximize performance. Our evaluation of WinAFL considers two modes, the DynamoRIO mode (WinAFL-DR) where WinAFL relies on dynamic binary instrumentation, and the PT mode where WinAFL uses the Intel PT hardware feature to collect code coverage. We enlarged the Intel PT ring buffer sizes from 128 kilobytes to 512 kilobytes to mitigate data-loss issues [34]. We performed the evaluation on an Intel Xeon E5-2670 v3 (24 cores at 2.30GHz) and 256 GB RAM. All the evaluations were run on Windows 10, except WinAFL-DR, which was run on Windows 7 as it did not run properly under Windows 10.

Target Program Selection. We generated 59 valid fuzzing harnesses with WINNIE. We ran all 59 programs test the applicability of WINNIE (§VII-A). For the other evaluations (§VII-B to §VII-D), we randomly chose 15 GUI or CLI applications among the 59 generated harnesses due to limited hardware resources (i.e., 15 apps × 24 hrs × 5 trials = 5,400 CPU hrs). We aimed to show that WINNIE can fuzz complicated GUI applications and that WINNIE also outperforms existing solutions on CLI programs. Thus, we chose a mixture of both types of binaries from a variety of real-world applications. For this evaluation, we mainly focused on programs that accept user input from a file, as their parsing components are usually complex (i.e., error-prone) and handle untrusted inputs.

A. Applicability of WINNIE

Figure 8 shows that WINNIE supports running a wider variety of Windows applications than WinAFL. Specifically, WINNIE successfully generates working harnesses for all programs and is able to test them efficiently. WinAFL-IPT failed to run 33 of out 59 harnesses (55.9%) while WinAFL-DR

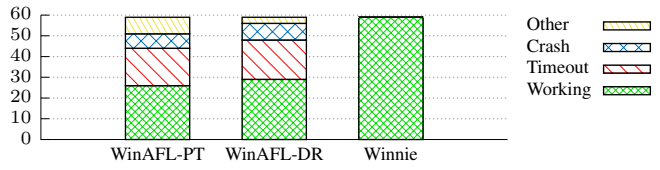

Fig. 8: Applicability of WINNIE and WinAFL. Among 59 executables, WinAFL-IPT and WinAFL-DR failed to run 33 and 30 respectively, whereas WINNIE was able to test all 59 executables. We provide a detailed breakdown of all 59 programs in appendix §X-B.

Program	Without Fork				Fork			
	Leak	Hang†	Speed	Cov.	Speed		Coverage	
7z			5.2	1430	49.3	(9.5×▲)	2117	(1.5×▲)
makecab	✗		14.8	576	49.4	(3.3×▲)	1020	(1.8×▲)
GomPlayer		✗	0.4	201	25.9	(64.7×▲)	1496	(7.4×▲)
Hwp-jpeg	✗		4.2	1045	25.9	(6.2×▲)	1847	(1.8×▲)
Hwp-tiff	✗	✗	0.3	1340	26.2	(87.3×▲)	2301	(1.7×▲)
EndNote			5.3	68	89.5	(16.9×▲)	693	(10.2×▲)
Total	3/6	2/6				(**31.3**×▲)		(**4.0**×▲)

TABLE VII: Evaluation of fork(). We ran six applications that both WinAFL and WINNIE could fuzz for 24 hours. We compared their speed and checked for memory and handle (i.e., file descriptor) leaks. `fork` not only improves the performance, but also mitigates resource leaks. Hang† means an execution speed slower than 1.0 exec/sec.

failed to run 30 (50.8%). For each program that WinAFL failed, we analyzed the cause and present the details in the appendix §X-B. Execution timeouts during the dry-runs dominate all failed cases of WinAFL (18 for WinAFL-IPT and 19 for WinAFL-DR). Specifically, before the fuzzing fully begins, WinAFL launches a few dry-runs to verify that the fuzzing setup is valid (e.g., harness quality). If the program times out during the dry-run, WinAFL will not be able to continue the testing. The second main failure mode was crashing during the dry-run. This contributed seven failures for WinAFL-IPT and eight for WinAFL-DR. We provide several case studies to understand why WinAFL fails to test these programs:

Unexpected Change in Global State. ① mspdbcmf.exe is a PDB (debug symbol file) conversion tool, and WinAFL failed with a timeout error. When the fuzzer executes the same function iteratively, the program falls into a termination condition, due to a corrupted global variable. In particular, the program assigns a non-zero value to the global variable (`g_szPdbMini`) in the first execution, and the changed value makes the application terminate during the second execution. In other words, the root cause was that the target function was not idempotent. Unfortunately, WinAFL misclassifies this unexpected termination as a timeout, and thus the fuzzer quits after the dry-run. ② ML.exe (**M**acro assembler and **L**inker) is an assembler program in Visual Studio that crashes when fuzzing begins. Similar to the aforementioned timeout issue, a crash happens at the second execution of the main function. In the first execution, the target program checks the global flag (i.e., `fHasAssembled`) to determine whether the assembly is done and then initializes necessary heap variables. Once the program finishes the first time, it changes the global flag to *true*. In the second execution, the program's control flow diverges because the `fHasAssembled` flag is *true*. This ultimately leads to a crash when it tries to access the uninitialized heap variable.

Program	Target	Size	API Calls	LoC	Fixed (LoC)	(%)
ACDSee	IDE_ACDStd.apl	3007K	19	506	CB (38), ST (174)	34.3
HWP-jpeg	HncJpeg10.dll	220K	3	92	CB (7), ST (8)	16.3
ezPDF	Pdf2Office.dll	3221K	4	112	CB (2), ST (8)	8.9
HWP-tiff	HncTiff10.dll	630K	3	82	CB (7)	8.5
UltraIso	UltraISO.exe	5250K	1	57	CB (2)	3.5
XnView	ldf_jpm.dll	692K	10	199	CB (4), pointer (2)	3.0
Gomplayer	avformat-gp.dll	4091K	7	116	pointer (2)	1.7
file	magic1.dll	147K	3	96	0	0.0
EndNote	PC4DbLib	2738K	1	55	0	0.0
7z	7z.exe	1114K	1	55	0	0.0
makecab	makecab.exe	50K	1	55	0	0.0
Tiled	tmxviewer.exe	113K	1	55	0	0.0
mspdbcmf	mspdbcmf.exe	1149K	1	55	0	0.0
pdbcopy	pdbcopy.exe	726K	1	55	0	0.0
ml	ml.exe	476K	1	55	0	0.0

CB: Callback function, **ST:** Custom struct

TABLE VIII: Harnesses generated by WINNIE. The majority of the harnesses worked out of the box with few modifications. Some required fixes for callback and struct arguments, which we discuss below. For a complete table of all 59 harnesses, see appendix Table XII.

IPT Driver Issues. The dynamic binary instrumentation adopted by WinAFL-IPT had unknown issues and sometimes prevented WinAFL from collecting code coverage. For example, for the program KGB archiver, we observed that the fuzzer could not receive any coverage due to a Intel-PT driver error.

B. Benefits of Fork

We tested whether `fork` makes fuzzing more efficient. To do so, we ran the selected programs under our fuzzer in `fork` mode, while we set WinAFL to create a new process for each execution (re-execution mode). Both of these configurations can run the target program reliably. As shown in Table VII, `fork` improves fuzzing performance: compared to re-execution mode, WINNIE achieved 31.3× faster execution speeds and discovered 4.0× more basic blocks. In particular, GomPlayer and EndNote recorded 64.7× and 87.3× faster executions and revealed 7.4× and 10.2× more basic blocks respectively.

We also evaluated whether `fork` makes fuzzing more stable. We configured WinAFL to use persistent mode, which runs a specific target function in a loop. Then, we tracked the system's memory and resource usage over time while fuzzing. Almost immediately, we observed memory leaks in the persistent mode harnesses for HWP-jpeg, HWP-tiff, and makecab. The HWP-jpeg and HWP-tiff harnesses also leaked file handles, which would lead to system handle exhaustion if the fuzzer runs for a long time. These types of leaks tend to cause fuzzing to unpredictably fail after long periods of fuzzing, creating a big headache for the human researcher. We explain this in further detail in §III-B. `fork` prevented the memory leaks and file handle leaks, improving stability. We further discuss the advantages and disadvantages of persistent mode in §VIII.

C. Efficacy of Harness Generation

In this section, we evaluate how well WINNIE helps users create effective fuzzing harnesses. To do so, we diffed the initial and final harness code in our evaluation. We analyzed the fixes required to make the harnesses work, and present the findings in Table VIII and Table IX. We also include general information about all 59 harnesses in appendix §X-B. As shown, the majority of the harnesses worked with no modifications. On

Program	Vendor	Input	GUI?	Size	Speed (exec/sec)			Coverage (# of new BBs)			p-value		Applied heuristics						
					W-DR	W-PT	WINNIE	W-DR	W-PT	WINNIE	W-DR	W-PT	T	L	DF	CS	CB	CF	DF
makecab	Windows 10	.txt	CLI	50KB	228.2	21.3	49.4	762	982	1020	<0.001	<0.001	✔	✔					
HWP-jpeg	Hancom 20	.jpg	GUI	220KB	25.2	21.0	25.9	1821	1498	1847	0.12	<0.001	✔			✔	✔		✔
7z	7-Zip	.7z	Both	1,114KB	8.7	17.0	49.3	1435	1530	2117	<0.001	<0.001	✔	✔					
EndNote	Clarivate	.pdt	GUI	2,738KB	2.1	50.4	89.5	8	37	693	<0.001	<0.001	✔			✔		✔	✔
Gomplayer	GOM Lab	.mp4	GUI	4,091KB	0.2	0.6	25.9	194	1068	1496	<0.001	<0.001	✔		✔	✔			✔
HWP-tiff	Hancom 20	.tif	GUI	630KB	0.2	✗	26.2	1279	✗	2301	<0.001	N/A	✔			✔	✔		✔
Tiled	T. Lindeijer	.tmx	Both	113KB	✗	✗	8.7	✗	✗	36	N/A	N/A	✔	✔					
file	libmagic	.png	CLI	147KB	✗	✗	52.5	✗	✗	116	N/A	N/A	✔	✔					
UltraISO	Ultra ISO	.iso	GUI	5,250KB	✗	✗	45.3	✗	✗	1558	N/A	N/A	✔	✔			✔		
ezPDF	Unidocs	.pdf	GUI	3,221KB	✗	✗	18.9	✗	✗	6355	N/A	N/A	✔			✔	✔		✔
XnView	XnSoft	.jpm	GUI	692KB	✗	✗	23.2	✗	✗	16702	N/A	N/A	✔		✔	✔	✔	✔	✔
mspdbcmf	VS2019	.pdb	CLI	1,149KB	✗	✗	8.1	✗	✗	9637	N/A	N/A	✔	✔					
pdbcopy	VS2019	.pdb	CLI	726KB	✗	✗	28.5	✗	✗	3302	N/A	N/A	✔	✔					
ACDSee	ACDsee	.png	GUI	3,006KB	✗	✗	63.1	✗	✗	618	N/A	N/A	✔		✔	✔	✔	✔	✔
ml	VS2019	.asm	CLI	476KB	✗	✗	44.0	✗	✗	2399	N/A	N/A	✔	✔					

T: Target identification, **L:** LCA, **DF:** Differential analysis, **CS:** Call sequence, **CB:** Callback, **CF:** Control-flow (exit, loop), **DF:** Data-flow (constant/variable, pointer)

TABLE IX: Comparison of WINNIE against WinAFL. Among 15 applications, WinAFL could only run 6, whereas WINNIE was able run all 15. Columns marked "✗" indicate that the fuzzer could not fuzz the application. Markers "✔" indicate which heuristics were applied during harness generation. When both WinAFL and WINNIE support a program, WINNIE generally achieved better coverage and throughput. Although WINNIE excels at fuzzing complicated programs, WinAFL and WINNIE achieve similar results on small or simple programs. We explain in further detail in §VIII. For all other programs, WINNIE's improvement was statistically significant (i.e., p<0.05). P-values were calculated using the Mann-Whitney U test on discovered basic blocks.

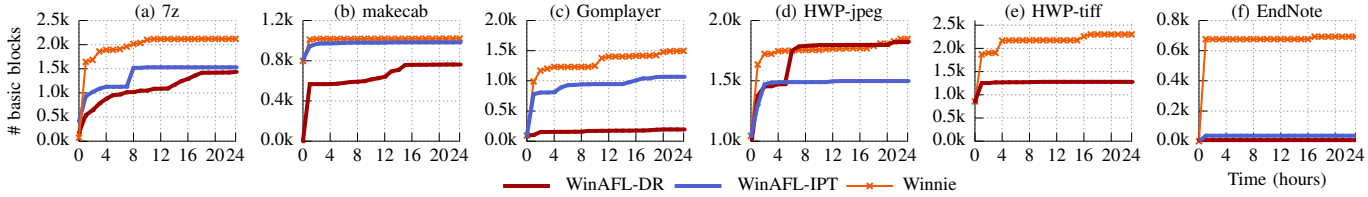

Fig. 9: Comparison of basic block coverage. We conducted five trials, each 24 hours long, with three fuzzers: WINNIE, WinAFL-DR, and WinAFL-IPT. Only programs which were supported by all fuzzers are shown here; WinAFL was unable to fuzz the rest. When a program can be fuzzed by both WINNIE and WinAFL, their performance is comparable. Nevertheless, most programs cannot be fuzzed with WinAFL.

average, the synthesized harnesses had 82.7 LoCs, relied on 3.2 heuristics, and required only 3.4% of the code to be modified. Based on our findings, we discuss the various strengths and weaknesses of the harness generator below.

Strengths of the Harness Generator. The execution tracer provides helpful information about the target program, such as promising fuzzing targets (i.e., Table IX: Target identification). This saves the user's time. While creating harnesses, we kept most the original code that WINNIE generated. Without the aid of our system, the user would have had to manually record all of the corresponding function calls and their arguments. The API sequences WINNIE generates also gives useful clues to the user. In the example harness for XnView, since WINNIE extracted 4 calls to the same API with differing arguments, one could conclude that the API's purpose was to initialize various attributes of an object. In our experiments, WINNIE successfully inferred some relationships present in the program (§IV-D). For example, WINNIE automatically detected that an opened file handle is passed to the next function (lines 6 and 10 in the example Figure 3) WINNIE also informs users about constant values, suggesting that they may be magic values that should not be modified.

To assess the usability of WINNIE and its ability to aid human researchers, we recruited two information security M.S. students who were unaware of the project. They were asked to use WINNIE to create fuzzing harnesses for Windows applications of their choice. Within 3 days, they were able to produce 7 functional harnesses, spending roughly only 3 hours per harness on average. The harness generator was most effective when it could rely on a single LCA API (e.g., Table VIII: 7z). In these cases, the user only needed to collect program run traces and provide them to the harness generator. Upon receiving the trace, WINNIE automatically calculated the LCA and generated C code to correctly invoke the function.

Weaknesses. Although most harnesses worked with few modifications, ACDSee and HWP-jpeg in particular required relatively large modifications (e.g., 34.3% and 16.3% respectively). This is mainly because they passed complex objects and virtual functions to the library's API. One challenge was reconstructing the custom structure layouts without the original source code. Although WINNIE dissects structures and pointer chains from the trace to provide plausible inferences, WINNIE is not perfect. To correct this, we analyzed the object using a decompiler and identified eight variables and four function pointers. Second, we manually extracted the callback functions by adding decompiled code. We followed the function pointers from the trace, and copied the decompiled code into the harness. There will always be some cases that WINNIE cannot handle. We discuss a few examples in §VIII, and we hope to support them in future versions of WINNIE.

D. Overall Results

1) Overall Testing Results: Figure 9 shows the ability of each fuzzer to find new coverage. Overall, WINNIE discovered 3.6× more basic blocks than WinAFL-DR and 4.1× more basic blocks than WinAFL-IPT. We also applied statistical tests,

Product	Buggy File	Size	Bug Type(s)	Bug(s)
Source Engine	**engine.dll**	6.1M	ND	2
MS WinDBG	**pdbcopy.exe**	743K	Arbitrary OOB read	1
MS Windows	**makecab.exe**	82K	Double free	1
Visual Studio	**ml.exe**	475K	SBOF	1
	undname.exe	23K	SOF	1
Alzip	**Egg.dll**	131K	ND	1
	Tar.dll	114K	Integer underflow	1
	Alz.dll	123K	Stack OOB read	1
Ultra ISO	**ultraISO.exe**	5.3M	Integer overflow, SOF	2
			Uninitialized use	1
XnView	**ldf_jpm.dll**	709K	HC, Integer overflow	2
Hancom Office	**HncBmp10.flt**	85K	Heap BOF	2
	HncJpg,Png,Gif	134-225K	ND	3
	HncDxf10.flt	242K	ND, Integer overflow	3
	HncTif10.flt	645K	HR, TC, FC, HC	6
	IMDRW9.flt	147K	ND, SBOF	2
	ISGDI32.flt	760K	Heap UAF, HC	3
	IBPCX9.flt	83K	Integer overflow, ND	2
FFMpeg	**FFmpeg.dll**†	12.8M	Div by zero	1
Uriparser	**uriparse.exe**†	157K	Integer underflow	1
Gomplayer	**RtParser.exe**	18K	SOF, SBOF, ND	3
EzPDF	**ezPDFEditor.exe**	23.9M	Race condition, ND	3
	Pdf2Office.dll	3.2M	SBOF, SOF, ND	3
VLC player	**Mediainfo.dll**	136K	Integer underflow	1
	libfaad.dll	273K	ND, Denial of service	2
Utable	**Utable.exe**	874K	SBOF	1
RetroArch	**bnes.dll**	2.4M	ND	2
	emux_gb.dll	419K	ND, Div by zero	3
	snes_9x.dll	2.8M	Heap OOB write	1
	quicknes.dll	1.0M	Div by zero	1
Capture2Text	**C2T_CLI.exe**	558K	ND	1
Total	32		19	61

ND: Null-ptr dereference, **HR:** Heap OOB read, **HC:** Heap corruption, **TC:** Type confusion, **FC:** Field confusion, **SOF:** Stack overflow, **SBOF:** Stack buffer overflow

TABLE X: Bugs found by WINNIE. We discovered total 61 unique vulnerabilities from 32 binaries. All vulnerabilities were discovered on the latest version of COTS binaries. We reported all bugs to the developers. "†" indicates that the bug existed in the released binary, but the developer had already fixed it when we filed our report.

using p-values to compare the performance of three fuzzers, as suggested by [37]. For WinAFL-DR and WinAFL-IPT, all trials except HWP-jpeg have p-values less than 0.05, meaning that WINNIE's improvement is statistically significant.

2) Real-world Vulnerabilities: WINNIE's approach scales to complex, real-world software. To highlight the effectiveness of our approach, we applied our system to non-trivial programs that are not just large in size but also accompany complicated logic and GUI code. We also included binaries from several well-known open-source projects because most of them have only been heavily fuzzed on Linux operating systems; thus their Windows-specific implementations may still contain bugs. Among them all, WINNIE found 61 previously unknown bugs in 32 binaries (shown in Table X). All these bugs are unique. These bugs cover 19 different types, including but not limited to stack and heap buffer overflow, type confusion, double free, uninitialized use, and null pointer dereference. At the time of writing, we have reported these bugs to their corresponding maintainers and are working with them to help fix the bugs.

VIII. DISCUSSION

Due to the difficulty of fuzzing closed-source, GUI-based applications, most Windows programs are tested either by unscalable manual efforts, or are only evaluated during the development by their vendors. In contrast, Linux programs are consistently tested and improved at all stages of the software lifecycle by researchers over the world. Most prior fuzzing work also has been concentrated on Linux systems. However, as shown in our evaluation, it is easy to find many bugs in Windows software we target—especially given the legacy code bases involved. Nevertheless, we identify several limitations of WINNIE, which can be addressed in the future to better test more programs.

Limitations of Harness-Based Testing. Testing the program with a harness limits the coverage within the selected features. In the case of WINNIE, we cannot reach any code in unforeseen features absent from the trace. Thus, the maximum code coverage possible is limited to the API set the trace covers; the number of generated harness is limited by the number of inputs traced. To mitigate this issue, we recommend users to collect as many sample inputs as possible to generate a diverse set of harnesses. Although we cannot eliminate this problem inherited from harness-based testing, automatic harness generation will help alleviate the burden of manually creating many harnesses.

Highly-Coupled Programs. It is more challenging for WINNIE to generate harnesses for applications tightly coupled with their libraries. As the logic is split into two binaries, the program may use frequent cross-module calls to communicate, making it hard to accurately identify and extract the relevant code we wish to fuzz. In Adobe Reader, for instance, the main executable `AcroRd32.exe` is simply a thin wrapper of the library AcroRd32.dll [3]. There are a lot of functions calls between these two binaries, or with other libraries, like `jp2.dll`. Thus, the harness generator needs to handle calls between the main executable and a library, callbacks from a library to the main executable, and calls between libraries. Our system focuses on handling cases where the communication merely happens within two components. To support more complicated invocations like in Adobe Reader, we plan to improve our tracer and generator to capture a complete trace of inter-module control- and data-flow.

False Positives. Inaccurate harnesses may generate invalid crashes or exceptions that do not occur in the original program. As a result, WINNIE will mistakenly assume the presence of a bug, leading to a false positive. As described in §IV-E, WINNIE combats false positives by pre-verifying candidate harnesses during synthesis. Still, eliminating false positives requires a non-negligible effort. Since bug validation must be conducted against the actual application, constructing a suitable input file and interacting with the GUI is required. For example, when fuzzing Adobe Reader's image parser, end-to-end verification requires creating a new PDF with the buggy image embedded, and then opening the image via the GUI. This step can be automated on a per-target basis, and it is mostly an engineering effort. Nevertheless, as long as WINNIE can generate high-quality harnesses, this validation incurs little overhead due to the small number of false crashes.

Focus on Shared Libraries. WINNIE's harness generator focuses testing shared libraries because shared libraries represent a clear API boundary. Past harness generation work also focuses on testing functions within libraries [8, 35]. Moreover, unlike calls to exported functions in libraries, private functions in the main executable are difficult to extract into independent functions. To fuzz the main binary, we rely on our injected fork-server, allowing any target address in the main binary to be fuzzed.

Performance Versus Persistent Mode. We noticed that WinAFL occasionally shows better performance on certain target applications, typically simple ones. Upon investigation, we found that the performance difference ultimately stems from WinAFL's strong assumptions about the target application. Specifically, WinAFL assumes the harness will not change any global state and will cleanly return back to the caller (§III-B). Therefore, it only restores CPU registers and arguments each loop iteration. Instead, WINNIE uses `fork` to comprehensively preserve the entire initialized program state, which incurs a little overhead. However, as shown in the evaluation, our conservative design makes WINNIE support significantly more programs. Although WinAFL performs better on simple programs, it could not test even half of the programs in our evaluation (§VII-A).

Other input modes. In our evaluation, we focused on fuzzing libraries which accept inputs from files or standard input. Another common way programs accept input is through network packets. WINNIE supports this case. To fuzz these network applications, we extended WINNIE by implementing a *de-socket* [13, 71] technique to redirect socket traffic to the fuzzer.

A. Future Work

Beyond this initial work towards practical Windows fuzzing, we identify several directions for future improvement. Among the following, we believe that handling structures and callback functions is fundamentally challenging, whereas supporting other ABIs or languages would be relatively straightforward.

Structures. Custom structures are challenging to both automatic testing tools and human researchers, and incorrect structures may lead to program crashes. To mitigate this issue, we could apply a memory pre-planning technique [66] to provide probabilistic guarantees to avoid crashes. We could also use memory breakpoints to trace the detailed memory access patterns of the program and infer the structure layouts.

Callback functions. Callback functions in the main executable make harness generation difficult. In our example Figure 3, we reconstructed the callback function by copying decompiled code from the main binary into the harness. For simple callbacks, we could automatically add decompiled code to the harness. For complicated cases, we could load the main binary and call the functions directly, as copied code is not always reliable.

Support for Non-C ABIs. WINNIE focuses on C-style APIs, and we did not investigate fuzzing programs with other ABIs. In our experience during the evaluation, these libraries are rare in practice. In the future, WINNIE can be extended to support other native languages' ABIs, like C++, Rust, or Go.

Bytecode languages and interpreted binaries. While WINNIE supports most native applications, it does not support applications compiled for a virtual machine (e.g., .NET, Java). To support these binaries, specialized instrumentation techniques [1] should be used to collect code coverage.

IX. Related Work

WINNIE is closely related to recent work on fuzzing and fuzzing harness generation. Fuzzing has evolved into a well-known program testing and bug finding technique since it was first introduced [42]. Various fuzzing techniques have been proposed [22, 24, 33, 36, 53, 60, 67], developed [5, 11, 14, 16, 25, 26, 38, 39, 50, 69], and used to find a large number of program bugs [12, 28, 52, 62, 69].

Windows Fuzzing. Although Windows fuzzing is restricted by the many challenges discussed in this paper, there are still many Windows fuzzer implementations. Black-box fuzzers like Peach [16] excel at scalability due to their simplicity but can only find shallow bugs. White-box fuzzers like Sage [24], leverage symbolic execution to explore deeper paths but are slow. Lastly, grey-box fuzzers like WinAFL [57], strike a good balance by using coverage feedback. However, existing WinAFL implementations suffer from unreliable persistent mode that limits the applications it can support (§III-B). WINNIE is based on WinAFL but uses a fork-server during execution, which is far more robust. As a result, WINNIE can fuzz far more programs than existing Windows fuzzers, as shown in §VII-A.

Fuzzing Harness Generation. Analyzing API usage patterns to generate code snippets is not a new idea. Some tools are primarily designed to help users understand an unfamiliar library [45, 46, 72]. IMF [32] analyzes kernel API call sequences recorded from the run trace to deduce API usage patterns and uses that knowledge during fuzzing (e.g., the order APIs should be called). Unlike past works which focus on static analysis of source code [8, 19, 35, 73] or dynamic analysis [32], WINNIE leverages hybrid analysis of run traces to extract code sequences. We further discuss WINNIE's differences in §II-B.

Partial Execution. There are several approaches to run code fragments to discover bugs [23, 51]. MicroX and UC-KLEE aim to run the code fragment under emulation and symbolic execution respectively. Unlike prior works, WINNIE aims to avoid heavy solutions that partially execute the program, such as emulation (like MicroX), or symbolic execution (like UC-KLEE). Also, WINNIE executes the target application under a realistic context. For any crash, there is a concrete input that helps reproduce the bug. UC-KLEE and MicroX execute it in the middle and thus may trigger an unrealistic execution path.

X. Conclusion

We proposed WINNIE, an end-to-end system to support fuzzing Windows applications. Instead of repeatedly running the program directly, WINNIE synthesizes lightweight harnesses to directly invoke interesting functions, bypassing GUI code. It also features an implementation of `fork` on Windows to clone processes efficiently. We tested WINNIE on 59 Windows closed-source binaries. WINNIE discovered 3.9× more program states and achieved 26.6× faster execution than existing Windows fuzzers. Overall, WINNIE found 61 bugs from 32 binaries.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Zhiyun Qian, for their helpful feedback. We are grateful to Weihan Huang for his initial implementation of the tracer for earlier versions of this paper. This research was supported, in part, by the NSF awards CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711, ONR under grants N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE under Agreement No. HR00112090034, and ETRI IITP/KEIT [2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

REFERENCES

[1] 0xd4d, ".NET module/assembly reader/writer library," https://github.com/0xd4d/dnlib, 2013.

[2] N. S. Agency, "Ghidra Software Reverse Engineering Framework," https://ghidra-sre.org/, 2019.

[3] Y. Alon and N. Ben-Simon, "50 CVEs In 50 Days: Fuzzing Adobe Reader," https://research.checkpoint.com/50-adobe-cves-in-50-days/, 2018.

[4] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "IJON: Exploring Deep State Spaces via Fuzzing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[5] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[6] C. authors, "Highlights of Cygwin Functionality," https://cygwin.com/cygwin-ug-net/highlights.html, 1996.

[7] AutoIt Consulting Ltd, "AutoIt Scripting Language," https://www.autoitscript.com/site/autoit/, 2019.

[8] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "FUDGE: Fuzz Driver Generation At Scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 975–985.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing As Markov Chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[10] Brian Krebs, "The Scrap Value of a Hacked PC," https://krebsonsecurity.com/2012/10/the-scrap-value-of-a-hacked-pc-revisited/, 2012.

[11] CENSUS, "Choronzon - An Evolutionary Knowledge-based Fuzzer," 2015, zeroNights Conference.

[12] O. Chang, A. Arya, and J. Armour, "OSS-Fuzz: Five Months Later, And Rewarding Projects," 2018, https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[13] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[14] P. Chen and H. Chen, "Angora: Efficient Fuzzing By Principled Search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[15] Dmytro Oleksiuk, "fork() for Windows," https://gist.github.com/Cr4sh/126d844c28a7fbfd25c6, 2016.

[16] M. Eddington, "Peach Fuzzing Platform," *Peach Fuzzer*, p. 34, 2011.

[17] Emma Woollacott, "Windows Of Opportunity: Microsoft OS Remains The Most Lucrative Target For Hackers," https://portswigger.net/daily-swig/windows-of-opportunity-microsoft-os-remains-the-most-lucrative-target-for-hackers, 2018, The Daily Swig.

[18] A. Fiscutean, "Microsoft Office Now The Most Targeted Platform, As Browser Security Improves," https://www.csoonline.com/article/3390221/microsoft-office-now-the-most-targeted-platform-as-browser-security-improves.html, 2019, cSO.

[19] J. Fowkes and C. Sutton, "Parameter-free Probabilistic API Mining Across GitHub," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 254–265.

[20] R. Freingruber, "Fuzzing Closed Source Applications," https://def.camp/wp-content/uploads/dc2017/Day1_Rene_Fuzzing_closed_source_applications_DefCamp.pdf, 2017.

[21] R. Freingruber, "Hack The Hacker: Fuzzing Mimikatz On Windows With Winafl & Heatmaps," https://sec-consult.com/en/blog/2017/09/hack-the-hacker-fuzzing-mimikatz-on-windows-with-winafl-heatmaps-0day/, 2017.

[22] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[23] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 539–549.

[24] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[25] Google, "Honggfuzz," 2016, https://google.github.io/honggfuzz/.

[26] Google, "Syzkaller - Linux Syscall Fuzzer," 2016, https://github.com/google/syzkaller.

[27] Google, "A New Chapter For OSS-Fuzz," https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html, 2018.

[28] Google, "Honggfuzz Found Bugs," 2018, https://github.com/google/honggfuzz#trophies.

[29] Google, "OSS-Fuzz - Continuous Fuzzing For Open Source Software," https://github.com/google/oss-fuzz, 2018.

[30] H. Gray, "Fuzzing Linux GUI/GTK Programs With American Fuzzy Lop (AFL) For Fun And Pr... You Get the Idea." https://blog.hyperiongray.com/fuzzing-gtk-programs-with-american-fuzzy-lop-afl/.

[31] I. Guilfanov, "IDA Pro - Hex Rays," https://www.hex-rays.com/products/ida/, 2018.

[32] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[33] C. Holler, K. Herzig, and A. Zeller, "Fuzzing With Code Fragments." in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[34] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[35] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic Fuzzer Generation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2020.

[36] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical Concolic Testing Techniques For COTS Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[37] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[38] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state Based Binary Fuzzing," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[39] LLVM, "LibFuzzer - A Library For Coverage-guided Fuzz Testing," 2017, http://llvm.org/docs/LibFuzzer.html.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40. ACM, 2005, pp. 190–200.

[41] Microsoft, "Frequently Asked Questions about Windows Subsystem for Linux," https://docs.microsoft.com/en-us/windows/wsl/faq, 2018.

[42] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study Of The Reliability Of UNIX Utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[43] M. Miller, "Trends, Challenges, And Strategic Shifts In The Software Vulnerability Mitigation Landscape," https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends,Challenges,andStrategicShiftsintheSoftwareVulnerabilityMitigationLandscape.pdf, 2019, BlueHat IL.

[44] J. Min, "Using WinAFL To Fuzz Hangul(HWP) AppShield," https://sigpwn.io/blog/2018/1/29/using-winafl-to-fuzz-hangul-appshield, 2018.

[45] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting Apis With Examples: Lessons Learned With The Apiminer Platform," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 401–408.

[46] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How Can I Use This Method?" in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 880–890.

[47] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead Through Coverage-guided Tracing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[48] D. Palmer, "Top Ten Security Vulnerabilities Most Exploited By Hackers," https://www.zdnet.com/article/these-are-the-top-ten-security-vulnerabilities-most-exploited-by-hackers-to-conduct-cyber-attacks/, 2019, zDNet.

[49] PaX Team, "PaX Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[50] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing By Program Transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[51] D. A. Ramos and D. Engler, "Under-constrained symbolic execution:

Correctness checking for real code," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.
[52] M. Rash, "A Collection Of Vulnerabilities Discovered By The AFL Fuzzer," 2017, https://github.com/mrash/afl-cve.
[53] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.
[54] M. Russinovich and D. A. Solomon, *Windows internals: including Windows server 2008 and Windows Vista*. Microsoft press, 2009.
[55] R. Schaefer, "Fuzzing Adobe Reader For Exploitable Vulns," https://kciredor.com/fuzzing-adobe-reader-for-exploitable-vulns-fun-not-profit.html, 2018.
[56] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.
[57] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, "AFL For Fuzzing Windows Binaries," 2016, https://github.com/ivanfratric/winafl.
[58] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, "How to Select A Target Function," https://github.com/googleprojectzero/winafl#how-to-select-a-target-function, 2016.
[59] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, "WinAFL Intel PT mode ," 2019, https://github.com/googleprojectzero/winafl/blob/master/readme_pt.md.
[60] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.
[61] symeon, "Fuzzing The MSXML6 Library With WinAFL," https://symeonp.github.io/2017/09/17/fuzzing-winafl.html, 2017.
[62] Syzkaller, "Syzkaller Found Bugs - Linux Kernel," 2018, https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md.
[63] XnSoft, "Supported file formats in XnView," https://www.xnview.com/en/xnviewmp/#formats, 2019.
[64] XnSoft, "XnView Image Viewer," https://www.xnview.com/en/, 2020.
[65] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.
[66] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "Pmp: Cost-effective forced execution with probabilistic memory pre-planning," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.
[67] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.
[68] M. Zalewski, "New In AFL: Persistent Mode," 2015, https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.
[69] M. Zalewski, "American Fuzzy Lop (2.52b)," 2018, http://lcamtuf.coredump.cx/afl/.
[70] M. Zalewski, "Fuzzing Random Programs Without Execve()," https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2019.
[71] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, USA, Aug. 2019.
[72] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining And Recommending API Usage Patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.
[73] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining Api Usage Examples From Test Code," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 301–310.

APPENDIX

A. Fork Internals

To implement our `fork` functionality, we reverse-engineered various internal Windows APIs and systems, namely `ntdll.dll`, `NtCreateUserProcess`, and the CSRSS (*Client/Server Runtime Subsystem*). Doing so, we identified several magic values required by them. After overcoming these challenges, we were able to implement a practical, robust fork-server for fuzzing.

> **Role of CSRSS.** The CSRSS is the user-mode process that controls the underlying layer of the Windows environment [54]. This daemon is responsible for allocating console windows and shutting down processes. New processes must connect to it to function properly.

We use Windows native system APIs to communicate with the kernel directly. Figure 6 and Figure 10 display our fork implementation. The steps are as follows:

❶ The parent process calls `NtCreateUserProcess` with the proper flags, creating a suspended child process with a CoW copy of the parent's address space (line 1). We keep the child process suspended until the parent ❷ calls `CsrClientCallServer` to inform the CSRSS that a new process was created (line 12). ❸ The parent now resumes the child, which proceeds to self-initialize (line 17). Then, the parent returns from `fork` (line 14). ❹ In the child process, because the address space matches the parent's, several global variables (e.g., `CsrServerApiRoutine` in `ntdll.dll`) that would be zero for new processes are already set. The child must de-initialize these manually by zeroing them out (line 18) to avoid crashing in the next step. ❺ The child now connects to the CSRSS by calling `CsrClientConnectToServer` (line 20). This step is critical for the child process to function properly. ❻ The CSRSS finally acknowledges the newly created process and thread, and the child process returns from `fork` (line 21).

```
1  NTSTATUS result = NtCreateUserProcess(
2      &hProcess, &hThread, MAXIMUM_ALLOWED, MAXIMUM_ALLOWED,
3      NULL, NULL, PROCESS_CREATE_FLAGS_INHERIT_FROM_PARENT
4      | PROCESS_CREATE_FLAGS_INHERIT_HANDLES,
5      THREAD_CREATE_FLAGS_CREATE_SUSPENDED,
6      NULL, &procInfo, NULL
7  );
8
9  if (!result) { // Parent process
10     // Inform the CSRSS that a new process was created
11     // via CsrClientCallServer(CreateProcessRequest)
12     NotifyCsrssParent(hProcess, hThread);
13     // Allow the child to connect to CSR and resume.
14     ResumeThread(hThread);
15     return GetProcessId(hProcess);
16 } else { // Child process
17     // De-initialize ntdll variables before re-initialization
18     memset(pCsrData, 0, csrDataSize);
19     // Connect to the CSRSS via CsrClientConnectToServer
20     ConnectCsrChild();
21     return 0;
22 }
```

Fig. 10: Fork implementation, We displayed the core fork() function only. Low level details and helper functions are omitted for brevity. For more detailed code, refer to our project's source code.

B. Tested Harnesses

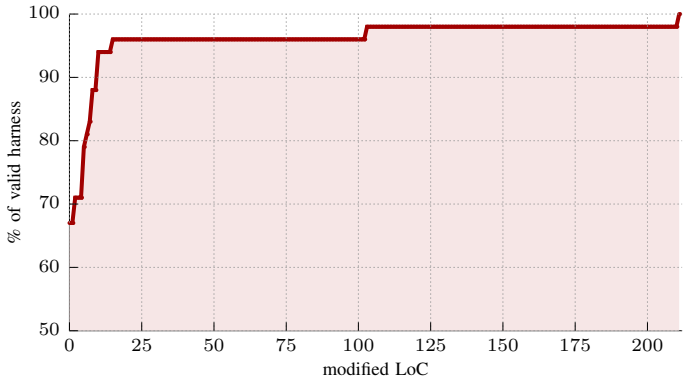

Fig. 11: Cumulative distribution plot for our harnesses. The graph plots how many of our harnesses fixed N LoC or fewer. As shown, nearly 70% of the harnesses worked without any modifications at all. 95% of our harnesses could be fuzzed with $\leq$ 10 LoC modifications.

#	Program	File	Size	WinAFL-PT	WinAFL-DR	#	Program	File	Size	WinAFL-PT	WinAFL-DR
1	**7zip**	7z.dll	1115KB	✔	✔	31	**IrfanView**	jpg_transform	404KB	timeout	timeout
2	**WinRAR**	rar.exe	557KB	✔	✔	32	**EzPDF**	pdf2html.dll	5596KB	crash	crash
3	**makecab**	makecab.exe	68KB	✔	✔	33		ezpdf2hwp.dll	1227KB	crash	crash
4	**GomPlayer**	avformat-gp-57	4092KB	✔	✔	34		pdf2office(xls)	3221KB	crash	crash
5	**expand**	expand.exe	53KB	✔	✔	35	**RetroArch**	bnes.dll	2407KB	timeout	timeout
6	**VLCPlayer**	libfaad.dll	273KB	✔	✔	36		handy.dll	736K	timeout	timeout
7	**uriparser**	uriparser.exe	157KB	✔	✔	37		quicknes.dll	1022KB	timeout	timeout
8	**AdobeRdr**	jp2klib.dll	847KB	✔	✔	38		bsnes.dll	31MB	timeout	timeout
9	**Starcraft**	storm.dll	453KB	✔	✔	39		fmsx.dll	523KB	timeout	timeout
10	**HWP**	impic9.flt	86KB	✔	✔	40		sfc.dll	2724KB	timeout	timeout
11		imcdr9.flt	70KB	✔	✔	41		vbam.dll	1683KB	timeout	timeout
12		hncbmp10.flt	85KB	✔	✔	42		fcemm.dll	683KB	timeout	timeout
13		hncgif10.flt	131KB	✔	✔	43		desmume.dll	5408KB	timeout	timeout
14		hncwmf10.flt	79KB	✔	✔	44	**ml**	ml.exe	476KB	crash	crash
15		imdrw9.flt	147KB	✔	✔	45	**mspdbcmf**	mspdbcmf.exe	1150KB	timeout	timeout
16		hncjpeg10.flt	220KB	✔	✔	46	**pdbcopy**	pdbcopy.exe	726KB	proc terminated	input file leak
17		hnctiff10.flt	629KB	✔	✔	47	**XnView**	cadimage.dll	4205KB	crash	crash
18	**monkey2**	mac.exe	408KB	✔	✔	48		ldf_jpm.dll	692KB	crash	crash
19	**GraphMagic**	core_rl_magic_	973KB	✔	✔	49	**UltraISO**	ultraiso.exe	5250KB	no inst.	crash
20	**undname**	undname.exe	23KB	✔	✔	50	**ACDSee**	IDE_ACDStd	3007KB	timeout	timeout
21	**EzPDF**	pdf2office(doc)	3221KB	✔	✔	51	**KGB**	paq6.dll	52KB	no inst.	keep terminating
22		pdf2office(ppt)	3221KB	✔	✔	52	**inkscape**	inkscape.exe	386KB	crash	crash
23	**Alzip**	egg.dll	131KB	✔	✔	53	**MuseScore3**	musescore3.exe	30.5MB	timeout	timeout
24		tar.dll	114KB	✔	✔	54	**MSSDK**	peverify.exe	257KB	timeout	timeout
25		alz.dll	123KB	✔	✔	55	**tar**	tar.exe	43KB	timeout	timeout
26	**Lib**	lib.exe	20KB	✔	✔	56	**link**	link.exe	1358KB	timeout	timeout
27	**HWP**	hncpng10.flt	479KB	input creation failed	✔	57	**esentutl**	esentutl.exe	341KB	hang	timeout
28		imgdrw9.flt	146KB	input creation failed	✔	58	**PowerISO**	macdll.dll	466KB	timeout	timeout
29	**Libmagic**	magic1.dll	147KB	no inst.	✔	59	**tiled**	tmxviewer.exe	109KB	timeout	timeout
30	**dxcap**	dxcap.exe	904KB	CreateProc fail	Prog failed.						

TABLE XI: Results of testing the generated harnesses with WinAFL. We generated 59 harnesses and tested on WINNIE, WinAFL-IPT, and WinAFL-DR. WINNIE was able to run all 59 harnesses. WinAFL-IPT and WinAFL-DR failed to run 33 and 30 harnesses respectively. "No inst." denotes no instrumentation.

Program	Target	Size	APIs	LoC	Fixed (LoC)	(%)	Program	Target	Size	APIs	LoC	Fixed (LoC)	(%)
AdobeRdr	jp2klib.dll	847KB	9	282	CB (89), ST (14)	36.5	**WinRAR**	rar.exe	557K	1	55	0	0.0
ACDSee	IDE_ACDStd	3007K	19	506	CB (38), ST (174)	34.3	**expand**	expand.exe	53K	1	55	0	0.0
HWP	HncJpeg10.dll	220K	3	92	CB (7), ST (8)	16.3	**VLCPlayer**	libfaad.dll	273K	1	55	0	0.0
	hncbmp10.flt	85KB	3	103	CB (2), ST (8)	9.7	**uriparser**	uriparser.dll	157K	1	55	0	0.0
ezPDF	Pdf2Office(d)	3221K	4	112	CB (2), ST (8)	8.9	**Monkey2**	mac.exe	408K	1	55	0	0.0
	Pdf2Office(p)	3221K	4	112	CB (2), ST (8)	8.9	**GraphMagic**	core_rl_magic	973K	1	55	0	0.0
	Pdf2Office(e)	3221K	4	112	CB (2), ST (8)	8.9	**undname**	undname.exe	23K	1	55	0	0.0
HWP	HncTiff10.dll	630K	3	82	CB (7)	8.5	**Alzip**	egg.dll	131K	1	55	0	0.0
ezPDF	Pdf2html.dll	5596K	4	110	ST (8)	7.2		tar.dll	114K	1	55	0	0.0
	ezpdf2hwp.dll	1227K	4	110	ST (8)	7.2		alz.dll	123K	1	55	0	0.0
HWP	imgpic9.flt	86KB	3	90	CB (2), ST (3)	5.6	**dxcap**	dxcap.exe	904K	1	55	0	0.0
	imdrw9.flt	147KB	3	90	CB (2), ST (3)	5.6	**RetroArch**	bnes.dll	2407K	1	55	0	0.0
	imcdr9.flt	70KB	3	92	CB (2), ST (3)	5.4		handy.dll	736K	1	55	0	0.0
	hncgif10.flt	131KB	3	92	CB (2), ST (3)	5.4		quicknes.dll	1022K	1	55	0	0.0
	hncwmf10.flt	79KB	3	92	CB (2), ST (3)	5.4		bsnes.dll	31MB	1	55	0	0.0
	hncpng10.flt	479KB	3	102	ST (8)	7.8		fmsx.dll	523KB	1	55	0	0.0
UltraIso	UltraISO.exe	5250K	1	57	CB (2)	3.5		sfc.dll	2724KB	1	55	0	0.0
XnView	cadimage.dll	4205	2	65	PTR (2)	3.1		vbam.dll	1683KB	1	55	0	0.0
	ldf_jpm.dll	692	10	199	CB (4), PTR (2)	3.0		fcemm.dll	683KB	1	55	0	0.0
EndNote	RMConvertLib	2027K	1	73	argument (2)	2.7		desmume.dll	5408KB	1	55	0	0.0
Gomplayer	avformat-gp.dll	4091K	7	116	PTR (2)	1.7	**Inkscape**	inkscape.exe	386KB	1	55	0	0.0
EndNote	PC4DbLib	2738K	1	55	0	0.0	**MuseScore3**	musescore3	30.5KB	1	55	0	0.0
file	magic1.dll	147K	3	96	0	0.0	**MSSDK**	peverify.exe	257KB	1	55	0	0.0
Starcraft	storm.dll	453KB	2	37	0	0.0	**tar**	tar.exe	43KB	1	55	0	0.0
7z	7z.exe	1114K	1	55	0	0.0	**link**	link.exe	1358KB	1	55	0	0.0
makecab	makecab.exe	50K	1	55	0	0.0	**lib**	lib.exe	20KB	1	55	0	0.0
Tiled	tmxviewer.exe	113K	1	55	0	0.0	**esentutl**	esentutl.exe	341KB	1	55	0	0.0
mspdbcmf	mspdbcmf.exe	1149K	1	55	0	0.0	**PowerISO**	macdll.dll	466KB	1	55	0	0.0
pdbcopy	pdbcopy.exe	726K	1	55	0	0.0	**KGB**	paq6.dll	52KB	1	55	0	0.0
ml	ml.exe	476K	1	55	0	0.0							

CB: Callback function, **ST:** Custom struct, **PTR:** Pointer

TABLE XII: All harnesses included in our evaluation. Of 59 harnesses, the majority worked without any modifications required, and only 3 required changing more than 10 lines of code. We discuss the problem of callbacks and structs in §VIII-A. The problematic Adobe Reader example is discussed in §VIII. The ACDSee and HWP-jpeg harnesses are discussed in §VII-C.