# EFFICIENT WINDOWS APPLICATION FUZZING WITH FORK-SERVER

A Dissertation
Presented to
The Academic Faculty

By

Stephen Tong

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in Computer Science in the
School of Computer Science
College of Computing

Georgia Institute of Technology

December 2020

**EFFICIENT WINDOWS APPLICATION FUZZING WITH FORK-SERVER**

Thesis committee:

Dr. Taesoo Kim
School of Computer Science
*Georgia Institute of Technology*

Dr. Mustaque Ahamad
School of Computer Science
*Georgia Institute of Technology*

Date approved: December 3, 2020

## ACKNOWLEDGEMENTS

Although I am just an undergraduate student, I am truly grateful to everyone I have met during my research.

First, I must thank Prof. Taesoo Kim for everything he has done during my studies. He was the perfect advisor for me. I first met Prof. Kim was before I enrolled at the university, while touring the campus. I found him at his office in Klaus, but I didn't realize how busy he is until later. Prof. Kim still took the time to show me, a prospective student, around the lab and introduce me. I felt that he saw potential in me, and that experience shaped how I felt about college.

I am grateful for all of the opportunities Prof. Kim has provided to me. He introduced me to the world of academia. I always felt included in the group and that I was treated like any other student despite being an undergrad. Whenever we chatted, he always provided helpful perspectives and insightful advice. Overall, Prof. Kim left a huge impact on my time at the university: in research, in lecture, and outside the lab. I hope that I can be as kind to others as he has been to me.

I am also grateful to my collaborators, especially Jinho Jung and Prof. Hong Hu. Jinho showed me the ropes for performing research, and Hong taught me all of the tricks of the trade in academia. Working with them, I felt like I went from a total beginner to a seasoned veteran. Jinho's diligence and dedication kept us focused, even when we were discouraged. Hong's sense of humor and witty remarks were always there to cheer us up. I felt that together we could tackle any obstacle in our way. I'm glad that I had the opportunity to learn from and work with them.

Thanks to my fellow researchers for their patience, kindness, and respect: Yechan Bae, Insu Yun, Mansour Alharthi, Dr. Daehee Jang, Prof. Sanidhya Kashyap, Dr. Kevin Koo, Ammar Askar, Fan Sang, Seulbae Kim, Hanqing Zhao, Jungwon Lim, Yonghwi Jin, Sujin Park, Wen Xu, Soyeon Park, Ren Ding, Meng Xu, Yu-Fu Fu, and Dr. Steffen Maass. Thank

you everyone for everything you taught me. Also, I'm sorry for falling asleep during the group meetings!

Thank you to my faculty readers, Dr. Taesoo Kim and Dr. Mustaque Ahamad.

Thanks to Transfer Learning for always being there and for making college such a wonderful and exciting journey. I can't describe how grateful I am to have met everyone.

Lastly, thanks to my mom, dad, and brother for their unconditional support.

## TABLE OF CONTENTS

## SUMMARY

Fuzzing is an effective technique for automatically uncovering bugs in software. Since it was introduced, it has found thousands of vulnerabilities. Nowadays, fuzzing is an indispensable tool in security researchers' arsenal.

Unfortunately, most fuzzing research has been concentrated on Linux systems, and Windows fuzzing has been largely neglected by the fuzzing community. Windows systems still represent a large market share of desktop computers, and as they are end-user systems, they are valuable targets to attackers. Windows fuzzing is still difficult-to-setup, slow, and generally troublesome. There exists a chicken-egg problem: because Windows fuzzing is challenging, little effort is invested in it; yet, because little effort is invested, Windows fuzzing remains challenging. We aim to break this cycle by attacking one of the root problems blocking easy and effective Windows fuzzing.

A key difference between Linux and Windows systems for fuzzing is the lack of a fork() functionality on Windows systems. Without a suitable fork() API, a fuzzer cannot quickly and reliably clone processes, an operation that fuzzing relies heavily upon. Existing Windows fuzzers such as WinAFL rely on *persistent-mode fuzzing* as a work-around for the lack of fast process cloning, unlike Linux fuzzers which rely on a *fork-server*.

In this work, we developed a fork() implementation that provides the necessary fast process cloning machinery and built a working fork-server on top of it. We integrated this fork-server into WinAFL, and applied several other key improvements and insights to bypass the difficulties of fuzzing typical Windows applications. In our evaluation, we ran our fuzzer against 59 fuzzing harnesses for 37 applications, and found 61 new bugs. Comparing the performance of our fork() implementation against other similar APIs on Windows, we found that our implementation was the most suitable and efficient. We believe that this marks the first Windows fork implementation suitable for fuzzing.

# CHAPTER 1
# INTRODUCTION

Software vulnerabilities are a leading cause of computer misuse and exploitation. Vulnerabilities are usually caused by simple programming errors or mistakes, called bugs. In the past few years, the activity of locating bugs, or bug hunting, has become a high-profile and even lucrative endeavor for computer security professionals. Both the defensive and offensive computer security communities place great emphasis on bug hunting due to the potential of new vulnerabilities to enable and unleash serious cyber-attacks. If an attacker is able to find a previously-unknown bug and weaponize it, millions of computer systems could be compromised. Likewise, if a defensive security analyst is able to locate and patch the bug before attackers can exploit it, such an attack would be thwarted. Therefore, the ability to quickly and efficiently locate bugs is paramount to computer security.

Fuzzing, a technique to automatically find software bugs, has been the focus of a significant body of recent research. When fuzzing a piece of computer software, random data is fed into the program as input, and the behavior of the program is closely monitored for errors or instabilities, such as program crashes or infinite loops ("hangs"). Although the overall idea dates back to the 1990s [1], fuzzing received increased attention following the release of the fuzzer American Fuzzy Lop (AFL) in 2013 [2]. Due in large part to AFL's practical success in discovering new bugs in well-known, widely-deployed software like Adobe Flash and Mozilla Firefox, fuzzing thus became a widespread technique employed by many security researchers.

AFL's revolutionary success is due to its use of *coverage feedback-guided greybox fuzzing*. In this mode of fuzzing, inputs that elicit new, previously-unseen behavior from the program are prioritized for further mutation in future iterations. Thus, the fuzzer "learns" to generate meaningful inputs over time that are more likely to trigger software bugs, rather

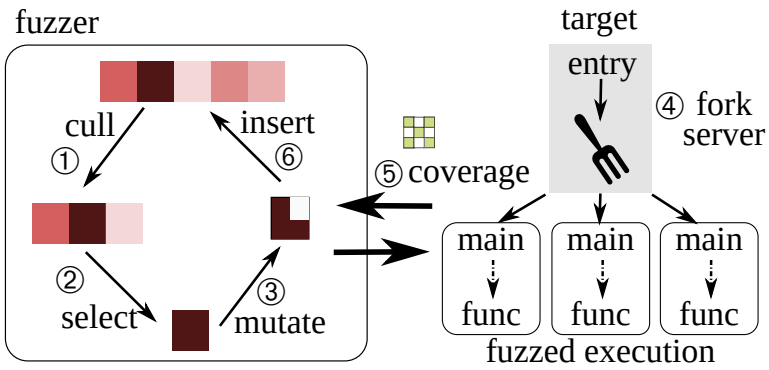


**Figure 1.1: An overview of the popular fuzzer AFL.** (1) The fuzzer maintains a queue of inputs. Each cycle, (2) it picks one input from the queue and (3) modifies it to generate a new input. (4) It feeds the new input into the fuzzed program and (5) records the code coverage. (6) If the execution triggers more coverage, the new input is added back into the queue. Figure reproduced from [8].

than random noise that is easily rejected and does not significantly test the program's logic. Several other fuzzers have been developed that also adopt AFL's principle of coverage-guided fuzzing. These projects aim to improve on AFL's shortcomings and include LLVM's LibFuzzer [3], Google's honggFuzz [4], and several others. Nevertheless, AFL's legacy remains strong in the research community today: a large amount of research still bases their work off of AFL as a starting point [5, 6, 7].

Unfortunately, because AFL's overwhelming popularity has dominated the research community, most recent research has focused on fuzzing Linux software. AFL was designed to target programs built for Linux systems [2], and Windows software is not compatible with Linux. Thus, Windows software, which represents a large portion of the consumer and desktop software market, has been mostly neglected by the fuzzing community. The current state-of-the-art for Windows fuzzing research is WinAFL, a port of AFL to Windows systems [9]. However, WinAFL suffers from several problems, including slow execution speeds, poor stability, and inaccurate instrumentation for coverage feedback. Moreover, since most commercial Windows software is closed-source, security researchers must expend tedious efforts reverse-engineering the software before they are able to fuzz it.

In this thesis, we aim to address these shortcomings by: ❶ introducing a new user-space fork() mechanism to facilitate fast and reliable fuzzing, ❷ developing new fuzzing

techniques to cope with the challenges of fuzzing complex, real-world software, and ❸ implementing coverage feedback with "full-speed" instrumentation.

This work was published as part of "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning" in NDSS'21 [8].

# CHAPTER 2

# RELATED WORK

Fuzzing, a technique which automatically finds software bugs by testing random inputs against programs, was first introduced in 1990 by Miller et al [1]. However, the technique was not popularized until the release of American Fuzzy Lop (AFL) by Michal Zalewski in 2015 [2]. Since the release of AFL, many researchers have expanded the body of research surrounding fuzzing. Many other fuzzers have been developed, such as LLVM's LibFuzzer [3] and Google's honggfuzz [4]. In general, the goal of a fuzzer is to find as many bugs as possible, as quickly as possible. However, today's fuzzers still miss many bugs. The large majority of fuzzing research attempts to address this in three main ways: ① improving the performance of fuzzers, ② applying fuzzing in a new and interesting fashion, and ③ fuzzing previously-untested target applications.

## 2.1 Improving fuzzer performance

The first category of fuzzing research, improving fuzzer performance, has received the most attention of the three. This category can be broken down into two main sub-categories: optimizing the fuzzer's raw performance and optimizing the fuzzer's input generation strategy. One technique developed to improve AFL's performance is full-speed coverage [10]. Under full-speed coverage, basic blocks are instrumented only once, and new code coverage is only reported when a new basic block is encountered. This is a drastic shift from past research which emphasized edge coverage and per-run block coverage. Full-speed fuzzing argues that the trade-off in execution speed over coverage granularity is economical. Another example of research aimed at improving fuzzer performance is AFLFast [11], which extends AFL using power schedules to improve its search strategy.

### 2.1.1 Improving input generation

The task of optimizing fuzzer input generation has received more attention from the research community than improving fuzzers' raw performance. The major breakthroughs in this area of research so far have been concolic or hybrid fuzzing [12], machine learning-guided fuzzing [13, 14, 15], multi-dimensional fuzzing [16, 17], and grammar-based fuzzing [18, 19, 20]. Hybrid fuzzing seeks to blend dynamic coverage information that traditional grey-box fuzzers like AFL rely on with white-box information gleaned from symbolic execution. These concolic (concrete and symbolic) fuzzers are thus able to solve branches and conditionals that would otherwise stump unequipped grey-box fuzzers. Machine learning-assisted fuzzing seeks to improve the fuzzer's input generation by using neural networks. Some approaches try to improve the mutation selection strategy [15], while others try to generate more meaningful inputs by training neural networks to recognize which inputs are interesting and which are not [13, 14]. Multi-dimensional fuzzing targets complex applications, like filesystems, by expanding the definition of "input" past files to also include holistic information about the execution environment such as API call sequences and thread scheduling order [16, 17]. Grammar-based fuzzing is a well-known fuzzing technique which generates inputs using grammar specifications to guarantee that inputs are well-formed [19]. They are typically used to tackle the challenge of fuzzing language parsers, such as C compilers [18] or Javascript interpreters [20].

## 2.2 Novel applications of fuzzing

The second main research direction, applying fuzzing in new ways or to new domains, is arguably the most diverse of the three categories. Researchers have noted that fuzzing can be modeled as a state-space exploration problem. Following this line of reasoning, some researchers have even used AFL to play Super Mario [21]. By doing so, they showed that fuzzing can have novel and interesting applications not just in the narrow realm of

computer security. Google created ClusterFuzz [22], an attempt to massively scale AFL up by leveraging Google's enormous compute power. ClusterFuzz backs the OSS-Fuzz project which helps find bugs in open-source software [23, 24]. Running AFL at such large scales presents its own set of interesting challenges and rewards. Lastly, Fuzzcoin [25] is a new exciting project that aims to match Cluster-fuzz in computing power parity by crowdsourcing computing power to commodity hardware owned by ordinary consumers.

### 2.3 Exploring new fuzzing targets

The last main category of fuzzing research aims to bring fuzzing to new targets that have not been thoroughly fuzz tested before. One shining example of research in this direction is Syzkaller [7]. While AFL is designed to fuzz Linux user-mode applications, Syzkaller was the first fuzzer to prove that kernel fuzzing is viable. It fuzzed the kernel by making random Linux system calls, in the hopes of triggering a crash or hang. Since Syzkaller, kAFL [6] has also tried to address the kernel fuzzing problem. kAFL improves on Syzkaller by improving the fuzzer's coverage feedback. Using Intel PT instrumentation, they improved the fuzzer by leveraging innovative hardware features. Meanwhile, FuzzGen [26] and FUDGE [27] aim to fuzz new targets by generating fuzzing harnesses based on static analysis of code which uses the fuzzed libraries. Lastly, WinAFL [9] attempts to address Windows applications, whereas previous research had focused on Linux applications.

Extending fuzzing to new targets is important because it allows security researchers to test and correct bugs in code that has never been fuzz tested before. Fuzzing is a phenomenally successful technique that has found bugs in virtually every code-base that it has been applied against. Fuzzing has even found bugs in formally-verified code [17]. Until code has been fuzz tested, it is overwhelmingly likely that there are bugs or edge-cases that the programmer has forgotten to consider, which would have otherwise been quickly rooted out by fuzzing.

This research aims to bring fuzzing to closed-source Windows binaries, many of which

have *never* been fuzz tested before. These applications represent a vast portion of the consumer desktop application market and have millions of users. For example, since Adobe Photoshop is closed-source, it is likely that no one has ever fuzzed it because closed-source Windows applications are very difficult to fuzz. Our aim is to bring fuzzing to popular commercial applications like Photoshop and eliminate the "low-hanging fruit" that attackers would exploit to compromise end-user systems.

# CHAPTER 3

# TOWARDS A PRACTICAL WINDOWS FUZZER

Windows closed-source applications pose two major challenges for fuzzing. First, they are difficult to instrument. Second, many of them behave problematically—for example: complex initialization code, self-termination, and file handle leaks all hinder fuzzing. To mitigate these problems, we propose a new Windows fuzzer that uses an injected fuzzing agent. Our fuzzer consists of two main components: the *fuzzing engine*, and the injected *fuzzing agent*. The fuzzing engine is responsible for processing code coverage information, updating coverage maps, and generating new inputs via mutation. The fuzzing agent performs the low-level work required to collect code coverage and also deals with various runtime issues that may arise in the target application, which we elaborate on below. To coordinate and communicate with the fuzzing engine, the agent uses a bidirectional pipe. This design enables the fuzzing agent to perform its work directly within the target application while also neatly separating the fuzzers' mutation and instrumentation functionality.

To implement our fuzzer, we build on top of WinAFL, a port of AFL for Windows systems. We offer three key improvements over WinAFL, which we expand on below: ❶ we introduce a new implementation of fork() suitable for high-speed Windows application fuzzing (§3.1); ❷ we provide fuzzing techniques that overcome the challenges of fuzzing real-world, commercial software (§3.2); and ❸ we adopt modern instrumentation methods, sidestepping hurdles caused by existing methods (§3.3). These improvements combined significantly improve the applicability, practicality, and performance of Windows fuzzing. In the following sections, we will discuss each of these contributions in further detail.

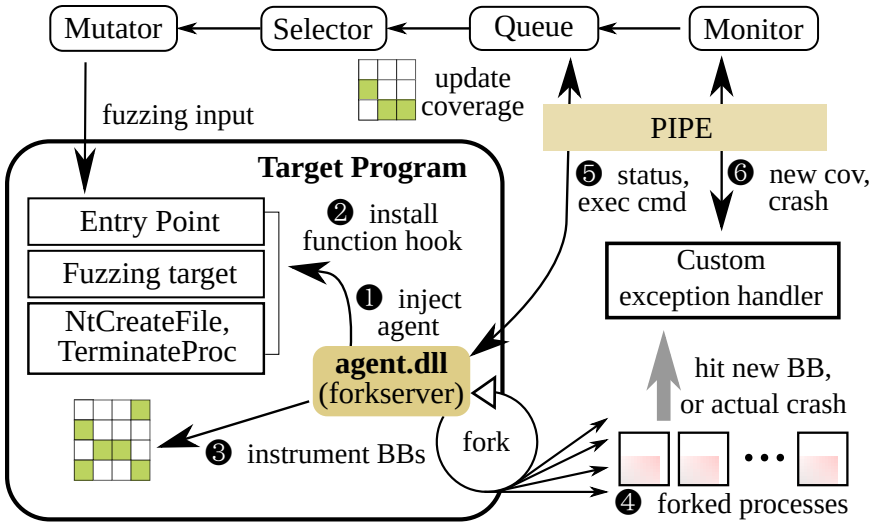


**Figure 3.1: Overview of our fuzzer.** We inject a fuzzing agent into the target. The injected agent spawns the fork-server, instruments basic blocks, and hooks several functions. This improves performance and sidesteps various instrumentation issues. Figure reproduced from [8].

## 3.1 Fork on Windows

Contemporary fuzzers such as AFL [2] adopt a *fork-server* architecture, which is extremely useful for fuzzing. When fuzzing under a fork-server, the fuzzed application runs normally until right before the input is read and processed. At this point, the application enters the *fork-server*, which spawns pre-initialized processes on-demand. Each forked child process executes a single input, and the fuzzer records the execution's outcome. The benefit of using a fork-server is two-fold: first, it improves performance by avoiding costly re-executions, and second, it improves stability by isolating the effects of an execution to a single process. Without a fork server, the fuzzer wastes a significant amount of time on irrelevant initialization code, as the program must be re-executed from scratch for each input.

One solution to avoid slow re-executions is to use *persistent-mode fuzzing* [9, 28, 29], in which multiple inputs are executed in the same fully-initialized process. However, this harms stability: unless the fuzzing target function is perfectly pure (i.e., without side-effects), differences in the program state will gradually accumulate across many executions, eventually leading to divergence. However, for Windows applications, most target functions

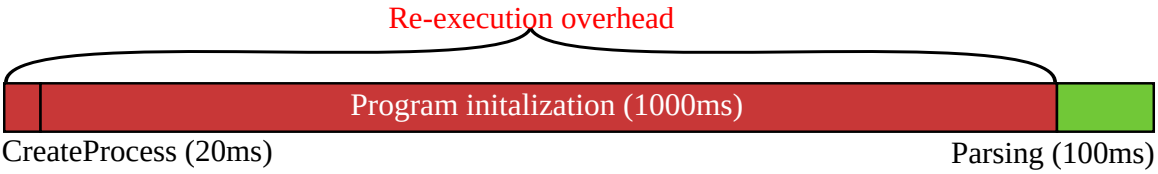


**Figure 3.2: Execution timeline for a complex Windows application.** The startup and program initialization often dominate execution times of Windows applications when fuzzing. Before reaching any parsing logic, an application must first run uninteresting initialization code, including GUI setup.

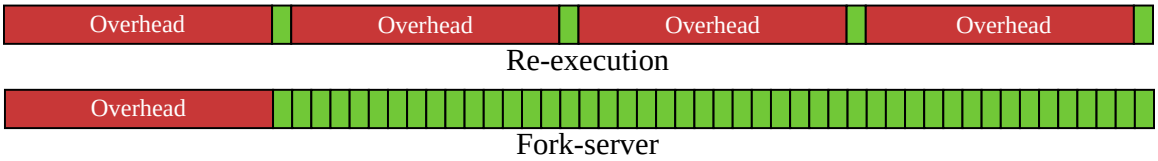


**Figure 3.3: Cost of re-execution when fuzzing.** Effective fuzzing campaigns require thousands or millions of repeated executions. As shown, frequent re-executions severely degrade fuzzing performance. Hence, it is crucial to minimize re-executions by the fuzzer.

have side-effects. For example, an application may handle errors by simply terminating itself. Thus, reliance on persistent mode severely limits the applicability of fuzzing. Meanwhile, using a fork-server avoids this problem altogether. Because each input is executed in a separate process, any possible side-effects that can obstruct fuzzing are safely contained, such as timeouts, crashes, and hangs. This greatly improves the stability and scalability of fuzzing. Nevertheless, existing Windows fuzzers cannot use a fork-server. Overall, having a fork-server is extremely beneficial to fuzzing, but depends on the existence of a fork() API.

Unfortunately, the Windows kernel does not expose a fork() API suitable for fuzzing. Thus, this work sets out to implement such process cloning machinery to aid Windows fuzzing. To clone a process, all data structures and memory owned by the process must be duplicated, including page contents, page tables, file descriptor tables, etc. It is possible to crudely approximate this behavior *manually* [30]; however, this approach has serious flaws. Not only is manual process cloning unreliable, it is also slow. We will elaborate on both these issues below.

First, manual process cloning is unreliable. The kernel maintains more information about processes than is accessible from user space: for example, references to kernel objects. It would not be possible to faithfully recreate these aspects of the program state, leading

to incorrect behavior or corruption. In general, operating systems are designed so that the kernel is the one responsible for loading and running processes; manually cloning processes would require making many assumptions about the cloned process. However, we cannot afford to make assumptions about our fuzzed application: recall that our goal is to fuzz *commercial, off-the-shelf* Windows software. Thus, to expand the applicability of fuzzing, we need a process cloning mechanism that is native to the Windows kernel.

Second, manual process cloning is slow. Modern operating systems that expose fork() do so using a technique called *Copy-on-Write* (CoW) [31]. When a process is cloned as CoW, only the bare-bones data structures holding process metadata are copied, such as page tables and process list entries. The actual full memory contents of the process are not copied when fork() is called. Instead, both processes share the same memory pages until one of them writes to a page. Thus, only the pages that are modified by the forked child process are copied, greatly improving performance. Since successful fuzzing campaigns involve millions of executions, a high-speed fork implementation is crucial to the scalability of fuzzing. For a heavy application (for example, with memory footprint >50MB), a manual process cloning method would have to copy all process memory each execution, seriously degrading performance. Thus, we need a fork implementation that is also Copy-on-Write.

In short, we need a fork implementation that is both native to the Windows kernel and also Copy-on-Write (CoW). To create our fork() implementation, we reverse-engineered several undocumented Windows APIs and subsystems. We extracted several key functionalities that are required to create a stable fork() implementation suitable for Windows fuzzing. Namely, we analyzed the function `CreateUserProcess` in ntdll and the CSRSS (Client/Server Runtime Subsystem) and found several key magic values that they require. This API is an undocumented, but first-party API that is exposed directly by the kernel to ordinary user programs, and it accurately clones processes. To implement the CoW fork functionality, we call `NtCreateUserProcess` with a NULL section handle argument. This satisfies both of the requirements outlined above.

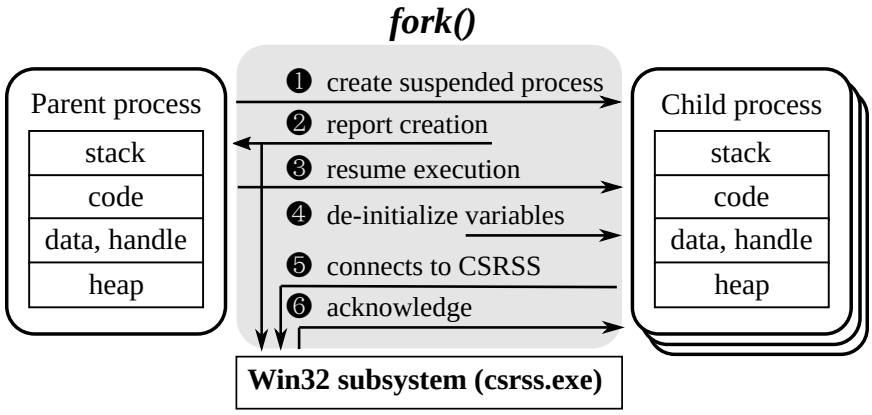


**Figure 3.4: Overview of fork() on Windows.** We analyzed various Windows APIs and services to achieve a CoW fork() functionality suitable for fuzzing. Fixing up the CSRSS is essential for fuzzing COTS Windows applications: if the CSRSS is not re-initialized, the child process will crash when accessing Win32 APIs. Figure reproduced from [8].

Next, whenever we fork a new child process, we must connect it to the CSRSS so that it may function properly. The CSRSS is a user-mode daemon which manages several underlying Windows components, such as console Windows. Newly-created processes must be connected to the CSRSS to work properly. Otherwise, operations like opening or saving a file may lead to a crash. Connecting to the CSRSS is normally done by Windows automatically, but we must do this ourselves. We begin by calling `CsrClientCallServer` in the parent process with message `BasepCreateProcess`. Next, because the child process was forked from a fully-initialized parent process, several variables in the child's address space in ntdll, such as `CsrServerApiRoutine`, must be *de-initialized*. Lastly, the child process calls `CsrClientConnectToServer`. These steps connect a newly-forked child process to the CSRSS and allow it to behave normally.

We tested the fork implementation for the following properties: ① robustness, ② speed, and ③ copy-on-write. To test the fork implementation's robustness, we created several simple programs that perform basic operations such as file I/O, printing console output, etc. We then forked these programs before the behavior and validated that they worked correctly in the child process. We also tested that global state was preserved properly during forking, by checking the value of a global counter variable that we incremented before

forking each time. To test the fork implementation's speed, we measured how quickly the fork implementation could run in executions per second. Lastly, to verify that the fork implementation was indeed Copy-on-Write (CoW), we compared the time taken to fork a process with large and small memory footprints and ensured they were similar.

### 3.2 Fuzzing Commercial Windows Applications

Next, we will describe how our fuzzer mitigates problematic behaviors common in commercial, off-the-shelf (COTS) Windows applications. To fuzz an application, the fuzzer needs to be able to easily call the target functionality repeatedly and automatically. For example, to fuzz a PDF reader, the fuzzer needs to be able to call the reader's PDF parsing functions in the fuzzing loop. For command-line (CLI) applications, this is simple: the fuzzer can simply re-run the program each time, passing the fuzzing input file as an argument or via standard input. However, many popular Windows applications only expose GUIs (graphical user interfaces), and it is much more difficult to fuzz GUI applications as they cannot be easily automated. Although it is possible to directly simulate keyboard and mouse inputs [32], this is slow and not scalable. A good fuzzer should run at speeds of at least 100 execs/sec, but using automation tools would limit speeds to less than 3 exec/sec. Thus, we need a way to bypass the GUIs obstructing the functionality we wish to fuzz.

One popular method to fuzz GUI applications is to create a *fuzzing harness*. A fuzzing harness essentially converts a GUI application into a CLI application and acts like an adaptor. The harness exposes a convenient command-line interface directly to the functionality we wish to fuzz. However, creating fuzzing harnesses is a challenging problem even for first-party, open-source software [27, 26]. For us, the situation is even more dire: we wish to fuzz *third-party, commercial* software. It is very difficult to create accurate fuzzing harnesses for COTS Windows software, and most of them are not designed to be fuzzed. We will now discuss the specific reasons as to why harness generation is challenging.

The ultimate source of the difficulty is that the problem of accurately extracting code

fragments from binary applications is fundamentally hard. The harness generation problem at its core boils down to a code extraction problem: the fuzzing harness must selectively extract necessary driver code for calling the target functionality, while excluding irrelevant GUI code. For example, Windows applications often have a lot of initialization and setup code, which must all be faithfully reproduced in a fuzzing harness, making generating valid harnesses difficult. Another serious obstacle for harness synthesis is the existence of call-back functions [33, 34]. Call-back functions are interface bindings provided to the fuzzed code that the harness must also reimplement accurately. Overall, the underlying code extraction problem is not just difficult; in general, it is *undecidable*. Notwithstanding theoretical concerns, *in practice* most simple cases can still be solved heuristically. Therefore, to expand the scope of fuzzing to commercial, off-the-shelf Windows applications, we need a way to simplify and reduce the harness generation problem to minimize the amount of code that must be extracted.

We propose a technique, the *injected fork-server*, that obviates complex harness synthesis. Rather than trying to completely extract all of the setup or call-back behavior, we simply run the application binary itself. The fuzzing agent is injected as soon as the program loads, before any application code has begun executing. Once injected, the fuzzing agent first hooks a function specified by the harness, and promptly returns control to the target application. Then, the target application is allowed to initialize itself. Once the hook is called, the application is halted and the fuzzing agent spins up the fork-server. In our experience, this technique resolves many problems caused by difficult-to-extract setup code. Meanwhile, since we spin up the fork-server only at some point deep within the program, this massively improves performance because the initialization code only runs once.

Windows applications also exhibit a myriad of miscellaneous problems that impede fuzzing efforts. These problems typically manifest during harness generation. Our fuzzer employs several strategies to mitigate these issues:

**Surviving Process Termination.** Many applications implement error handling by simply

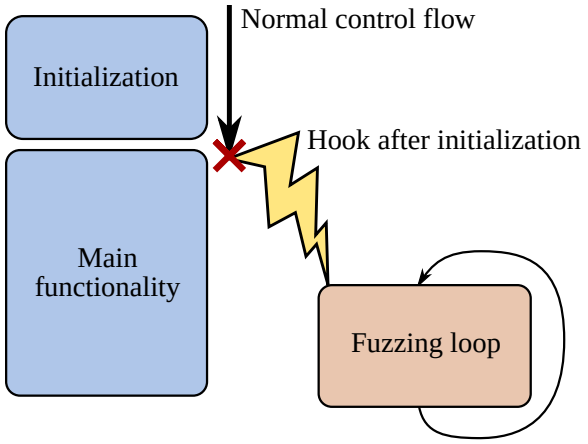


**Figure 3.5: Injected fork-server technique.** The target program is started normally and allowed to self-initialize. We place a hook immediately after the initialization code, before the program begins its main functionality (e.g., a GUI). We then hijack and redirect the control flow to our fuzzing loop.

terminating the program. Because most inputs generated during fuzzing are invalid, this would demand constant re-executions from scratch, severely degrading performance. Our fuzzer sidesteps this problem, since we can readily spawn pre-initialized child processes thanks to our new implementation of fork() (see §3.1).

**Sharing the input file.** Since the fuzzer must be able to overwrite the input file for each iteration, the input file must be opened non-exclusively. To resolve the issue, we hook the function `NtCreateFile`, a sink for file-related operations. By hooking the function, we can check whether the file being opened is the input file, and if so, add the write-sharing flag to the parameters before resuming. This solves the common issue of the target application locking the file, blocking the fuzzer.

**Resolving Self-unpacking Code.** Our fuzzer supports binaries which employ self-unpacking code as an anti-reverse-engineering tactic. We do so by employing *guard pages*, which are similar to memory breakpoints. By setting the target function's page protection to be inaccessible, our exception handler is notified whenever it is accessed by the target application. We do not install our instrumentation hooks until the target application begins executing that page; e.g., after it has finished unpacking itself. This deals with self-unpacking code in an elegant and target-agnostic fashion.

### 3.3 Improved Instrumentation

Effective fuzzing relies on the availability of practical, reliable program instrumentation to profile the fuzzed program's runtime execution. Existing Windows instrumentation solutions and fuzzers suffer from serious stability and performance penalties. DynamoRIO suffers from reliability issues and is prone to crashes when fuzzing with WinAFL [35, 36]. WinAFL relies on the debug API. Using the debug API frequently causes errors, since many applications do not behave normally when they detect they are under a debugger. This is especially true when the target application uses software protection mechanisms (e.g., third-party packers or obfuscators [37, 38]). Generally speaking, these external solutions are often unwieldy and unreliable. Instead, we propose an *internal* solution by injecting the fuzzing functionality directly into the target program. We integrate the fork-server, binary instrumentation, and exception handling code into one library, the *fuzzing agent*, which is forcibly loaded into the target application at startup. Because it can perform its work directly inside the target application's process address space, the fuzzing agent avoids using any debug APIs and can instrument the binary more effectively.

To collect coverage, we use *fullspeed fuzzing* [10] and a custom exception handler in place of the problematic debug API. Fullspeed fuzzing collects boolean basic block coverage, meaning that new coverage is only reported when new basic blocks are reached. To be precise, each basic block receives a single-shot breakpoint, which yields control to the fuzzing agent when reached. The fuzzing agent then records coverage information before removing the breakpoint. Thus, the breakpoint is excluded from any future runs. When the basic block is reached during subsequent executions, it is un-instrumented and execution proceeds uninterrupted. Thus, since new coverage is rare, the target application runs nearly at native speed during fuzzing. Moreover, breakpoints need only be installed *once* thanks to the fork-server: child processes inherit the same set of breakpoints as the parent. Since applications may easily contain as many as 100,000 basic blocks, this is a

crucial optimization as instrumenting all of the basic blocks takes a considerable amount of initialization time.

# CHAPTER 4

# EVALUATION

To evaluate our fuzzer, we conducted end-to-end experiments on real-world applications as well as individual experiments on our Windows fork implementation. The end-to-end experiments were run on Intel Xeon E5-2670 v3 CPUs, and the fork implementation tests we run on an Intel i7-7700 CPU.

In our end-to-end evaluation [8], we constructed 59 fuzzing harnesses for 37 applications. We were able to fuzz all of these harnesses using our fuzzer, and we found 61 vulnerabilities from 32 binaries across 16 applications. WinAFL in Intel PT mode failed to run 33 out of the 59 harnesses; WinAFL in DynamoRIO mode failed to run 30 of the 59 harnesses. We chose 6 harnesses that WinAFL was able to run for an in-depth side-by-side comparison. In 4 of those 6 harnesses, we noticed major issues such as memory or handle leaks, or unacceptably slow performance (i.e., $< 1.0$ executions/second). On those 6 harnesses, our fuzzer improved raw fuzzing speed by 31.3x and coverage by 4.0x on average. Thus, our fuzzer significantly expands the scope, efficacy, and practicality of Windows fuzzing.

We compared several process spawning and cloning APIs against our new fork() implementation. The closest comparison in terms of functionality would be Cygwin's fork() implementation; nevertheless, we also included `CreateProcess`, the Windows Subsystem on Linux (WSL)'s fork implementation, and finally Linux's native fork.

We compared our fork implementation against a similar mechanism from the Cygwin project which also provides process cloning functionality. In terms of speed, our fork implementation achieved speeds of around 300 forked processes per second, whereas Cygwin achieved only 72.8. Unlike Cygwin, our implementation is Copy-on-Write (CoW), and also directly supported by the kernel. These factors contribute to our implementation's improved performance. As discussed in §3.1, manual fork implementations like Cygwin's

are undesirable for fuzzing purposes. In short, manual process cloning methods scale poorly as they are not CoW. They are also less versatile because they are not directly supported by the operating system and can only perform user-mode operations. Hence, in summary, our fork implementation outperforms Cygwin's in terms of both speed and versatility.

We compared our fork implementation against process spawning APIs. `CreateProcess` is the standard Windows function used to spawn new processes with a default program state. In terms of speed, `CreateProcess` achieves speeds of roughly 100 new processes per second. This is slower than our fork implementation because the operating system must allocate, prepare, and initialize an entire new process for each `CreateProcess` call. Considering the complicated steps required to properly load a Windows executable (dynamic linking, resolving relocations, etc.), this incurs a heavy overhead. On the other hand, forking a pre-initialized process with CoW is simple: all that must be copied are auxiliary data structures such as page tables and file descriptor lists. Unlike APIs that *spawn* new processes such as `CreateProcess`, our fork implementation *clones* existing processes. As discussed in §3.1, this difference is paramount for fuzzing. The ability to quickly clone pre-initialized processes enables fuzzers to cleanly and efficiently re-execute the target functionality across many different inputs. Thus, not only does our fork implementation outperform creating new processes from scratch, it also provides essential functionality for Windows fuzzing.

We compared our fork implementation against the one used by the Windows Subsystem for Linux (WSL). The Windows Subsystem for Linux is a syscall translation layer that allows Windows systems to run Linux binaries directly, roughly the opposite effect of projects like WINE [39, 40]. As a POSIX implementation, WSL features a fully-functional fork implementation. WSL's fork outperforms ours in terms of raw speed, achieving about 400 forked processes per second, whereas ours achieved only 300. Internally, WSL leverages similar kernel functions as our fork implementation to direct the kernel to clone a user-space process [41], leading to roughly comparable speeds. However, WSL achieves better speeds because it can make more assumptions about the processes to be forked; namely,

they are special WSL processes, which are like a subclass of general Windows processes. Nevertheless, WSL's fork implementation is incompatible with Windows fuzzing. WSL is designed only to run native Linux ELF binaries, whereas our goal is to fuzz typical Windows PE binaries. Overall, our fork implementation enjoys speeds comparable to WSL's fork implementation but supports native Windows applications instead.

Lastly, we compared our fork implementation against Linux's native fork implementation. For process creation, Linux greatly outperforms Windows in terms of raw speed: Linux's fork implementation could prepare 5,000 forked processes per second, whereas ours could only prepare 300. There is little we can do about this problem. The Linux kernel hosts a completely different operating system and is architected differently from the Windows NT kernel. It could be that the Linux process creation process is simpler or more efficient than its counterpart on Windows, or both. In any case, although it may be more efficient to fuzz cross-platform applications using their Linux version, many popular applications are limited to Windows platforms only. Thus, albeit imperfect, our fork implementation fills the gap on Windows systems and fulfills the need for a Windows fork implementation suitable for fuzzing purposes.

Our fork implementation has a few idiosyncrasies that must be taken into account, primarily due to the design of the Windows operating system. Similar to its Linux counterpart, if a multi-threaded program calls fork(), only the calling thread is cloned. Also similar to Linux, any handles (similar to file descriptors) a process has open when forking are not inherited by the child process by default. However, we can sidestep this issue by manually enumerating and marking all handles as inheritable before calling fork().

# CHAPTER 5
# CONCLUSION

Fuzzing is a proven technique for uncovering software bugs that has enjoyed success and attention in both industry and academia. Unfortunately, most existing fuzzing efforts are centered around Linux systems, and can therefore only fuzz applications that support Linux platforms. Current fuzzing efforts are also heavily focused on open-source projects. As a result, many popular Windows applications like Adobe Photoshop or Steam have not been thoroughly fuzzed. Thus, there are likely many "low-hanging fruit" vulnerabilities that are waiting to be exploited, that otherwise would have been eliminated through fuzzing.

Fuzzing Windows applications is difficult. There exists a negative feedback loop, in that Windows applications are un-fuzzed because Windows fuzzing is difficult; meanwhile, Windows fuzzing is difficult because there has been little effort invested in it. The fundamental difficulty comes from two primary reasons: ① the closed-source Windows ecosystem prevalent with GUI applications, and ② the lack of a fork-like API for efficiently cloning processes. These two halves can be thought as two complementary problems: the harness synthesis problem and the fuzzer implementation problem [8]. This work aims to tackle the second problem: the lack of an effective and versatile Windows fuzzer implementation.

In this thesis, we proposed the following: ❶ we introduced a new implementation of fork() suitable for high-speed Windows application fuzzing; ❷ we provided fuzzing techniques that can overcome the challenges of fuzzing real-world, commercial software; and ❸ we implemented new instrumentation methods, sidestepping hurdles caused by existing methods. We evaluated our fuzzer on real-world programs and found that our fork implementation sidesteps many common fuzzing obstacles. Furthermore, comparisons with other implementations showed that our fork implementation is competitive and well-suited to Windows fuzzing.

## REFERENCES


[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] M. Zalewski, *American fuzzy lop*, http://lcamtuf.coredump.cx/afl/, 2015.

[3] K. Serebryany, "Libfuzzer–a library for coverage-guided fuzz testing," *LLVM project*, 2015.

[4] Google, *Honggfuzz*, https://github.com/google/honggfuzz, 2010.

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

[6] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "KAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[7] Google, *Syzkaller: an unsupervised, coverage-guided kernel fuzzer*, https://github.com/google/syzkaller, 2018.

[8] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning (to appear)," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, Feb. 2021.

[9] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, *AFL For Fuzzing Windows Binaries*, https://github.com/ivanfratric/winafl, 2016.

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

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

[12] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Aug. 2020.

[13] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "Neufuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36 340–36 352, 2019.

[14] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[15] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops (SPW)*, IEEE, 2018, pp. 116–122.

[16] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[17] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework (to appear)," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[18] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, Jun. 2011.

[19] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[20] W. Syndder and M. Shaver, "Building and breaking the browser," *Black Hat USA Briefings (Black Hat USA)*, 2007.

[21] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1597–1612.

[22] Google, *Fuzzing for Security*, https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

[23] ——, *OSS-Fuzz - continuous fuzzing of open source software*, https://github.com/google/oss-fuzz, 2016.

[24] O. Chang, A. Arya, K. Serebryany, and J. Armour, *OSS-Fuzz: Five months later, and rewarding projects*, https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017.

[25] D. Jang and A. Askar, "FuzzCoin: A Digital Currency with Fuzzing as a Proof-of-Work," 2020.

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

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

[28] M. Zalewski, *New In AFL: Persistent Mode*, https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html, 2015.

[29] A. Souchet, I. Fratric, J. Vazquez, and S. Denbow, *How to Select A Target Function*, https://github.com/googleprojectzero/winafl#how-to-select-a-target-function, 2016.

[30] *Highlights of Cygwin Functionality*, https://cygwin.com/cygwin-ug-net/highlights.html, 1996.

[31] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A fork() in the road," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ACM, 2019, pp. 14–22.

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

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

[34] R. Schaefer, *Fuzzing Adobe Reader For Exploitable Vulns*, https://kciredor.com/fuzzing-adobe-reader-for-exploitable-vulns-fun-not-profit.html, 2018.

[35] *WinAFL issue tracker #125*, https://github.com/googleprojectzero/winafl/issues/125, 2018.

[36] *WinAFL issue tracker #172*, https://github.com/googleprojectzero/winafl/issues/172, 2019.

[37] Oreans Technology, *Themida: Windows software protection system*, https://www.oreans.com/themida.php, 2020.

[38] VMProtect Software, *VMProtect Software Protection*, https://vmpsoft.com/, 2020.

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

[40] WineHQ, *Wine Project*, https://www.winehq.org/, 2020.

[41] Microsoft, *WSL architectural overview*, https://docs.microsoft.com/en-us/previous-versions/windows/desktop/cmdline/wsl-architectural-overview, 2019.