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.

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., \texttt{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 $1.5 \times -2 \times$ faster on Linux [70]. \texttt{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 \texttt{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., \textit{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, \textsc{WINNIE}, to address the aforementioned challenges and make fuzzing Windows programs more practical. \textsc{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 \texttt{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 \texttt{fork} on Windows systems for fuzzing.

We implemented \textsc{WINNIE} in 7.1K lines of code (LoC). We applied \textsc{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 $26.6 \times$ faster execution and discovered $3.6 \times$ more basic blocks than WinAFL, the state-of-the-art fuzzer on Windows. By fuzzing these 59 harnesses, \textsc{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 \textsc{WINNIE}, which can automatically generate harnesses for Windows binaries to bypass GUI code;
- We implemented the first efficient Windows fork-server;
- \textsc{WINNIE} successfully generated efficient harnesses for 59 Windows binaries and found 61 bugs in 32 binaries.

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

![Fig. 2: Fuzzing overview.](image)

\textbf{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 \texttt{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 \textit{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. \textit{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. \textit{Table I} compares Linux AFL with popular Windows fuzzers. WinAFL is a fork of AFL ported for Windows systems [57] and supports feedback-
driven fuzzing. HonggFuzz supports Windows, but only for fuzzing binaries in dumb mode, i.e., without any coverage feedback [25]. Peach is another popular fuzzier 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.

**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.

**TABLE II: Execution times (ms) with and without GUI.** GUI code often intertwine the asynchronous GUI code with the input sequence 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 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.

**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.
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 inside the ldjpm.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.

1 // 1) Declare structures and callbacks
2 int callback1(void* a1, int a2) { ... }
3 int callback2(void* a1, int a2) { ... }
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(sizeof(int), num);
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], 0xA, &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 ldjpm.dll library of the application XnView. 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.

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.

3 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.

4 Control-flow and data-flow dependence. It is often 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
We propose \textit{Winnie}, an end-to-end system that addresses aforementioned obstacles to effectively and efficiently fuzz Windows applications. \textit{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 \texttt{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 \texttt{fork-server} to find bugs.

### C. Our Solutions

We propose \textit{Winnie}, an end-to-end system that addresses aforementioned obstacles to effectively and efficiently fuzz Windows applications. \textit{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 \texttt{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 \texttt{fork-server} to find bugs.

### IV. Harness Generation

To generate the harness, \textit{Winnie} followed the four steps previously outlined in §III-A. Consider XnView as an example:

1. For \textit{target discovery} (§IV-A), we trace XnView while opening several JPM files, and then search the traces for input-related APIs, such as \texttt{OpenFile} and \texttt{ReadFile}.
2. For \textit{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.
3. For \textit{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.
4. Finally, we establish the \textit{relationships} (§IV-D) among the various calls and variables presented in the harness skeleton and emit the final code after briefly \textit{testing} (§IV-E) the candidate harness. \textit{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, \textit{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.  

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
<th>What to record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>string</td>
<td>name, path, module</td>
</tr>
<tr>
<td>Call/Jump</td>
<td>inter-module</td>
<td>thread id, caller, callee, symbols, args</td>
</tr>
<tr>
<td>Return</td>
<td>intra-module</td>
<td>same as above, only for main .exe</td>
</tr>
<tr>
<td>Argument</td>
<td>intra-module</td>
<td>thread id, callee, caller, retval</td>
</tr>
<tr>
<td>RetVal</td>
<td>constants</td>
<td>concrete value</td>
</tr>
<tr>
<td></td>
<td>pointers</td>
<td>address and referenced data (recursively)</td>
</tr>
</tbody>
</table>

Fig. 4: Overview of \textit{Winnie}. Given the target program and a set of sample inputs, \textit{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 \texttt{fork}.

overcome the re-execution overhead [68]. In persistent mode, the fuzzer repeatedly invokes the target function in a tight loop \textit{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 \textit{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, unrepeatable 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.

#### 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).

sections, we will use the harness shown in Figure 3 as an example to explain the design of each component of \textit{Winnie}.
call the APIs. To find functions taking file descriptors as input, we identify callers of well-known file-related APIs such as ReadFile.

This is because the main executable is responsible for calling the new content. Second, the LCA should reach locations that opened the input file, we still can modify it so the program uses the file.

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 as 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 ACDSSee. The function at address 0x5ce80 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.

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 unrepeatable 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., $f_0.a_0$ in Figure 3). We consider the following possible cases:

- **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 $f_0.a_0$ 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 functions return object pointers. 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
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.

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.

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 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.

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.
Then, the fuzzer creates the agent reports program’s status and
The fuzzing agent, which

TerminateProc
Fuzzing target
NtCreateFile,
Entry Point

The injected agent
int 3
instruction. Then, we fuzz the patched program and wait
new coverage when a new basic block is visited. To implement
block coverage, fullspeed fuzzing only considers there to be
the fuzzer discovers a new basic block. Based on boolean basic
blocks, and hooks several functions. This improves performance
fullspeed fuzzing does not introduce any overhead except when
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 applica-
bigility 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 \( \times 24 \text{hrs} \times 5 \text{trials} = 5,400 \text{CPU hrs} \)). 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
Table VII: Evaluation of fork()

<table>
<thead>
<tr>
<th>Program</th>
<th>Without Fork Leak</th>
<th>Hang⁴</th>
<th>Fork Speed</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7z</td>
<td>5.2</td>
<td>1430</td>
<td>49.9</td>
<td>2171</td>
</tr>
<tr>
<td>makecab</td>
<td>18.4</td>
<td>576</td>
<td>49.4</td>
<td>1020</td>
</tr>
<tr>
<td>GomPlayer</td>
<td>0.4</td>
<td>201</td>
<td>25.9</td>
<td>1496</td>
</tr>
<tr>
<td>Hwp-jpeg</td>
<td>4.2</td>
<td>1045</td>
<td>25.9</td>
<td>1847</td>
</tr>
<tr>
<td>Hwp-tiff</td>
<td>0.3</td>
<td>1340</td>
<td>26.2</td>
<td>2301</td>
</tr>
<tr>
<td>EndNote</td>
<td>5.3</td>
<td>68</td>
<td>89.5</td>
<td>693</td>
</tr>
<tr>
<td>Total</td>
<td>3/6</td>
<td>2/6</td>
<td>(31.3×)</td>
<td>(4.0×)</td>
</tr>
</tbody>
</table>

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.

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
10 in the example Figure 3) WinAFL. Only programs which were supported by all fuzzers are shown here; WinAFL was unable to fuzz the rest. When a program can
weaknesses of the harness generator below. For example, W
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.

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 of 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. 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, ACDSsee 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. While 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,
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.

<table>
<thead>
<tr>
<th>Product</th>
<th>Buggy File</th>
<th>Size</th>
<th>Bug Type(s)</th>
<th>Bug(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Engine</td>
<td>engine.dll</td>
<td>6.1M</td>
<td>ND</td>
<td>2</td>
</tr>
<tr>
<td>MS WinDBG</td>
<td>pdbcopy.exe</td>
<td>743K</td>
<td>Arbitrary OOB read</td>
<td>1</td>
</tr>
<tr>
<td>MS Windows</td>
<td>makecache.exe</td>
<td>82K</td>
<td>Double free</td>
<td>1</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>ml.exe</td>
<td>475K</td>
<td>SBOF</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>unname.exe</td>
<td>23K</td>
<td>SOF</td>
<td>1</td>
</tr>
<tr>
<td>Alzip</td>
<td>Egg.dll</td>
<td>131K</td>
<td>ND</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Tar.dll</td>
<td>114K</td>
<td>Integer underflow</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Alz.dll</td>
<td>123K</td>
<td>Stack OOB read</td>
<td>1</td>
</tr>
<tr>
<td>Ultra ISO</td>
<td>ultraISO.exe</td>
<td>5.3M</td>
<td>Integer overflow, SOF</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Uninitialized use</td>
<td>1</td>
</tr>
<tr>
<td>XnView</td>
<td>ldf_jmp.dll</td>
<td>709K</td>
<td>HC, Integer overflow</td>
<td>2</td>
</tr>
<tr>
<td>Hancorn Office</td>
<td>HncBmp10.fft</td>
<td>85K</td>
<td>Heap BOF</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>HncJpg_Png_Gif</td>
<td>132-225K</td>
<td>ND</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>HncDxf10.fft</td>
<td>242K</td>
<td>ND, Integer overflow</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>HncTif10.fft</td>
<td>645K</td>
<td>HR, TC, FC, HC</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>IMDRV9.fft</td>
<td>147K</td>
<td>ND, SBOF</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>ISGDI12.fft</td>
<td>760K</td>
<td>Heap UAE, HC</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>IBPCX9.fft</td>
<td>83K</td>
<td>Integer overflow, ND</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>FFMpeg.dll⁺</td>
<td>12.8M</td>
<td>Div by zero</td>
<td>1</td>
</tr>
<tr>
<td>Uriparse</td>
<td>uriparse.exe</td>
<td>157K</td>
<td>Integer underflow</td>
<td>1</td>
</tr>
<tr>
<td>Compiler</td>
<td>RTParser.exe</td>
<td>18K</td>
<td>SOF, SBOF, ND</td>
<td>3</td>
</tr>
<tr>
<td>EzPDF</td>
<td>ezPDFEditon.exe</td>
<td>23.9M</td>
<td>Race condition, ND</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Pdf2Office.dll</td>
<td>3.2M</td>
<td>SBOF, SOF, ND</td>
<td>3</td>
</tr>
<tr>
<td>VLC player</td>
<td>Mediainfo.dll</td>
<td>136K</td>
<td>Integer underflow</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>libpad.dll</td>
<td>273K</td>
<td>ND, Denial of service</td>
<td>2</td>
</tr>
<tr>
<td>Utable</td>
<td>Utable.exe</td>
<td>874K</td>
<td>SBOF</td>
<td>1</td>
</tr>
<tr>
<td>RetroArch</td>
<td>bmes.dll</td>
<td>2.4M</td>
<td>ND</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>emux_g.dll</td>
<td>419K</td>
<td>ND, Div by zero</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>snes_9x.dll</td>
<td>2.8M</td>
<td>Heap OOB write</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>quicknss.dll</td>
<td>1.0M</td>
<td>Div by zero</td>
<td>1</td>
</tr>
<tr>
<td>Capture2Text</td>
<td>C2T_CLL.exe</td>
<td>558K</td>
<td>ND</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>32</td>
<td>19</td>
<td>61</td>
<td></td>
</tr>
</tbody>
</table>

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

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.d11. 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 desocket [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.


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:

1. 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 2 calls CsrClientCallServer to inform the CSRSS that a new process was created (line 12). The parent now resumes the child, which proceeds to self-initialize (line 17). Then, the parent returns from fork (line 14).

2. 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.

3. The child now connects to the CSRSS by calling CsrClientConnectToServer (line 20). This step is critical for the child process to function properly.

4. The CSRSS finally acknowledges the newly created process and thread, and the child process returns from fork (line 21).

5. If (!result) {  // Parent process
   // Inform the CSRSS that a new process was created
   // via CsrClientCallServer(CreateProcessRequest)
   NotifyCsrssParent(hProcess, hThread);
   // Allow the child to connect to CSR and resume.
   ResumeThread(hThread);
   return GetProcessId(hProcess);
}

6. else {  // Child process
   // De-initialize ntdll variables before re-initialization
   memset(pCsrData, 0, csrDataSize);
   // Connect to the CSRSS via CsrClientConnectToServer
   ConnectCsrChild();
   return 0;
}

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 ≤ 10 LoC modifications.
<table>
<thead>
<tr>
<th>#</th>
<th>Program</th>
<th>File</th>
<th>Size</th>
<th>WinAFL-PT</th>
<th>WinAFL-DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>zip</td>
<td>7z.dll</td>
<td>1115KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>WinRAR</td>
<td>rar.exe</td>
<td>597KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>makecab</td>
<td>makecab.exe</td>
<td>68KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>GomPlayer</td>
<td>avformat-gp-57</td>
<td>4902KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>expand</td>
<td>expand.exe</td>
<td>53KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VLCPlayer</td>
<td>libfaad.dll</td>
<td>273KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>uriparser</td>
<td>uriparser.exe</td>
<td>157KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AdobeRdr</td>
<td>ppklib.dll</td>
<td>847KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Starcraft</td>
<td>storm.dll</td>
<td>453KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>HWMP</td>
<td>impsi9.exe</td>
<td>86KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>imcd9.exe</td>
<td>70KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>hncbmp10.lib</td>
<td>85KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>hncgif10.lib</td>
<td>131KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>hncwmf10.lib</td>
<td>79KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>imcd9.exe</td>
<td>147KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>hncjpeg10.lib</td>
<td>220KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>hntcfl10.lib</td>
<td>629KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>monkey2</td>
<td>mac.exe</td>
<td>408KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>raw.dll</td>
<td>973KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>undname</td>
<td>undname.exe</td>
<td>23KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>EzPDF</td>
<td>pdf2office(doc)</td>
<td>3221KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Alzip</td>
<td>egg.dll</td>
<td>131KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>tar</td>
<td>tar.dll</td>
<td>114KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Lib</td>
<td>lib.exe</td>
<td>208KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>HWP</td>
<td>hncpng10.lib</td>
<td>479KB</td>
<td>no inst.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>imgrdw9.lib</td>
<td>146KB</td>
<td>input creation failed</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Libmagic</td>
<td>magic.l.dll</td>
<td>147KB</td>
<td>no inst.</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>dxcap.exe</td>
<td>904KB</td>
<td>CreateProc fail</td>
<td>Prog failed</td>
</tr>
</tbody>
</table>

**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.