No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting

Leo Stone
Virginia Tech
leo@vt.edu

Rishi Ranjan
Virginia Tech
rishiranjan@vt.edu

Stefan Nagy
University of Utah
snagy@cs.utah.edu

Matthew Hicks
Virginia Tech
mdhicks2@vt.edu

***Abstract*—Coverage-guided fuzzing remains today's most successful approach for exposing software security vulnerabilities. Speed is paramount in fuzzing, as maintaining a high test case throughput enables more expeditious exploration of programs—leading to faster vulnerability discovery. High-performance fuzzers exploit the Linux kernel's customizability to implement process *snapshotting*: fuzzing-oriented execution primitives that dramatically increase fuzzing throughput. Unfortunately, such speeds remain elusive on Windows. The closed-source nature of its kernel prevents current kernel-based snapshotting techniques from being ported—severely limiting fuzzing's effectiveness on Windows programs. Thus, accelerating vetting of the Windows software ecosystem demands a *fast*, *correct*, and *kernel-agnostic* fuzzing execution mechanism.**

We propose making state snapshotting an application-level concern as opposed to a kernel-level concern via *target-embedded snapshotting*. Target-embedded-snapshotting combines binary- and library-level hooking to allow applications to snapshot themselves—while leaving both their source code and the Windows kernel untouched. Our evaluation on 10 real-world Windows binaries shows that target-embedded snapshotting overcomes the speed, correctness, and compatibility challenges of previous Windows fuzzing execution mechanisms (i.e., process creation, forkserver-based cloning, and persistent mode). The result is 7–182x increased performance.

1. Introduction

With software exploits dominating today's cyber-threat landscape, developers and security practitioners are in a constant race against time to find and mitigate security bugs before attackers can exploit them to cause harm. Over the last decade, coverage-guided fuzzing has become the most popular and successful technique for automatically uncovering security vulnerabilities in software. At its core, fuzzing operates by ① generating massive amounts of random test cases, ② executing each on the target program and tracing their code coverage, and ③ triaging test cases by their observed behaviors (e.g., code coverage and crashes).

Many academic and industrial advancements have greatly improved fuzzing's test case generation [1], [2], [3] and triage [4], [5], [6] abilities. However, as fuzzing aims to scrutinize programs with a large volume of test cases, maintaining a high performance—and hence, a high test case throughput—is critical to effective fuzzing. As research shows that test case execution and code coverage tracing are fuzzing's most resource-intensive steps [7], many successful fuzzing optimizations act on the coverage-tracing level [8], [9], [10], [11]. The success in eliminating coverage tracing overhead leaves the overarching process execution mechanism (i.e., preparing a process for each test case) as the current limiting factor for fuzzing speed [12], [13], [14].

To facilitate effective software fuzzing, an execution mechanism must uphold two key properties:

1) **Efficiency:** avoiding re-execution of target code (e.g., GUI initialization) or kernel code (e.g., virtual memory management). This increases test case throughput by reducing time spent executing uninteresting code.
2) **Correctness:** ensuring execution is semantically correct no matter the number or order of previous test cases. This eliminates the possibility of false positives (e.g., spurious crashes) and false negatives (e.g., missed vulnerabilities).

In recent years, the Linux fuzzing community is abandoning slow-yet-correct process execution mechanisms (e.g., fork–exec and AFL's forkserver [15]) in favor of *fast-and-correct* process *snapshotting*: fuzzing-tailored execution primitives implemented via extensions to Linux's open-source kernel [16], [13], [17]. Snapshotting represents the fastest-possible correct process execution in fuzzing today [17], enabling orders-of-magnitude higher test case throughput for expedited software vulnerability discovery.

Unfortunately, the benefits of snapshot-accelerated execution have yet to appear in Windows software fuzzing. The Windows kernel's closed-source nature prevents third-party porting of Linux-based snapshot primitives, forcing Windows fuzzers to rely on slowest-of-all execution via process *creation* (e.g., `CreateProcess()` [18]). Recent reverse-engineering efforts expose hidden copy-on-write process *cloning* primitives [19], yet their version-specific nature leaves them incompatible with older—and newer—kernels [20]. Worse yet, for reasons undisclosed by Windows, existing process creation and cloning mechanisms are **up to 1,000× slower** than their Linux counterparts [21], [19]. Some fuzzers are embracing faster in-memory process *looping* (e.g., WinAFL's persistent mode [18]), yet

its inability to reset process states leads to semantically-incorrect execution, leading to both **spurious and missed crashes**. Without fast *and* correct execution supportive of *all* Windows kernels, Windows application fuzzing will remain impaired, impeding efforts to vet the software of one of the world's largest computing ecosystems.

To overcome these challenges, this paper introduces the concept of *target-embedded snapshotting*: a technique to make fuzzed Windows applications snapshot *themselves*. Our approach is guided by the insight that fuzzing-relevant program state (i.e., stack, globals, and heap) is controllable purely by language-level constructs—enabling the responsibility of snapshotting to be shifted from the kernel *to the fuzzed program* instead. At a high level, our technique injects snapshotting's state restoration inside of conventional in-memory looping, resetting program state on every program termination (i.e., when test cases finish executing). For stack state, we repurpose C++'s non-local jumping construct [22] to reset the instruction, frame, and base pointers; for global state, we copy and restore the program's data segment (using Windows guard pages [23] to increase efficiency); and for heap state, we introduce a lightweight bookkeeping mechanism to intercept dynamically-allocated memory chunks and free them on program exit.

We design a *binary-only* implementation of target-embedded snapshotting, WINFUZZ, and evaluate it against the leading execution mechanisms used in Windows fuzzing. Across 10 binary-only benchmarks of varying type, size, and structure, WINFUZZ outperforms WinAFL's process creation and Winnie's forkserver-based process cloning by covering **18x** and **6x** more test cases, respectively, and attaining **5%** and **15%** higher code coverage, respectively.

In summary, this paper contributes the following:

- We examine existing execution mechanisms' design trade-offs, developing a criteria of the ideal characteristics needed to support effective Windows software fuzzing.
- We leverage this criteria to design *target-embedded snapshotting*: a fast *and* correct process execution mechanism for fuzzing binary-level Windows applications.
- We show that it is possible to combine readily-available programming language constructs with lightweight binary instrumentation to enable programs to automatically restore their own stack, heap, and global state during fuzzing—without needing to modify the Windows kernel.
- We implement a prototype of target-embedded snapshotting, WINFUZZ, and evaluate it against current state-of-the-art execution mechanisms available to Windows software fuzzing. We show that WINFUZZ's target-embedded snapshotting facilitates fuzzing with higher test case throughput, code coverage, and crash discovery than the leading Windows fuzzing execution mechanisms.
- We open-source WINFUZZ, our implementation of target-embedded snapshotting, and our evaluation artifacts at: `github.com/FoRTE-Research/winfuzz`.

2. Background

In this section we introduce the topics central to target-embedded snapshotting: coverage-guided fuzzing, process execution state, and fuzzing process execution mechanisms.

2.1. Coverage-guided Fuzzing

Fuzzing is by far today's most popular and successful software testing technique for automatically discovering software bugs and security vulnerabilities. As shown in Figure 1, given a target program and an initial set of seed inputs for it, the process of fuzzing encompasses three high-level steps:

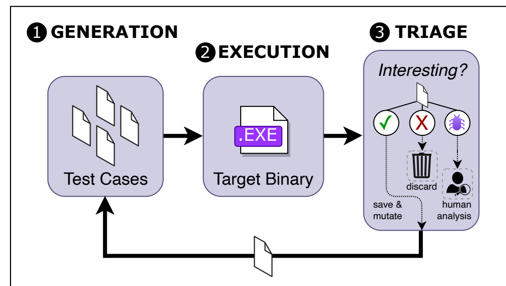

Figure 1: A visualization of software fuzzing's fundamental steps.

1) **Generation:** Batches of test cases are created by mutating saved test cases or starting seed inputs. Existing mutation techniques generally rely on injecting random bits or bytes [15], pre-defined [24], [2] or dynamically-learned [25], [3] input grammars, or token values extracted from the program [26].
2) **Execution:** The target program is run on each generated test case, with lightweight monitoring used to flag those that reveal unusual execution behavior (e.g., crashes).
3) **Triage:** Test cases are grouped based on their observed execution behavior: those deemed *interesting* are saved and prioritized for future rounds of mutation, those deemed *uninteresting* are discarded, and those triggering *bugs* are saved for offline analysis by practitioners.

Fuzzing's most common form in practice is *coverage-guided fuzzing* [27], [28], [14]: a strategy that aims to maximize exploration of the target's code by mutating *only* those test cases that reach previously-unexplored code regions. To achieve this, coverage-guided fuzzers augment fuzzing's execution with program instrumentation (e.g., compiler instrumentation [27], dynamic binary translation [29], [30], [31], or static binary rewriting [32], [33], [34]) to collect each test case's *code coverage* (e.g., basic blocks [7] or control-flow edges [14]). Code coverage novelty is then assessed in the triage phase: test cases reaching *new* coverage are saved and fed back into test case generation, with the goal being to mutate them and further increase the frontier of code

coverage; while test cases covering *already-seen* code are merely discarded. Popular coverage-guided fuzzers include AFL++ [14] and libFuzzer [27].

2.2. Process Execution State

At a high level, *process state* refers to a program's full execution context at any point in its execution: its active instruction and subroutine, and the values of its variables and other data objects. From a lower level, state spans four fundamental constructs: local *stack* state, *register* state, *heap* memory state, and *global* data state.

Stack State. The stack (or call stack) is a linear, last-in-first-out data structure whose state represents a program's active subroutines and their corresponding local data (e.g., arguments, return address, variables). As subroutines are entered during execution, they are allocated temporary stack memory regions—called frames—at the stack's top-most region.[1] When a subroutine returns, its frame is cleared, and the preceding frame (i.e., the *caller* of the now-returned subroutine) becomes the stack's new top-most frame.

Register State. Execution also changes the state of several small-yet-fast temporary storage bins called registers. In stack frame allocation, the *stack pointer* and *base pointer* (or frame pointer) registers take on the active subroutine's frame boundaries: the stack pointer (64-bit: `rsp`, 32-bit: `esp`) is assigned the stack's top-most address, while the base pointer (64-bit: `rbp`, 32-bit: `ebp`) denotes the address directly after the previous frame. As instructions are executed, the *instruction pointer* register (64-bit: `rip`, 32-bit: `eip`) is also updated to reflect the address of the current instruction. Additionally, other general-purpose registers are updated whenever they are used for temporary storage (e.g., `mov` **`rax`** ← `0x1234`).

Heap State. The heap is a hierarchical data structure accommodating *dynamic* memory allocation (e.g., using `malloc()`, `calloc()`, `realloc()`, or C++'s `new`). At a high level, blocks of heap memory are allocated by their size, with pointers used to access the resulting regions. Upon deallocation, freed regions immediately become available for future allocation. However, while stack memory is controlled by code generated by the compiler at function call boundaries, the responsibility of managing heap state is left entirely to the *programmer*.

Global State. Beyond local data, software commonly uses global data: objects instantiated at program entry (i.e., non-local to any subroutines). Most executable formats (e.g., Windows PE32+, Linux ELF) store global data in distinct memory regions (e.g., `.bss` and `.data`); at runtime, they are loaded into the virtual address space, and subsequently cleared on exit. As global program state is accessible—and often mutable—by the entire application, it is deeply intertwined with modern programs' decision-making logic.

1. For brevity, we refer to the stack's lowest address as its *top*.

2.3. Current Fuzzing Execution Mechanisms

Fuzzing scrutinizes programs by generating—and executing—a massive volume of test cases. At the core of fuzzing's test case execution are *execution mechanisms*: the machinery tasked with initiating the target process per every incoming test case and cleaning things up after execution completes. As shown in Figure 2, current fuzzing execution mechanisms span four types: *process creation*, *forkserver-based cloning*, *in-memory looping*, and *kernel-based snapshotting*. We discuss each below.

Process Creation. Process creation represents the flagship program execution technique supported by Windows. At a high level, process creation works as follows:

1) Load the target executable file into a new child process.
2) Initialize the child process and begin executing its code.
3) On exit, free the child process' resources and wait for a new incoming test case.

Since Windows 3.1, process creation is facilitated by the `CreateProcess()` API [39]. In fuzzing, the steps of process creation are performed for *each* generated test case. Many Windows fuzzers support or rely on process creation, such as WinAFL [18], Manul [35], and KillerBeez [36].

Forkserver-based Cloning. In 2014 [12], Linux-based fuzzers began adopting forkserver-based process cloning: a technique to initiate test case executions from *after* target initialization. Forkserver-based cloning operates as follows:

1) Load the target once, creating a *forkserver* process.
2) Instrument a post-initialization target subroutine (e.g., `main()`) to perform copy-on-write process cloning.
3) For each test case, fork a new child and execute it.
4) On exit, free child process resources and go to Step 3.

By performing target initialization just once, forkserver-based cloning is spared the cost of re-executing initialization routines per test case. While the forkserver's required copy-on-write cloning primitives are not directly supported by Windows, Jung et al. [19] reveal the possibility of reverse-engineering hidden copy-on-write cloning primitives from Windows' closed-source kernel.

In-memory Looping. Unlike process creation and forkserver-based execution, in-memory looping avoids spawning processes by instead executing in a single, persistent (non-exiting) child. These techniques (e.g., AFL's persistent mode [14]) interpose a loop around the subroutine(s) targeted for fuzzing, with each loop iteration devoted to a single test case. Its key steps are:

1) Load the target once, creating a *persistent* process with the target subroutine (e.g., `main()`) wrapped in a loop.
2) For each test case, perform a new execution of the loop.
3) On exit, jump back to the loop's entry from Step 2.

Many Windows fuzzers support or rely on in-memory looping, like WinAFL [18], TinyAFL [37], and Jackalope [38].

Kernel-based Snapshotting. Recent state-of-the-art fuzzers (e.g., AFL++ [14]) are capitalizing on an emerging

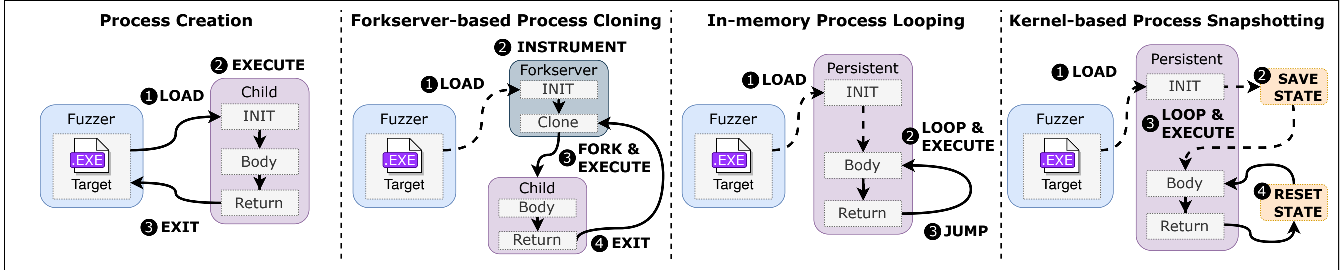

Figure 2: A high-level overview of the workflows of current process execution mechanisms used in fuzzing: process creation [18], [35], [36], forkserver-based process cloning [19], in-memory process looping [18], [37], [38], and kernel-based process snapshotting [13], [17], [16].

Execution Mechanism	Fuzzing Implementations	Level of Efficiency Target	Level of Efficiency Kernel	Execution Correctness	Windows Kernel Compatibility
Process Creation	WinAFL [18], Manul [35], KillerBeez [36],	✗	✗	✔	**full**
Forkserver-based Cloning	Winnie [19]	✔	✗	✔	**partial**
In-memory Looping	WinAFL [18], TinyAFL [37], Jackalope [38]	✔	✔	✗	**full**
Kernel-based Snapshotting	AFL++ LKM [17], Xu et al. [13], Zhao et al. [16]	✔	✔	✔	**none**
Target-embedded Snapshotting	WinFuzz	✔	✔	✔	**full**

TABLE 1: A feature comparison of fuzzing execution mechanisms process creation, forkserver-based process cloning, in-memory process looping, kernel-based snapshotting, and target-embedded snapshotting.

execution technique called *snapshotting*: an extension of in-memory looping where process state is reset between executions. At a high level, snapshotting operates as follows:

1) Load the target once, creating a *persistent* process with the desired subroutine (e.g., `main()`) wrapped in a loop.
2) After the target is initialized, save its full process state.
3) For each test case, perform a new execution of the loop.
4) On exit, reset process state and jump back to Step 3.

Existing efforts implement their specialized snapshot primitives via custom kernel extensions. Currently, all snapshotting fuzzers are Linux-based (e.g., AFL++ LKM mode [17], Xu et al. [13], Zhao et al. [16]).

3. Motivation: Inefficient, Incorrect, and Incompatible Windows Execution Mechanisms

Achieving fast and effective fuzzing on Windows programs is challenging. We explore the challenges inherent to Windows fuzzing and investigate the fundamental limitations of existing execution mechanisms found in popular fuzzers (Table 1)—efficiency, correctness, and kernel compatibility—distilling a criteria of the ideal design qualities a Windows execution mechanism must achieve.

3.1. Limitations of Windows Fuzzing Execution

The *Inefficient:* Process Creation and Cloning. Effective fuzzing demands a high test case throughput. As test case execution is shown to make up over 90% of fuzzers' runtimes [7], a fuzzer's speed hinges on its execution mechanism's efficiency. Unfortunately, current Windows fuzzers are restricted by the high target and kernel overheads of process creation and initialization. Process creation's re-execution of the *full* program incurs the cost of memory management, binary loading, dynamic linking, library initialization, and other target-specific startup routines for *every* test case, making it fuzzing's slowest execution mechanism.

Porting the Linux-based strategy of using a forkserver (e.g., AFL [15]) is not a complete solution to Windows' poor fuzzing performance. While forkserver-based cloning differs from process creation in that it executes test cases from a pre-initialized target state, *both* execution mechanisms see significant overhead from their numerous—and fuzzing-irrelevant—kernel procedures invoked when spawning each child process (e.g., address space initialization, process ID assignment, resource duplication, background memory management to support copy-on-write, and OS state updates) [13], [16]. Additionally, forkserver-based cloning still incurs the cost of process teardown. Worse yet, the

opaque mechanics of the Windows kernel adds further bloat to these mechanisms' primitives, making them **over 20–1,000x slower** than their Linux counterparts [19], [21]. To uphold Windows fuzzing speed—and effectiveness—an execution mechanism must maintain a two-fold **efficiency across both the target program *and* the kernel.**

> **Criterion 1:** Efficient across the target program and kernel.

The *Incorrect:* In-memory Looping. Fuzzing's many test case executions constantly change the target program's stack, register, heap, and global state. A corruption of target state (e.g., an instruction pointer overwrite) typically resolves in a *crash*, indicating that fuzzing has uncovered a memory safety violation (e.g., a stack buffer overflow), causing the test case to be saved for post-fuzzing vulnerability triage. However, fuzzing's goal of finding error-inducing test cases is quickly derailed when process state is corrupted *by the execution mechanism*: though in-memory looping offers the most target- and kernel-efficient execution today, it does *not* reset process state between test cases—overwhelming fuzzers with *false-positive crashes* caused by polluted heap and global state [19]. While modifying target programs to reset their state themselves is easily achieved in open-source fuzzing contexts (e.g., OSS-Fuzz [40]), the often *closed-source* nature of Windows software—stripped, obfuscated, or otherwise opaque executables—makes the requisite reverse engineering and binary rewriting infeasible. Thus, to support effective Windows fuzzing, an execution mechanism must uphold correct execution: ***fully* and *automatically* restoring process state between test cases.**

> **Criterion 2:** Fully reset program state for correctness.

The *Incompatible:* Cloning and Kernel Snapshotting. Windows fuzzing efforts are integrating advancements from Linux fuzzers [41], [42], [43], [44], however, reliably porting Linux-based execution mechanisms to Windows remains an unsolved challenge. The closed-source nature of the Windows kernel prevents the addition of the third-party execution primitives used by popular kernel-based snapshotting efforts [13], [16], [17]. While Jung et al. [19] introduce a form of forkserver-based execution to Windows by reverse-engineering hidden copy-on-write primitives from its proprietary kernel components (i.e., `ntdll.dll`, `NtCreateUserProcess`, and the `CSRSS` subsystem), their technique is ultimately *kernel-specific*—and officially unsupported beyond Windows 10 v1809 build 17763.973 [20]—requiring unscalable manual re-tooling whenever the kernel is updated or for older versions. As the Windows kernel continues to see significant change each year—with more than ten major updates released since 2015 [45]—expanding fuzzing's reach across the ever-growing Windows software ecosystem demands an execution mechanism that is **fully *kernel-agnostic*.**

> **Criterion 3:** Kernel-agnostic performance and effectiveness.

3.2. Towards Fast & Effective Windows Execution

Our survey of popular fuzzing execution mechanisms (Table 1) reveals significant limitations with respect to Windows fuzzing: process creation is easily supported via Windows' standard primitives, yet its per-test-case re-execution of target initialization and kernel bookkeeping procedures quickly deteriorates fuzzing's throughput; forkserver-based cloning sees higher speeds from avoiding target re-initialization, but it pays many of the same heavy kernel costs as process creation, leaving its overall performance low; in-memory looping offers the least-invasive execution, but its inability to reset process state between test cases derails fuzzing with spurious crashes and otherwise incorrect execution behavior; and while snapshotting represents the best trade-off of speed and correctness, the closed-source nature of the Windows kernel leaves both it and forkserver-based cloning broadly incompatible with modern Windows systems. The lack of an *efficient*, *correct*, and *fully-compatible* Windows execution mechanism leaves Windows software fuzzers orders-of-magnitude slower and less effective than their Linux counterparts.

> **Impetus:** To bridge the speed and effectiveness gap between Linux and Windows fuzzing, we apply our criteria of execution mechanism design qualities. Namely, that an execution should ① be efficient across the target program and kernel; ② maintain correctness by restoring process state between test cases; and ③ be realized in a kernel-agnostic design.

4. Target-embedded Snapshotting

We present *Target-embedded Snapshotting*, a new fuzzing-oriented execution mechanism enabling efficient, correct, and kernel-agnostic fuzzing on Windows. Target-embedded snapshotting combines the efficiency of the fastest execution mechanism (in-memory looping) with the correctness of process creation by enabling fuzzing targets to restore their own state, using only language- and API-level functionality. The result is a fast, correct, and kernel-agnostic execution mechanism. As stated previously, in-memory looping is fast, but not correct, since it uses the same process to run multiple executions of a single program, leading to inconsistent—and semantically impossible—test case starting states across executions. Starting from an impossible program state leads to both missed crashes (i.e., false negatives) and impossible crashes (i.e., false positives). Missing crashes results in bugs remaining hidden, while creating new, impossible crashes wastes programmer time in the already human- and time-intensive task of triaging fuzzer-found crashes.

Our solution is to harness the speed of in-memory looping, but to reset program state between test cases—solving the correctness problem. Where previous approaches create an entirely new process for each test case or use kernel support for on-demand state copying, we focus on hooking programs at the binary level to have them track and restore their own state. Essentially, we snapshot program state before starting the first test case, then restore that snapshot between all subsequent test cases. The challenges that we solve are ① identifying program state that changes during execution, ② efficiently tracking that state, and ③ efficiently restoring all state changes between test cases.

4.1. Worked Example

To demonstrate how target-embedded snapshotting works, imagine we want to fuzz a typical command-line interfacing Windows program that parses files from disk, such as `tcpdump`. During execution, the program uses a global variable as a counter to iterate through its command line arguments, as shown in Listing 1. The target also makes several heap allocations to use as data buffers, and subsequently frees *some* of them later in its execution.

In such programs, a state-unaware execution—such as in-memory looping—creates a two-fold risk of state *corruption*: first, repeatedly looping over the program will clobber written global variables (e.g., `optind` in Listing 1), leaving future executions with this variable initialized incorrectly; and second, the non-freed heap chunks will result in memory leaks during all subsequent executions. These subtle state corruptions will eventually cause the target process to *crash*, presenting the fuzzer with what appears to be a valid bug—despite it being a *false positive*.

```
1 while ((op = getopt_long(argc, argv,
2     SHORTOPTS, longopts, NULL)) != -1)
3     switch (op) {
4    ...
```

Listing 1: Command-line iterating code found in tcpdump 4.99.1. Function `getopt_long()` takes global variable `optind` (not shown) as a counter.

To eliminate the possibility of state corruption, target-embedded snapshotting operates as follows. After the target initializes and reaches `main()`, our instrumentation creates a snapshot of the program's state—capturing the values of the stack, CPU registers, and global data. To track heap state, we hook and save the addresses of all dynamically-allocated memory regions; and should a memory region be freed, we stop tracking it. To reset process state for subsequent executions, we ① restore the original stack frames, registers values, and global variables from our snapshot; and ② free all of the target's outstanding heap allocations. In the following sections, we provide a finer-grained view of the technical details behind our techniques.

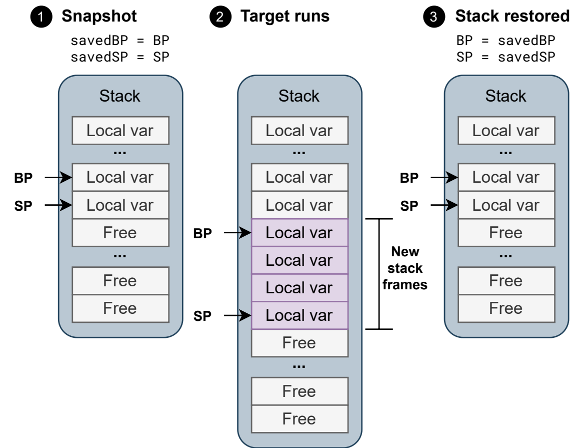

Figure 3: Overview of the stack. Note that the stack traditionally grows towards lower addresses, and the diagram is drawn with higher addresses above lower addresses.

4.2. Resetting Program State

Our technique automatically tracks and restores program state changed during fuzzing: the stack, registers, heap, and global memory. We describe our handling of each below.

Stack. The stack is a region of a process's memory used to store local variables. Like a LIFO, a process only adds or removes variables from the top of the stack, which is an address in the underlying memory region indicated by a special register called the stack pointer. The stack pointer divides the stack's memory region into two contiguous parts: used memory and free memory. New memory is allocated by "pushing" data to the top of the stack, which moves the stack pointer up by the new allocation's size, increasing the size of the used part of the underlying memory region and reserving memory where the new data is written. Memory is freed by "popping" data from the top of the stack, which saves the popped data to a register and moves the stack pointer down by the amount of memory freed, decreasing the size of the reserved part of the underlying memory region and allowing the memory to be used by a subsequent push.[2] This interface enables automatic memory management for local variables, because each function call pushes its local variables to the stack, and is responsible for popping them off the stack before it returns. Because the stack is used through this interface, new memory can only be allocated at or freed from the top of the stack. So, at any given time, anything above the top of the stack is undefined, and anything below the top of the stack is currently in use. Each function call adds a new stack frame to the top of the stack, which is used to hold the local variables existing within the scope of that function. When a function returns, its stack frame

2. Incrementing or decrementing the stack pointer can also be used to free or reserve new stack memory. This is equivalent to pushing or popping, except it discards the popped values and leaves pushed values uninitialized.

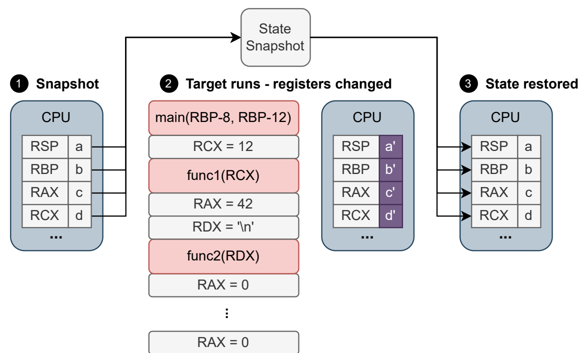

Figure 4: Overview of register state restoration.

is removed from the stack, moving the stack pointer back to the value it had before the function was called and thus automatically freeing up the memory it used.

Resetting the Stack. To reset the stack to the state it had before calling the target function, we erase all of the stack frames created after our harness's call to `main()` (Figure 3). If we know the value of the stack pointer before calling the target function, we know anything on the stack above that point must have been created after calling the target function. We then free all of this memory by moving the stack pointer back to its original value. We also set the base pointer, which points to the bottom of the current stack frame, back to its original value, since local variable addresses are often defined in terms of the base pointer.

Registers. Registers hold values currently being operated on by a process. They change frequently during execution, since most instruction parameters must be placed in registers, and they also may be used to hold function parameters or function return values.

Resetting Registers. To prepare for a new iteration, we reset each register to its original value from the snapshot (Figure 4). Most of these values will likely be ignored by the target and simply overwritten. If the target does take in register parameters, they would fall into two possible categories: simple values or pointers. If a parameter is a value, its meaning is self-contained in the value held by the register, so we can safely restore it from a copy. If a parameter is an address, its meaning is defined by some data located elsewhere in the address space of the process. If we allowed the target to exit the scope of the function that calls the target function, this data could potentially be lost to a `free()` or stack frame destruction. We avoid calling any cleanup routines or destroying important stack frames by only calling the target from injected code, so the target returns safely to our code instead of the original binary. This ensures that pointers existing in the scope of the target function will not be cleaned up by the target, so we can safely restore register state by copying and restoring register values.

The target may also call functions that end the process before it would return naturally, such as `TerminateProcess()` or `ExitProcess()`. This creates another possible return path from the target function, which will have a different stack and register state than the more direct case in which it returns normally. If the target function always returned naturally, we could assume that everything it placed on the stack would be cleaned up, since returning from a function destroys its stack frame and any stack frames created after calling it. However, we may have to end the iteration from within the context of `ExitProcess()` or `TerminateProcess()`, which makes us responsible for cleaning up stack and register state. In this context, multiple target-created stack frames will still be present on the stack, and registers may hold incorrect values. We clean up stack frames by resetting the stack and base pointers to their original value, which automatically removes the extra data on the stack. Whether the target function returns normally or not, some registers might have a different state than when the target function was first run, since functions are not always guaranteed to preserve register values. The compiler should enforce a calling convention that prevents code from relying on caller-save registers being preserved, but for the sake of simplicity, we restore every register whether the target returned naturally or through `ExitProcess()`/`TerminateProcess()`, since the overhead of doing this is minimal.

Heap. Like the stack, the heap is another region of a process's memory used as storage for variables. Unlike the stack, the heap grows in size to hold large amounts of data, and new allocations must be managed manually by the programmer. Manual memory management allows programmers to allocate memory without being bound to any particular lifetime or scope, but it also makes the programmer responsible for properly freeing allocated heap memory. Failing to do so causes unused heap memory to accumulate over time. A program that runs for a short period of time may not have much reason to care about cleaning up its heap allocations, since this will be done automatically by the operating system once the process exits. However, in-memory looping executes large numbers of target function iterations in the same process, making even a small leak a significant threat to fuzzer stability.

Resetting the Heap. We cannot assume targets will behave correctly, especially during fuzzing, and we are circumventing an OS feature that programmers may rely on by reusing the process and preventing natural heap cleanup, so we need to keep track of any heap allocations the target makes to prevent memory leaks. We install hooks on all functions that allocate heap memory to keep track of allocations made by the target, and then free any unfreed allocations at the end of each iteration (Figure 5). Also, we prevent the target from freeing any allocations it did not make: if a pointer was created outside the scope of the target function (i.e., before our harness was invoked), we need to

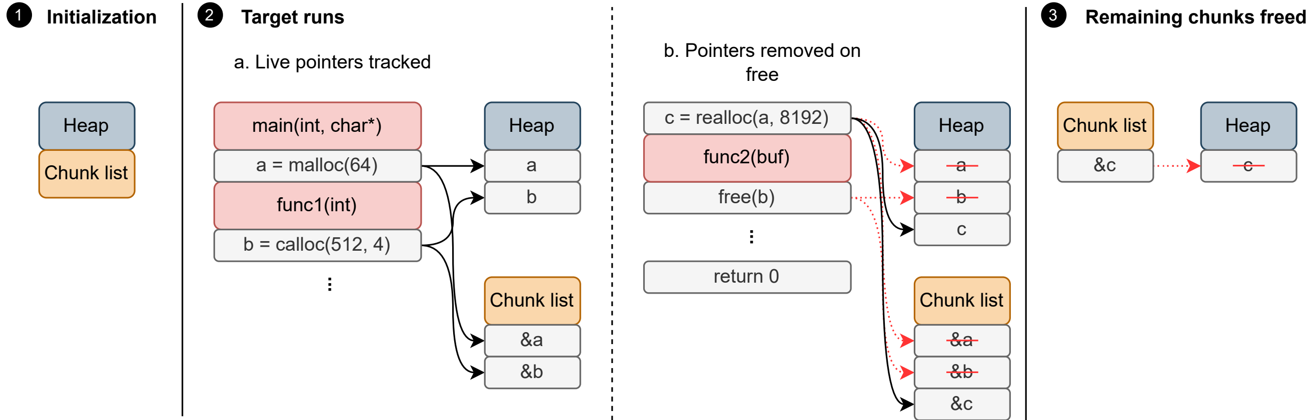

Figure 5: Overview of heap memory tracking. At any given time, the chunk list holds pointers to all heap chunks allocated by the target, so they can easily be freed after the target exits.

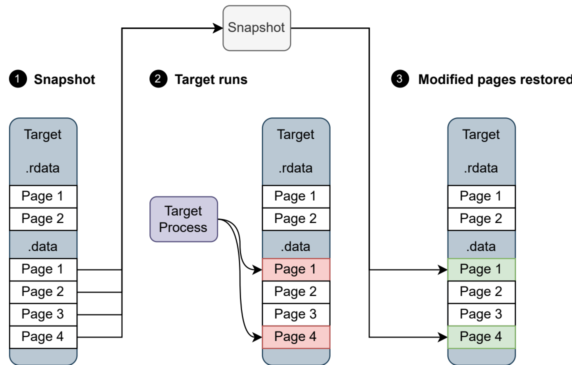

Figure 6: Overview of global memory restoration. Only mutable global state is tracked, and only modified pages are restored.

preserve it across multiple iterations.

Global Memory. Global memory correctness is one of the biggest sources of error for in-memory looping. Global memory often contains data that influences program behavior (e.g., global variables), and under normal execution, programs can safely assume they will start execution with the global state intended by the programmer. In-memory looping violates this assumption as soon as the target function modifies any part of global state, causing incorrect behavior.

Resetting Global Memory. We rectify global memory correctness issues by restoring every page of global memory the target modifies to our known-good state between each iteration (Figure 6). In order to maximize efficiency, we embed snapshotting code into the target, so that state copies are stored in the address space of the program, and state restoration is done from within the target process itself. This avoids the overhead of having to call `WriteProcessMemory()`/`ReadProcessMemory()` or an equivalent, or having to transition control to another process. After adding some other optimizations described in § 5.2, we retain almost all of in-memory looping's performance while maintaining semantic correctness.

For network-based targets, we provide function hooks to remap the input file to data received over a socket. These hooks also prevent corruption of process state from socket API misuse by preventing the target from interacting with the real socket API at all. We hook calls that open files to manage access to files in a similar way.

This approach also considerably relaxes the restrictions on the target function compared to simple in-memory looping. Fuzzers based on in-memory looping expect their users to make sure the target function does not modify global program state or leak memory, but this imposes a large burden on security analysts, who have to either restrict themselves to fuzzing pure functions, or manually debug their target and create harnesses to reset problematic state changes themselves. Target-embedded snapshotting eliminates the requirement for the target function to be pure by automatically resetting all relevant state, which enables plug-and-play whole-program fuzzing for existing binaries.

Other Residual State. In addition to those addressed previously, there are many other residual elements of program state that would persist across iterations. For example, these would include: IPC and synchronization objects (such as mutexes, condition variables, or memory mapped files), installed exception handlers, chosen locale, or other OS-specific features (e.g. on Windows, registry artifacts, queued APCs, GDI objects, COM objects, or NTFS transactions). We did not encounter residual state from these elements in any of our benchmarks, and they represent edge cases which could be supported with minimal instrumentation similar to our existing design, which tracks relevant state changes so they can be removed later. For example, a fuzzing target on Windows may create a mutex with `CreateMutexA()`, which returns a handle to a new mutex object. To ensure this object does not affect subsequent iterations, it must be

released with `ReleaseMutex()`. We would accomplish this in the same way we clean up heap memory; with hooks that track and remove leaked objects.

5. Implementation: WinFuzz

To create the highest-performance Windows fuzzer, we implement target-embedded snapshotting atop of Winnie [19]. Winnie is a Windows implementation of coverage-guided tracing [7] on top of WinAFL [18], making it the current fastest Windows fuzzer. Note that because target-embedded snapshotting is contained within a fuzzer's execution mechanism, our approach is compatible with any coverage tracing system. That is to say that our approach is orthogonal to the coverage tracing approach.

5.1. Design Overview

Our system functions in four main steps: interposing on the target process, taking the state snapshot, tracking state changes, and restoring state.

Interposing on the target: The target process is created in a suspended state using a call to `CreateProcess()` with the `CREATE_SUSPENDED` flag set. While the process is suspended, we use `CreateRemoteThread()` to run `LoadLibrary()` in the target process, and pass in the path to our injected DLL. This loads our DLL into the target process and runs our `DLLMain()` function. Our `DLLMain()` function installs a hook onto the target function that sets up our target-embedded snapshotting system by installing our state tracking hooks and taking the initial state snapshot. We resume the target process, and let it initialize normally until it reaches the target function hook, at which point we install our heap memory tracking hooks and our guard page exception handler.

Snapshotting: In addition to setting up our state tracking hooks, we also take the initial state snapshot right after the target function is reached. This locks-in the changes from process initialization, while providing a rollback point to discard state changes due to individual test cases.

Tracking state changes: During execution, our heap hooks and guard page exception handler track changes to the heap and global memory regions, respectively.

State restoration: Every time the fuzzer executes the target function, we first reset the stack and heap, followed by restoring the process's register and global state from the snapshot.

5.2. Achieving State Restoration

Our system for target-embedded snapshotting involves tracking four components of program state: the stack, registers, heap memory, and global memory.

Registers and Stack. To efficiently restore the target's registers to their known-good values, we use the x86 assembly code shown in Listing 2.

```
1  xor eax, eax
2  not eax
3  mov edx, eax
4  lea ecx, [xsaveData]
5  xrstor [ecx]
6  mov esp, [savedregsEsp]
7  popfd
8  popad
9  mov esp, [savedEsp]
10 call [fuzz_iter_address] // Call target
11 mov [in_target], 0 // Disable memory hooks
12 jmp [report_end] // Finish iteration
```

Listing 2: Injected assembly snippet for register state restoration.

This includes the registers controlling the stack, so the stack is also reset at the same time as the other registers. Our assembly code only assumes that you can save and restore register values, use global variables, and jump to or call functions, so it is portable to any major architecture, but this implementation is architecture specific.

Heap. To prevent memory leaks, we track all heap memory allocations the target makes using function hooks attached with the Detours library [46]. We use a global variable (`in_target` in the previous assembly code) to denote whether or not we are currently executing the target. Setting `in_target` to 0 disables all of our heap memory hooks, to prevent our own heap allocations from being tracked. While this flag is set, we track any heap allocations from the target, and maintain a list of all heap allocations the target has not freed. At the end of every iteration, we iterate through the list of unfreed heap allocations, freeing them.

Global memory. To ensure the correctness of global memory, we create a master copy of any global memory sections that the target has write permissions to. These sections are identified by parsing the PE binary header, which contains a list of all sections in the binary and their permissions. On Windows, a binary will typically contain two global data sections: a `.rdata` section that contains read-only data, and a `.data` section that can be modified, so for most binaries we copy and track the `.data` section. We allocate the section copies on the heap of the target process, and copy the original sections from the address specified in the PE header using `memcpy()`. The address of each copy is stored together with the address of its original section, which we use to create a function that takes the address of a page in a mutable global section in the target, and returns the address of our copy. We use this function to efficiently restore the state of global memory sections from the master copy with another `memcpy()`.

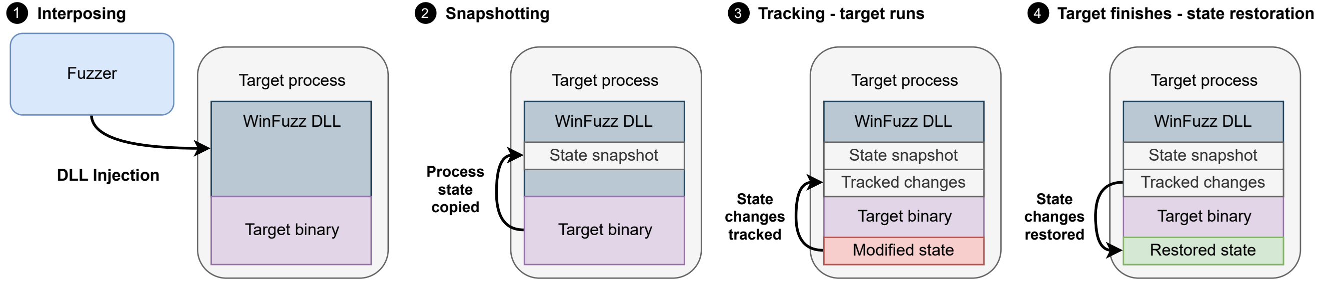

Figure 7: High-level overview of WINFUZZ's operation.

5.3. Optimizing Global Memory Restoration

To avoid restoring every mutable page in the target on every iteration, we add a system to track which pages have been modified, and then only restore the modified pages. At the beginning of every iteration, guard pages [23] are placed on all tracked sections. When these sections are accessed by the target at runtime, a guard page exception is triggered, which runs our guard page exception handler. The guard page exception handler checks the address of the exception to make sure it refers to a page we placed a guard flag on, and then marks that page as modified. Guard pages are automatically removed when they trigger an exception, so no further accesses by the target will incur any performance penalty. Finally, before the beginning of the next iteration, these marked pages will be restored from our snapshot. This guard page system minimizes the overhead of our snapshotting system, since we will only restore pages that were actually touched by the target.

Although this use of guard pages may not be supported on all operating systems, it is possible to implement the effect in a portable way by removing write permissions from tracked pages, and then handling access violation exceptions. We provide an alternate implementation that does this in our code, which can be enabled with a preprocessor switch (`#define USER_SEGHANDLE`). We use guard pages because they are faster than using access violation exceptions, despite the fact that they will actually mark pages as requiring a reset on both reads and writes.

6. Evaluation

We focus our evaluation on three major criterion that WINFUZZ improves: ① **Correctness:** does our new execution mechanism maintain state correctness across test cases? ② **Performance:** does WINFUZZ increase both test case throughput and total code coverage? ③ **Kernel Compatibility:** how well does WINFUZZ support the ever-changing Windows operating system?

6.1. Evaluation Setup

We evaluate our solution by comparing it to the other state-of-the art greybox fuzzers on Windows: Winnie (highest performance) and WinAFL (widest adoption). We configure WinAFL to use DynamoRIO instrumentation in block coverage mode, and configure WINFUZZ and Winnie to use their built in coverage-guided tracing instrumentation. We use the standard C program entry point function (`main()`) as the target function for fuzzing, with no harness code to enforce correct program behavior, in order to demonstrate WINFUZZ's ability to perform out-of-the-box whole program fuzzing. We run all of our evaluation experiments on Azure instances using Windows 10 Pro 21H2 (build 19044.1889) with a single Intel Xeon core running at 2.1 GHz, with 3.5 GB of RAM.

Benchmark Selection. To ensure a rigorous evaluation of WinFuzz, we use a benchmark corpus (Table 2) representative of real-world Windows binaries, with a wide range of sizes, total basic blocks, and file formats. We attempted to reuse Winnie's benchmarks, however as detailed in the Winnie paper, they require custom harnesses—which Winnie's developers have not released (likely due to them targeting closed-source/commercial binaries, which may violate the "no reverse engineering" clauses for those programs' licenses). To match the benchmark characteristics evaluated by Winnie—but with publicly available harnesses—we have chosen a separate set of binaries that do not require custom harness code, including several that are closed-source (`tar`, `nconvert`, `smpdf`, and `irfanview`). We expect that our more readily-harnessable benchmark set will facilitate future Windows fuzzing research.

Issues with Coverage Generation on Winnie. Winnie failed to generate any coverage while running certain benchmarks, due to a bug. These benchmarks are marked as "No cov" in the table, meaning that the fuzzer runs, but does not find any new paths, indicating incorrect execution or a coverage system bug. WINFUZZ is able to find coverage for these benchmarks using the same coverage system, so this bug is related to Winnie's execution mechanism. We reported the bug to Winnie's developers, but they could not fix it. The developers also state that they no longer maintain

Program	WinFuzz	Winnie	WinAFL	Source	File Format	Size (KB)	Number of Basic Blocks
tar	✔	✔	✗	Proprietary	.tar	606	30758
nconvert	✔	✔	✗	Proprietary	.png	2458	91550
freetype	✔	✔	✔	Open Source	.ttf	482	20891
audiofile	✔	✔	✔	Open Source	.wav	45	1504
flac	✔	✔	✔	Open Source	.flac	686	19292
nanosvg	✔	✔	✔	Open Source	.svg	47	1966
sqlite	✔	✔	✔	Open Source	.db	802	46758
smpdf	✔	No cov	✗	Proprietary	.pdf	3379	39816
irfanview	✔	No cov	✔	Proprietary	.png	1946	55187
jq	✔	No cov	✗	Open Source	.json	2662	13965

TABLE 2: Overview of fuzzer compatibility with benchmarks. “No cov” indicates a benchmark that Winnie failed to produce new coverage for, and ✗ indicates a benchmark that did not run for that fuzzer.

Winnie as of January 2022 [20].

6.2. RQ1: Correctness

To verify that our state restoration system maintains correct process state, we perform a correctness test for each binary using a set of saved test cases from our previous experiments (a full 24-hour fuzzing trial), because these test cases capture all target behavior seen during fuzzing. To check that our system restores state for each input correctly, we compare two state snapshots taken after parsing the input, and just before the process reports its coverage and finishes the iteration: one from a new process, and one taken after executing all other inputs in the queue. [3] This snapshot includes the general purpose registers and global variables of the target, which capture the ending point (stack pointer, return value) and problematic global variable changes. If the program is executing correctly, we expect it to make exactly the same set of changes to program state in both snapshots.

Nondeterministic State. Some state may vary between executions (e.g. timestamps, pseudorandom number generator output), and we detect these by capturing multiple snapshots for the same input, each in a fresh process. If an element varies between any two executions, it is marked as nondeterministic, and excluded from the test. Of course, in a nondeterministic value that spans multiple bytes, changes to every byte may not be observed (e.g. only the least significant byte is affected by arithmetic operations). To mitigate this, we consider bytes within a short distance (3 bytes) of nondeterministic values as also being nondeterministic.

We run the correctness test using a full saved queue to verify each of our benchmarks, and see no state correctness issues. [4] We include the correctness test as part of our open source implementation, so it can be used with any properly harnessed fuzzing target to verify correct state restoration.

3. To be precise, these snapshots are taken immediately after the target has exited into our code, and just before we report coverage to the fuzzer process and end the iteration.

4. Queue sizes - `tar`: 240, `nconvert`: 449, `freetype`: 142, `audiofile`: 19, `flac`: 216, `nanosvg`: 63, `sqlite`: 173, `smpdf`: 145, `irfanview`: 402, `jq`: 102

6.3. RQ2: Test Case Throughput

Previous work shows throughput is the most important factor in discovering new bugs [12], [13], [14], [33], so we compare WinFuzz’s test case throughput to the execution mechanisms offering correct execution on Windows: Winnie’s custom forkserver implementation, and WinAFL’s process creation. We run 5x24-hour trials for all benchmarks that each fuzzer supports. Table 3 shows the average throughput increase WinFuzz provides over Winnie and WinAFL. The results show that WinFuzz scales to targets of varying sizes and complexities, and outperforms both Winnie and WinAFL.

`Irfanview` shows the least performance improvement of all our benchmarks, which we determined was due to it opening a window on every iteration, despite being command line interfacing. This adds a significant overhead to each execution which could have been avoided with a more complicated harness that skips execution of the GUI code and exposes the program’s parsing code to the fuzzer directly, but WinFuzz is still able to execute new test cases more efficiently than both Winnie and WinAFL.

6.4. RQ3: Code Coverage

Discovery of new code coverage indicates the fuzzer has generated an input that reaches new target behavior, which is the most significant indicator a greybox fuzzer has that it is getting closer to discovering new bugs. At a high level, new code coverage generation is an indicator that the fuzzer is performing well. We use the saved test cases from the previous section’s experiments to evaluate WinFuzz’s code coverage generation, measured as new edges between basic blocks covered (Figure 8). We do not consider Winnie’s results for the benchmarks that it could not generate new paths for in our evaluation of code coverage generation.

Differences in Code Coverage Systems. WinAFL uses a coverage system that can track how many times each basic block is visited, whereas the coverage-guided tracing used in Winnie and WinFuzz will only report new coverage when a previously undiscovered basic block is executed. In a comparison with coverage-guided tracing, WinAFL’s

Benchmark	Winnie	ρ	WinAFL	ρ	WinAFL (No hitcounts)	ρ
tar	5.83	0.004	✗	✗	✗	✗
nconvert	7.69	0.004	✗	✗	✗	✗
freetype	8.09	0.003	209.73	0.002	199.08	0.001
audiofile	8.86	0.001	221.76	0.002	220.09	0.001
flac	6.62	0.001	181.68	0.002	175.41	0.002
nanosvg	8.86	0.003	250.70	0.003	233.89	0.004
sqlite	7.24	0.004	182.83	0.007	172.73	0.002
smpdf	6.08	0.002	✗	✗	✗	✗
irfanview	1.40	0.017	49.22	0.002	44.83	0.001
jq	9.30	0.003	✗	✗	✗	✗
Average	7.00		182.65		174.34	

TABLE 3: WinFuzz's improvement in fuzzing test case throughput relative to Winnie and WinAFL, along with statistical significance values for each result computed with the Mann-Whitney U test.

Benchmark	Winnie	ρ	Paths	WinAFL	ρ	Paths	WinAFL (No hitcounts)	ρ	Paths
tar	1.08	0.148	1.18 (202)	✗	✗	✗	✗	✗	✗
nconvert	1.23	0.006	1.43 (408)	✗	✗	✗	✗	✗	✗
freetype	1.32	0.003	2.99 (52)	1.28	0.002	1.72 (90)	1.33	0.001	2.98 (51)
audiofile	1.05	0.037	1.30 (16)	0.95	0.216	0.84 (25)	1.06	0.002	1.49 (15)
flac	1.02	0.016	1.12 (210)	1.09	0.002	0.90 (262)	1.01	0.27	1.77 (134)
nanosvg	1.04	0.007	1.09 (58)	0.87	0.003	0.13 (472)	1.04	0.05	0.94 (67)
sqlite	1.26	0.004	1.62 (108)	1.16	0.007	0.63 (277)	1.26	0.002	1.84 (95)
smpdf	No cov	✗	✗	✗	✗	✗	✗	✗	✗
irfanview	No cov	✗	✗	0.97	0.012	0.81 (501)	1.06	0.001	2.87 (142)
jq	No cov	✗	✗	✗	✗	✗	✗	✗	✗
Average	1.15		1.53	1.05		0.84	1.13		1.98

TABLE 4: Relative improvement of WinFuzz's edge coverage generation versus Winnie and WinAFL, including ρ-values calculated using the Mann-Whitney U test. $\rho < 0.05$ indicates a statistically significant result. The "paths" column shows WinFuzz's relative improvement in the number of coverage-increasing inputs found, and the actual number of coverage-increasing inputs (queue size) in parentheses.

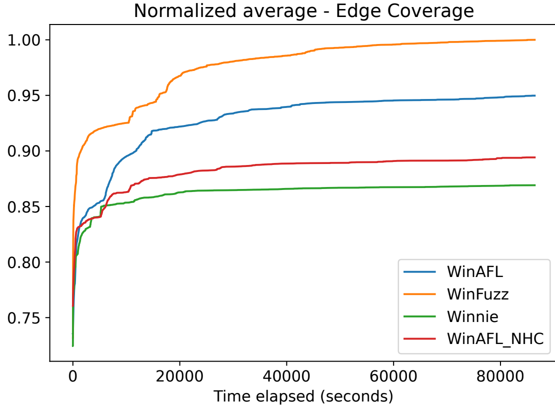

Figure 8: WinFuzz, WinAFL, and Winnie's mean edge coverage over time. We scale edge coverage relative to the lowest performer.

DynamoRIO-based dynamic binary instrumentation (DBI) sacrifices performance for the ability to recognize and queue test cases that produce new target behavior but do not hit any new basic blocks, which coverage-guided tracing would not recognize as potentially interesting. Coverage-guided tracing's goal is to maximize performance by removing all coverage instrumentation overhead for the common case of an input that does not encounter new code. This allows the fuzzer to execute inputs that do not encounter new basic blocks (the vast majority of test cases) at native speed, at the cost of not recognizing hitcount-increasing inputs as being potentially interesting. In summary, coverage-guided tracing moves more quickly towards inputs that cover entirely new regions of code, while WinAFL's DBI remains sensitive to smaller hitcount-only coverage changes that may be fruitful targets for mutation, at the cost of some performance. We also ran a set of experiments using a modified version of WinAFL that does not track hitcounts similar to coverage-guided tracing. This effectively negates the one benefit of WinAFL's slower instrumentation system, since it can no longer recognize when an input causes a hitcount-only coverage increase and thus discovers new target behavior, but the results are included in Table 3, Table 4, and Figure 8 for the sake of completeness.

Results: Winnie. Compared to Winnie, WinFuzz generates 15% more edge coverage, with every single benchmark averaging higher edge coverage versus Winnie. Mann-Whitney U tests indicate a statistically significant improvement ($\rho < 0.05$) on all but one benchmark (`tar`, $\rho = 0.148$). WinFuzz's improvement in number of coverage-increasing inputs ("paths") discovered for each trial corresponds to an improvement in edge coverage for each benchmark.

Results: WinAFL. Compared to WinAFL, WinFuzz generates 5% more edge coverage. Mann-Whitney U tests indicate a statistically significant improvement on 3 benchmarks, no difference on 1, and less edge coverage on

2 benchmarks. Although WINFUZZ has a much higher test case throughput than WinAFL, the differences in each fuzzer's coverage tracing system result in a large difference in the number of new coverage producing test cases found by each fuzzer, with WinAFL being able to recognize more potentially interesting test cases than WINFUZZ on every benchmark except for `freetype`. However, our results for edge coverage show that these test cases do not cover as much target behavior as the smaller number of queued test cases generated by WINFUZZ. This is due to the differences in coverage systems discussed earlier - WinAFL's hitcount sensitivity causes the fuzzer to recognize coverage increases that are due solely to differences in the hitcount of a basic block, which leads to more saved queue entries overall. A hitcount-only difference could be something as simple as running the same loop a different number of times, and does not necessarily lead to any new edges, whereas covering an entirely new region of code will always lead to a previously unseen edge. Thus, it follows that focusing processor time on discovering only new regions, as in coverage-guided tracing, will typically lead to a higher average edge coverage among inputs recognized as coverage-increasing compared to a slower, hitcount-sensitive instrumentation. However, this blinds the fuzzer to potentially interesting hitcount-increasing inputs, and a mutation path that leads to a bug could be missed. In the case of our experiments, we find that the saved inputs from WINFUZZ represent a larger number of basic block edges overall, despite being smaller on average than WinAFL's queues. Our experiments demonstrate that there are benefits to both coverage systems (coverage-guided tracing is faster and has better binary compatibility, while DynamoRIO recognizes more interesting test cases), and WINFUZZ's approach can be used with either instrumentation system or with Windows implementations of coverage-preserving coverage-guided tracing [11].

Results: WinAFL without Hitcounts. The modified version of WinAFL that did not recognize hitcount-only coverage increases generated less edge coverage than stock WinAFL on every benchmark. Given that this modification can only decrease the number of coverage-increasing inputs recognized and saved by the fuzzer, this is expected.

6.5. RQ4: Kernel Compatibility

To evaluate WINFUZZ's kernel compatibility, we attempted to compile and run it on several different versions of Windows. Winnie depends on specific undocumented kernel functions to implement its forkserver, so we evaluated its compatibility using the same procedure. We found that Winnie is able to compile and run on Windows 10 21H1 (build 19043.928) and 21H2 (build 19044.1826), but does not support Windows 8.1 or any version of Windows 11 we tested (builds 22000.856 and 22533.1001). During initialization, Winnie's custom forkserver implementation copies

OS	WINFUZZ	Winnie
Windows 8.1 - build 9600	✔	✗
Windows 10 - build 19043.928	✔	✔
Windows 10 - build 19044.1826	✔	✔
Windows 11 - build 22000.856	✔	✗
Windows 11 - build 22533.1001	✔	✗

TABLE 5: Fuzzer support for different Windows kernel versions.

Binary	Category	WINFUZZ	Winnie	WinAFL
flac	Null ptr deref	12.25 s	15.6 s	243.8 s
nconvert	Illegal address	2.1 hrs	✗	✗
nconvert	Invalid free	3.6 hrs	✗	✗
nconvert	Invalid ptr deref	7.2 hrs	✗	✗
nconvert	Heap overflow	8.5 hrs	15.8 hrs	✗
nconvert	Illegal address	✗	4.5 hrs	✗
nanosvg	Stack overflow	1.4 min	✗	✗
nanosvg	Null ptr deref	1.8 min	✗	✗
nanosvg	Null ptr deref	✗	✗	21.9 hrs
audiofile	Illegal Address	12 min	✗	5.2 hrs
	WINFUZZ's speedup		1.56x	23x

TABLE 6: Time-to-bug results for each fuzzer, calculated as average time taken by each fuzzer to find the first occurrence of each bug. WINFUZZ's relative improvement is calculated as the average improvement in bug discovery time for bugs found by both fuzzers.

an array of zeros to a region of memory in `ntdll.dll`, in order to prevent later calls to the CSRSS subsystem from crashing. The structure of `ntdll.dll` is not the same on Windows 8.1 as it is on Windows 10, and this causes the forkserver library to use an incorrect value for the size of this internal Windows structure. On a 32-bit system, the forkserver fails to compile. Compiling for a 64-bit system generates a call to `memcpy()` with a negative integer passed in place of an unsigned `size_t` parameter, which causes the forkserver to crash during initialization. On Windows 11, the forkserver library compiles and runs normally, but crashes while injecting the forkserver DLL into the target process. As shown in Table 5, WINFUZZ supports every version of Windows we tested, including Windows 11.

6.6. RQ5: Bug Discovery Time

The ultimate goal of fuzzing is to find crashing inputs for a program that expose a bug, allowing an analyst to identify and fix the bug. We evaluate WINFUZZ's ability to find new bugs by running 5x24-hour trials for each fuzzer to collect inputs that expose unique bugs, and comparing the time each fuzzer takes to find each bug. Our experiments found 10 unique bugs, with both Winnie and WinAFL finding 3, and WINFUZZ finding 8. Most of the bugs that WINFUZZ found were not discovered by WinAFL or Winnie, although they did find one bug each that WINFUZZ did not find. WINFUZZ achieved an average 1.56x and 23x faster bug discovery time versus Winnie and WinAFL, respectively (Table 6).

6.6.1. 0-day Bugs and Bug Disclosure. Of the bugs found during our original evaluation experiments, one was a 0-day (unpatched) bug, found in `nconvert`. This bug is an access violation, which causes the program to crash. We notified `nconvert`'s author of the bug on January 18, 2023, and received a response on January 26. To demonstrate the practical value of WINFUZZ, we also ran additional fuzzing trials solely for the purpose of finding new bugs, which discovered 8 additional 0-day bugs (see Table 7 for details of all bugs). This led to the discovery of the other bugs in Table 7, including the infinite loop in `AudioFile`, which was triggered by setting a field in the file to a value (-8) that would cancel out the other additions to the counter of a `for` loop. This `for` loop would only terminate when the counter had reached a certain value, but because the counter could never progress, the loop would never terminate. We notified the author and submitted a patch to fix the bug on March 12, 2023, we received a response on April 4, and the patch was merged on April 17. All of the additional bugs caused a unique crash in the fuzzer, and were verified in a memory error checking tool (Dr. Memory). For example, the invalid pointer write in `gpmf-parser` was caused by a value parsed from the file leading to the size passed to `malloc()` being zero, which then leads to an invalid write (heap overflow) later in the code when an 8-byte value is written to the address returned by `malloc(0)`. The authors of `jhead`, `flvmeta`, `gpmf-parser`, and `pdf2json` were all notified of their respective bugs on June 2, 2023.

Results: WinAFL. We also ran experiments using WinAFL to see if it could find the bugs found during our additional testing. WinAFL was given the same experimental setup as WINFUZZ: a desktop machine running Windows 10, with an 8-core Intel i7-9700K and 64GB of RAM, running 8 trials lasting 24 hours for every benchmark. WinAFL was able to find two of the same bugs that WINFUZZ found; one in `gpmf-parser` and one in `pdf2json`. Given that the only difference between Winnie and WINFUZZ is execution speed, we expect that Winnie would also be able to find these bugs, but after a much longer amount of time, similar to what was observed in our time-to-bug results.

Binary	Description
nconvert	2 invalid ptr reads, 3 invalid ptr writes
audiofile	Infinite loop
jhead	Invalid ptr read
flvmeta	2 invalid ptr reads
gpmf-parser	1 invalid ptr read, 1 invalid ptr write
gpmf-parser	Invalid ptr write
pdf2json	Stack buffer overrun (ntdll.dll)
pdf2json	Stack buffer overrun (pdf2json.exe)
pdf2json	Stack overflow
Total 0-day bugs:	9

TABLE 7: Table of all previously undiscovered bugs found by WINFUZZ during all fuzzing experiments.

7. Discussion

In addition to efficiency and correctness, one of the requirements for our design was portability. Our current implementation is Windows-specific, but our approach applies to other operating systems and architectures.

7.1. Supporting Other Operating Systems

In addition to language-level primitives, implementing target-embedded snapshotting requires support for code injection and function hooking. These features exist on other major operating systems. On Linux, the `LD_PRELOAD` environment variable could be used with a shared object to inject target-embedded snapshotting code into a process and hook functions. The C standard library's `setjmp()`/`longjmp()` functions would be used for non-local jumping in place of `GetThreadContext()`/`SetThreadContext()`, and registers would be saved and restored using the same assembly instructions we use on Windows. Thus, we expect similar performance improvements on Linux as we see compared to Winnie.

7.2. Supporting Other Architectures

Only small parts of WINFUZZ involve assembly level code. In addition to the assembly described in § 5.2, we use a single line of assembly in each heap memory hook to check the origin of the call:

```
1 mov [stack_pointer], ebp
```

These code snippets have equivalents on non-x86 architectures. For example, the ARM version looks like:

```
1 mov [stack_pointer], sp
```

7.3. Addressing Edge Cases

Our experiments on a diverse set of binaries show that WINFUZZ offers a significant and scalable performance improvement over the other fuzzers we evaluated. These experiments also demonstrate support for the most common state semantics seen in programs, but there are some edge cases that we did not encounter in our testing, which we leave for future work.

Modifying heap-allocated parameters created outside the scope of the target function will result in incorrect behavior. The contents of these pointers are not restored between executions, which was not a problem for our evaluation, because we only targeted the `main()` function for all of our harnesses, meaning that anything outside the target

function's scope was part of the CRT, and would not usually be modified by the program. To support fuzzing library functions with harnesses, we would need to add support for restoring heap allocations made outside the target function, since modification of heap-allocated parameters would be more common in library functions. This could be done efficiently using an exception handler approach like the one we used to track changes to global memory.

Calls to `VirtualAlloc()` may not allocate new memory, and instead reserve some portion of the process' virtual address space. Future calls to `VirtualAlloc()` may then allocate memory for pages in that reserved region. We did not encounter this behavior in any of the binaries we tested, and did not implement proper support for this feature. Instead, we assume calls to `VirtualAlloc()` always reserve and allocate memory at the same time.

We did not encounter any binaries that used `HeapCreate()/HeapDestroy()`, so we do not clean up custom heaps created by a process. `HeapAlloc()` and `HeapFree()` are supported.

7.4. Harnessing Deeper than main()

To highlight WINFUZZ's ability to address the state correctness issues that often prevent persistent mode from working without a specialized harness, all of the benchmarks in our corpus were fuzzed using a harness that does nothing more than expose the address of the `main()` function to the fuzzer as the target function for fuzzing. However, WINFUZZ is still capable of using other functions as entry points for fuzzing. For example, the shared library version of the FLAC library contains an API for parsing files in the FLAC format. A harness for this binary would load the DLL with `LoadLibrary()`, and then use `GetProcAddress()` to find pointers to any functions it needs to use (e.g, `FLAC__stream_decoder_new()`, `FLAC__stream_decoder_init_file()`). It would then use these saved function pointers in its own function that invokes the FLAC library and parses the input, which would be exposed to the fuzzer in the same way our current harnesses expose the `main()` function.

8. Related Work

WINFUZZ builds upon previous advancements aimed at making fuzzing more effective for Windows programs and more general techniques for increasing fuzzing performance.

8.1. Windows Application Fuzzing

While Linux is home to many industrial and academic fuzzing efforts, the Windows fuzzing ecosystem is much smaller—even though the application ecosystem is much larger. Attempts to fuzz Windows applications on non-Windows kernels (e.g., on Linux via WINE [47]) are far too error-prone to be reliably used, leaving *on-Windows* fuzzing the only realistic option for Windows software fuzzing. To this end, WinAFL [18] remains today's most popular Windows fuzzer, bringing many successful features from the original Linux-based AFL [15] to the Windows environment. TinyAFL [37] and Jackalope [38] leverage the power of Google's fast TinyInst [48] instrumentation. Winnie accelerates WinAFL [19] with forkserver execution via reverse-engineered copy-on-write cloning primitives. Many emerging Windows fuzzing efforts are extending traditionally Linux-only fuzzing enhancements to the WinAFL platform. WinAFLFast incorporates the mutation scheduling of AFLFast [49], while WinAFL-PowerMopt adds that of MOPT [50]; WinAFL-IntelPT integrates Intel-Processor-Trace-based code coverage tracing seen in Linux-based fuzzers such as PTrix [51] and PTfuzz [52]; and NetAFL [44] adds support for stateful networking applications like that in AFLNet [53]. As fast test case execution is critical to fuzzing, target-embedded snapshotting can strengthen current and future Windows fuzzers.

8.2. Orthogonal Fuzzing Optimizations

Many prior efforts accelerate fuzzing with optimizations to its test case execution and code coverage tracing steps. UnTracer [7], Zeror [10], and HeXcite [11] introduce code coverage oracles to restrict coverage tracing to only coverage-increasing test cases, eliminating the overhead of tracing redundant-coverage test cases. InsTrim [8] uses control-flow analysis to intelligently minimize the set of program locations where coverage-tracing is performed. RIFF [9] and CollAFL [54] rewrite coverage-tracing instrumentation to use fewer overall instructions. RetroWrite [15], StochFuzz [55], and ZAFL [33] develop fast static binary rewriting to accelerate coverage tracing in binary-only contexts. As these efforts all increase fuzzing speed, we believe that combining them with target-embedded snapshotting will yield synergistic improvements in fuzzing performance.

9. Conclusion

WINFUZZ is the fastest and most effective fuzzer for Windows applications. WINFUZZ interposes on program binaries to enable fast and correct in-memory looping by tracking and resetting program state changes. Pushing the responsibility of program state cleanup between test cases down to the application level, fast and effective OS-invariant fuzzing is possible. Our evaluation with real-world Windows binaries shows that, compared to the other state-of-the-art grey-box Windows fuzzers, WINFUZZ achieves up to 32% better code coverage and 72x better performance on average across ten benchmarks, while maintaining compatibility with current and future versions of Windows.

Acknowledgments

Special thanks to our shepherd and all the reviewers for their help in making this work the best it could be. This material is based upon work supported by the National Science Foundation under Grant No. 2046589.

References

[1] C. Lemieux and K. Sen, "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage," in *ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE, 2018.

[2] C. Aschermann, P. Jauernig, T. Frassetto, A.-R. Sadeghi, T. Holz, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Network and Distributed System Security Symposium*, ser. NDSS, 2019.

[3] P. Srivastava and M. Payer, "Gramatron: effective grammar-aware fuzzing," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2021.

[4] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer, "Igor: Crash Deduplication Through Root-Cause Clustering," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2021.

[5] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic Crash Bucketing," in *ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE, 2018.

[6] X. Zhang, J. Chen, C. Feng, R. Li, W. Diao, K. Zhang, J. Lei, and C. Tang, "DeFault: mutual information-based crash triage for massive crashes," in *International Conference on Software Engineering*, ser. ICSE, May 2022.

[7] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2019.

[8] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing," in *NDSS Workshop on Binary Analysis Research*, ser. BAR, 2018.

[9] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, and J. Sun, "RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing," in *USENIX Annual Technical Conference*, ser. ATC, 2021.

[10] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling," in *IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2020.

[11] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2021.

[12] M. Zalewski, "Fuzzing random programs without execve()," 2014. [Online]. Available: http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html

[13] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2017.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies*, ser. WOOT, 2020.

[15] M. Zalewski and Google, "AFL," 2020. [Online]. Available: https://github.com/google/AFL

[16] K. Zhao, S. Gong, and P. Fonseca, "On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications," in *European Conference on Computer Systems*, ser. EuroSys, 2021.

[17] A. Fioraldi and N. Gregory, "AFL-Snapshot-LKM," 2020. [Online]. Available: https://github.com/AFLplusplus/AFL-Snapshot-LKM

[18] Google Project Zero, "WinAFL," 2016. [Online]. Available: https://github.com/googleprojectzero/winafl

[19] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning," in *Network and Distributed System Security Symposium*, ser. NDSS, 2021.

[20] J. Jung and S. Tong, "Winnie-AFL," 2021. [Online]. Available: https://github.com/sslab-gatech/winnie

[21] M. Geelnard, "Benchmarking OS primitives." [Online]. Available: https://www.bitsnbites.eu/benchmarking-os-primitives/

[22] Microsoft, "Using setjmp and longjmp," 2022. [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/using-setjmp-longjmp?view=msvc-170

[23] ——, "Creating Guard Pages," 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/memory/creating-guard-pages

[24] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[25] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing Structure while Fuzzing," in *USENIX Security Symposium*, ser. USENIX, 2019.

[26] A. A. Ebrahim, M. Hazhirpasand, O. Nierstrasz, and M. Ghafari, "FuzzingDriver: the Missing Dictionary to Increase Code Coverage in Fuzzers," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER, 2022.

[27] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *IEEE Cybersecurity Development Conference*, ser. SecDev, 2016.

[28] Google, "ClusterFuzz," 2018. [Online]. Available: https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md

[29] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference*, ser. ATC, 2005.

[30] 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 Conference on Programming Language Design and Implementation*, ser. PLDI, 2005.

[31] M. Heuse, "AFL-DynamoRIO," 2018. [Online]. Available: https://github.com/vanhauser-thc/afl-dynamorio

[32] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2020.

[33] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing," in *USENIX Security Symposium*, ser. USENIX, 2021.

[34] M. Heuse, "AFL-Dyninst," 2018. [Online]. Available: https://github.com/vanhauser-thc/afl-dyninst

[35] M. Shudrak, "Manul," 2020. [Online]. Available: https://github.com/mxmssh/manul

[36] GRIMM, "KillerBeez," 2018. [Online]. Available: https://github.com/grimm-co/killerbeez

[37] L. H. Q. Linh, "TinyAFL," 2022. [Online]. Available: https://github.com/linhlhq/TinyAFL

[38] Google Project Zero, "Jackalope," 2020. [Online]. Available: https://github.com/googleprojectzero/Jackalope

[39] Microsoft, "CreateProcessA function (processthreadsapi.h)," 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

[40] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," in *USENIX Security Symposium*, ser. USENIX, 2017.

[41] R. Johnson, "WinAFL-IntelPT," 2018. [Online]. Available: https://github.com/intelpt/winafl-intelpt

[42] M. Böhme, "WinAFL Fast," 2018. [Online]. Available: https://github.com/mboehme/winaflfast

[43] H. Shah, "WinAFL PowerMopt: WinAFL with mopt mutators and AFLFast power schedulers," 2018. [Online]. Available: https://github.com/hardik05/winafl-powermopt

[44] M. Shudrak, "netafl: WinAFL patch to enable network-based apps fuzzing," 2018. [Online]. Available: https://github.com/intelpt/winafl-intelpt

[45] Microsoft, "Windows 10 release information," 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows/release-health/release-information

[46] G. Hunt and D. Brubacher, "Detours: Binary interception of win32 functions," 1999. [Online]. Available: https://www.microsoft.com/en-us/research/publication/detours-binary-interception-of-win32-functions/

[47] Fioraldi, Andrea, "Fuzz with Wine Demo," 2019. [Online]. Available: https://github.com/AFLplusplus/Fuzz-With-Wine-Demo

[48] Google Project Zero, "TinyInst," 2022. [Online]. Available: https://github.com/googleprojectzero/TinyInst

[49] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing As Markov Chain," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2016.

[50] C. Lv, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimize Mutation Scheduling for Fuzzers," in *USENIX Security Symposium*, ser. USENIX, 2019.

[51] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary," in *ACM ASIA Conference on Computer and Communications Security*, ser. ASIACCS, 2019, arXiv: 1905.10499.

[52] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided Fuzzing with Processor Trace Feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[53] V. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, ser. ICST, 2020.

[54] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *IEEE Symposium on Security and Privacy*, ser. Oakland, 2018.

[55] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting," in *IEEE Symposium on Security and Privacy*, ser. Oakland, May 2021.