Hindawi
Security and Communication Networks
Volume 2022, Article ID 4911587, 13 pages
https://doi.org/10.1155/2022/4911587

Research Article

SpotFuzzer: Static Instrument and Fuzzing Windows COTs

Yeming Gu, Hui Shu, Rongkuan Ma, Lin Yan, and Lei Zhu

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China

Correspondence should be addressed to Hui Shu; shuhui123@126.com

Received 25 January 2022; Accepted 10 August 2022; Published 30 August 2022

Academic Editor: Luigi Coppolino

The security research on Windows has received little attention in the academic circle. Most of the new methods are usually designed for the Linux system and are difficult to transplant to Windows. Fuzzing for Windows programs always suffers from its closed source. Therefore, we need to find an appropriate way to achieve feedback from Windows programs. To our knowledge, there are no stable and scalable static instrumentation tools for Windows yet, and dynamic tools, such as DynamoRIO, have been criticized for their performance. To make matters worse, dynamic instrumentation tools have very limited usage scenarios and are impotent for many system services or large commercial software. In this paper, we proposed SpotInstr, a novel static tool for instrumenting Windows binaries. It is lightweight and can instrument most Windows PE programs in a very short time. At the same time, SpotInstr provides a set of filters, which can be used to select instrumentation points or restrict the target regions. Based on these filters, we propose a novel selective instrumentation method which can speed up both instrumentation and fuzzing. After that, we design a system called SpotFuzzer, which leverages the ability of SpotInstr and can fuzz most Windows binaries. We tested SpotInstr and SpotFuzzer in multiple dimensions to show their superior performance and stability.

1. Introduction

Security research and software analysis technologies on Windows cannot match its market share. The focus of academic research remains on UNIX-like platforms. One of the main reasons is that most applications software on Windows are closed source, which requires more effort for researchers to do a lot of reverse engineering. There is no doubt that Windows is the most widely used operating system. We should pay more attention to its software security.

Vulnerabilities are the main threat to system security. Security researchers use static analysis [1] or dynamic analysis to locate vulnerabilities in software. In our experience, one of the most popular dynamic methods for vulnerability mining is Fuzzing [2]. Especially since AFL [3] appeared in 2013, fuzzing has made great progress. We can find fuzzing tools for file parser [4], system kernel [5], net protocol [6], or IoT devices [7]. The feedback technology introduced by AFL is still the most effective way to find vulnerabilities. Over the years, there are a lot of AFL-like tools [8–10] developed for different scenarios. The key idea of the feedback technology is leveraging the instrumentation technology to trace the execution path. The default instrumentation mode of AFL is to patch the compiler and insert some code snippet into the target. This compile-time instrumentation has minimal side effects on the target, so it is the preferred choice for AFL. To cope with the closed source software, AFL also supports the QEMU [11] mode, which uses a virtual machine to dynamically trace the execution path of the target. After 3 years of waiting, the Windows version of AFL was finally released in 2016. WinAFL [12] made a lot of changes to adapt to Windows. It uses DynamoRIO [13] to instrument target dynamically instead of QEMU mode, and drops the compile way. Finally, WinAFL implements roughly the same feedback capabilities as AFL.

Although many practical problems have been solved in the field of fuzzing, there are still many shortcomings. AFL and its successors can only be used for Linux platform. WinAFL was designed for Windows, but it uses DynamoRIO, which makes the target much slower and its applicability is limited. DynamoRIO can only run some simple programs with acceptable overhead. If we want fuzzing

COTs on Windows, we need to come up with a new approach to overcome these problems.

In this paper, we have designed a new fuzzing system for Windows. The system relies on static instrumentation against Windows binaries. The key idea of instrumentation is to extract memory points by reverse analysis, and instrument the target at these points using binary rewriting technologies. We find that existing tools always pursue high rate of basic block coverage and instrument as more as they can. According to our experience, most of the regions in a program are vulnerable free. Therefore, it is not a good idea to instrumenting everywhere in a program, which leads to higher overhead for analysis, instrumentation, and execution. We propose a novel method to filter the points of instrumentation, which can make static instrumentation more efficient, lightweight, and robust.

We developed SpotInstr as our static instrumentation tool, which can be divided into two parts: the analysis front-end for extracting memory points and the binary rewriting back-end for instrumenting the target. The analysis front-end was designed as a plugin for IDA Pro [14]. It leverages the advantage of IDA Pro's disassembly capabilities and uses its interfaces to analyze the target binary. We have done extensive work to understand the Intel instructions [15] to extract the basic blocks in the assembly code. We also implemented a set of interfaces for filtering the memory points. The back-end is based on PeLib [16]. We have few choices for PE file manipulation libraries. After some research, we finally found the PeLib, an old and no longer maintained Library. PeLib is not well developed and still has many bugs in it. So, we made many patches to make it work properly. In the process of instrumenting, we found that both the analysis and instrumentation phases took a lot of time when working on large binaries. We made a lot of optimizations on the algorithms in both stages and achieved significant performance improvements.

We developed SpotFuzzer based on SpotInstr. The most obvious improvement of SpotFuzzer is that it uses a new architecture for fuzzing running processes on Windows. We find that some programs on Windows cannot start directly or always depend on another program, so ordinary fuzzer cannot fuzz these targets directly. SpotFuzzer uses an agent to inject into the target process and builds a belt within the target and the fuzzer.

We demonstrate the applicability of SpotInstr and SpotFuzzer by instrumenting and fuzzing more than 20 COTS software or Windows components. First, we compared SpotInstr with pe-afl [17] and syzygy [18]. Then, we compared SpotFuzzer with pe-afl and WinAFL. In conclusion, our instrumentation tool runs dramatically fast. Compared to pe-afl, the average time cost of our tool is about 90% lower and the compatibility is better. Thanks to SpotInstr, our fuzzing tool also has better performance than pe-afl and WinAFL, and it can find more vulnerabilities in less time.

This paper makes the following contributions:

First, we developed an instrumentation tool for Windows binaries, called SpotInstr, which supports almost all Windows PE files, and offers a significant performance improvement over the state of art tools.

Then, we designed a new selective instrumentation technique that focuses on memory-related vulnerabilities. This technique reduces the number of instrumentations significantly and makes the target execution speed close to the original one, while having the better vulnerability discovery capabilities.

We also propose a new fuzzing architecture for Windows components, called SpotFuzzer, which leverages the capabilities of SpotInstr. SpotFuzzer makes the fuzzing Windows COTs much easier and offers better performance than the popular WinAFL.

2. Motivation

2.1. Instrument Binary-Only Programs on Windows. Most commercial Windows software are binary-only programs, security researchers must instrument these programs before fuzzing them. Dynamic instrumentation is an effective way to analyze the binary file, and it is widely used in the modern fuzzers. WinAFL [12] leverages DynamoRIO [13] to fetch feedback during execution. WINNIE [19] relies on Intel Pin [20] for dynamic binary instrumentation. However, the dynamic instrumentation introduces additional runtime overhead.

But at present, there is no stable static instrumentation tools for Windows yet. The main challenge for static instrumentation is how to rewrite a PE file correctly and quickly. There are some studies dedicated to improving binary rewriting. However, the recent works such as e9patch [21] and RetroWrite [22] are all designed for Linux.

2.2. Focus on Memory Issues. Fuzzing with sanitizers is the most effective way to find memory-related vulnerabilities. When fuzzing on Linux, there are several sanitizers to use to detect memory issues, like AddressSanitizer, MemorySanitizer, and LeakSanitizer. ParmeSan [23] is a sanitizer-guided fuzzer, which greatly reduces the time-to-exposure (TTE) of real-world bugs. However, one must use the compiler to recompile the target with source code.

Another way to speed up detecting memory issues is called Selective Instrumentation. According to our experience, when we compile a simple program, the compiler will generate hundreds of functions and thousands of basic blocks, while the actual main function only contains several lines. This indicates that there always be a lot of non-functional codes in target binary. If we want to fuzzing a target, we would better skip these codes or focus on memory-related areas to save instrumentation time and approve fuzzing performance.

3. Design

As mentioned in Section 2.2, there is a big gap between theory and practice for static instrumentation for Windows binaries. We cannot find any static instrumentation tool for PE64, and the on-hand tools for PE32 is just too old to fit

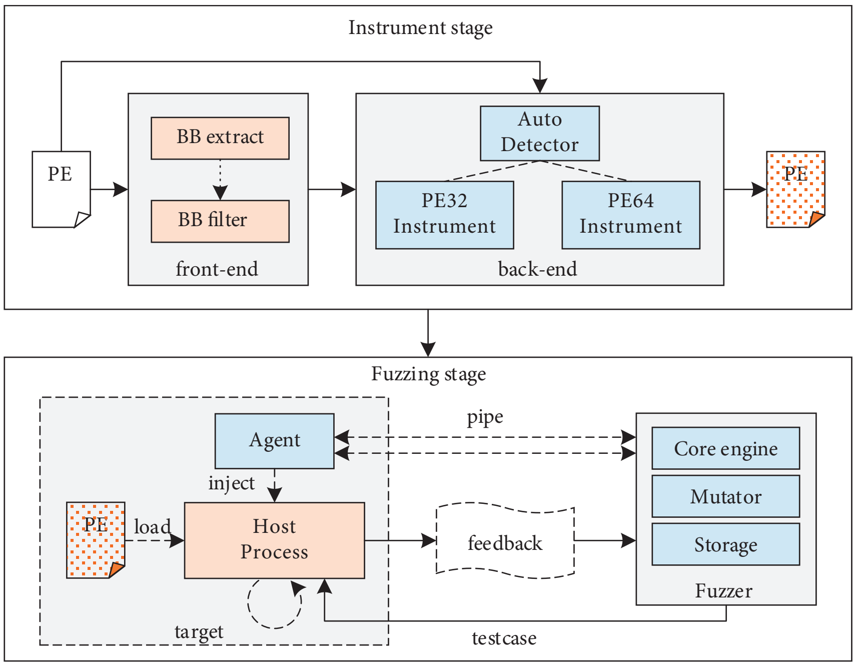


Figure 1: Top view of SpotFuzzer.

new PE format. By the way, we need implement a set of interfaces for user to control where to instrument. So, we design SpotInstr as a lightweight, robust, and efficient static instrumentation tool for both PE32 and PE64.

When the target binary is finish instrumented, we use SpotFuzzer to load and test it with the feedback technology. For some Windows service-related target, we design an agent-based fuzzing framework. All these efforts we make are aiming to the goal: fuzzing Windows binaries with static instrumentation easily, scalable, and efficiently.

3.1. System Overview. Our fuzzing workflow contains two stages: instrumentation and fuzzing. When a target is chosen for fuzzing, the font-end of SpotInstr should be used in the instrumentation stage to analyze and extract memory points, and the back-end of SpotInstr completes the binary instrumentation. In the fuzzing stage, the instrumented binary is used as the fuzzing target for SpotFuzzer. The top view of our system is shown in Figure 1.

3.2. Basic Block Extract. The purpose of static instrumentation is to cooperate with the fuzzer, while SpotFuzzer reads the feedback of execution path from the instrumented binary. So first, we need to find out all basic block in the target binary. The basic block is a sequence of contiguous instructions that contains no jumps or labels, as shown in Figure 2.

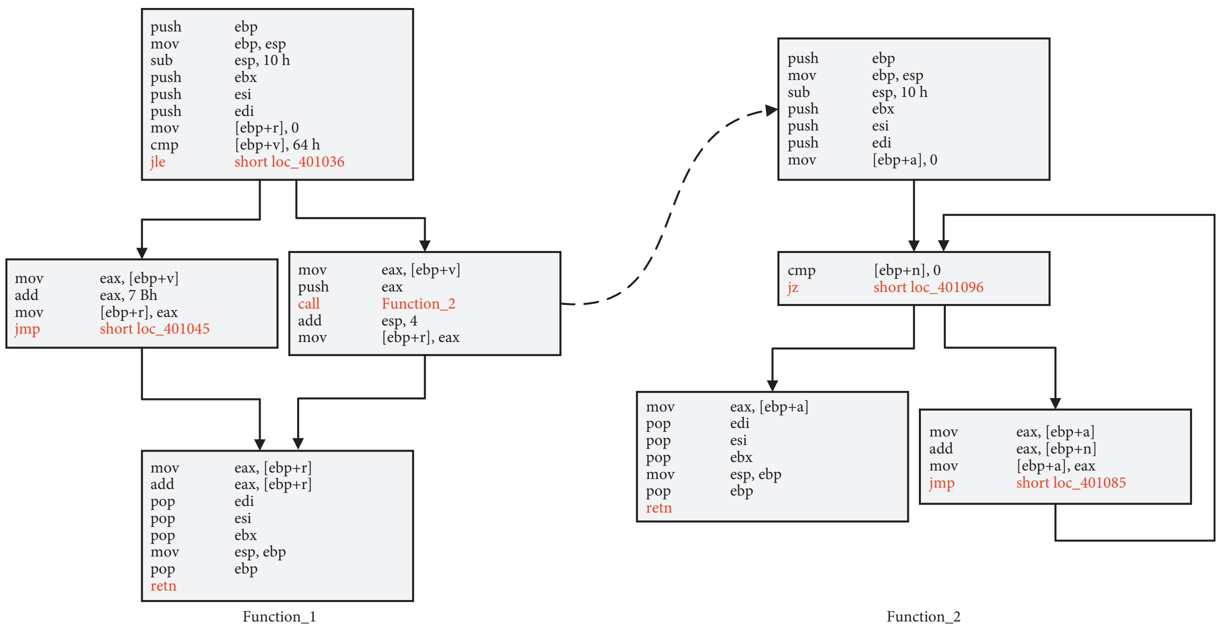
It is easy to observe that the basic block always starts at the function entry, the call destination, the *jmp* destination or the *jcc* destination. And the basic block always ends with a *jmp* instruction, a *jcc* instruction, a ret instruction or ends before next basic block head. To extract all basic block's head, we should analyze all assembly codes in the text segment. The key point is to find all control flow transfer instructions, which include *call*, *jmp*, *jcc*, and ret. And calculate out the destination address according to the operand value.

3.2.1. CALL Instruction. The call instruction has 10 different formats according to Intel's Instruction Set Reference. It is easy to identify such instructions, which always starts with Opcode 0 × E8, 0 × 9A or 0 × FF. In 64-bit mode, we should take care of the REX prefixes in the instruction. It is more complex to calculate the destination address of the call instruction. Different calculation method should be taken for different operant type.

3.2.2. JMP Instruction. The *jmp* instruction has 11 different formats according to Intel's Instruction Set Reference. We can find these instructions with Opcode 0 × EB, 0 × E9, 0 × EA, or 0 × FF. Also, we should consider the REX prefixes in 64-bit mode. The calculation of destination address is similar to the one of call instruction.

3.2.3. JCC Instructions. The *jcc* instructions include a set of conditional jump instructions, which include *ja*, *jb*, and *jc*. There are 95 different formats of opcode according to Intel's Instruction Set Reference. *jcc* instructions always start with one byte [0 × 70~0 × 7F] or two bytes 0 × 0F + [0 × 80~0 × 8F]. The calculation of destination address is similar to the one of call instruction.

3.2.4. Jump Tables. The most difficult situation is the special *jmp* instructions which we call them jump tables. These instructions always use a register as its operand (like *jmp*



```
push    ebp
mov     ebp, esp
sub     esp, 10 h
push    ebx
push    esi
push    edi
mov     [ebp+r], 0
cmp     [ebp+v], 64 h
jle     short loc_401036
```

```
mov     eax, [ebp+v]
add     eax, 7 Bh
mov     [ebp+r], eax
jmp     short loc_401045
```

```
mov     eax, [ebp+v]
push    eax
call    Function_2
add     esp, 4
mov     [ebp+r], eax
```

```
mov     eax, [ebp+r]
add     eax, [ebp+r]
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn
```

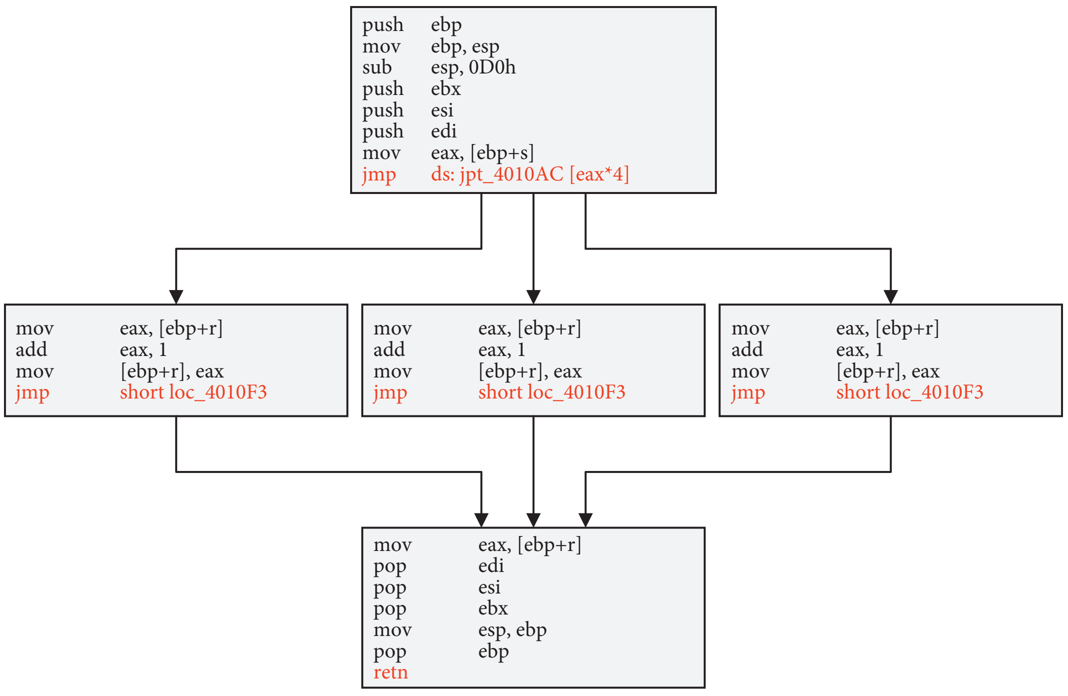
```
push    ebp
mov     ebp, esp
sub     esp, 10 h
push    ebx
push    esi
push    edi
mov     [ebp+a], 0
```

```
cmp     [ebp+n], 0
jz      short loc_401096
```

```
mov     eax, [ebp+a]
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn
```

```
mov     eax, [ebp+a]
add     eax, [ebp+n]
mov     [ebp+a], eax
jmp     short loc_401085
```

Figure 2: Basic blocks in the program.



```
push    ebp
mov     ebp, esp
sub     esp, 0D0h
push    ebx
push    esi
push    edi
mov     eax, [ebp+s]
jmp     ds: jpt_4010AC [eax*4]
```

```
mov     eax, [ebp+r]
add     eax, 1
mov     [ebp+r], eax
jmp     short loc_4010F3
```

```
mov     eax, [ebp+r]
add     eax, 1
mov     [ebp+r], eax
jmp     short loc_4010F3
```

```
mov     eax, [ebp+r]
add     eax, 1
mov     [ebp+r], eax
jmp     short loc_4010F3
```

```
mov     eax, [ebp+r]
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn
```

Figure 3: Jump table in the program.

rcx). It is hard to calculate the destination address, but we can use context before the *jmp* instruction to figure out the position of jump table. With the jump table data, we can then calculate all destination addresses. Figure 3 shows an example of the jump table.

We design two different algorithms to detect jump tables in PE32 and PE64. Then, we can add all destination address in jump tables into the basic block list.

3.3. Instrument Points Filter Interfaces. With all the basic block information, we can take some strategies to filter the basic block. According to observation, there are a lot of nonfunctional code, initializing code, and helper code in target. Most of which have little chance to trigger critical vulnerability. Instrumenting on these codes increases the analysis time and instrumentation time and slows down the execution. Even worse, more instruments also increase the possibility of program errors. In this paper, we provide three interfaces for user to include or exclude some basic blocks.

Address including: The basic block list contains all basic block information contains starting address, instruction size, relative address position, etc., and the list should be sorted by the starting address. So, it is very simple to specify an address or address range to tag as included and then delete all other items which is not tagged.

Address excluding: The basic block whose address specified to be excluded will be deleted from the basic block list. Because the basic block list is sorted by the starting address, the deleting should be very fast.

Function name regular matching: For some cases, we may have the symbol file for the target or just rename a set of functions. Then, we can filter the basic block list by the function names. At this moment, SpotInstr supports using regular expression to include or exclude functions, basic blocks in which will be included or excluded.

3.4. Static Binary Rewriting for PE File. In this paper, we support both trampoline and inline mode to instrument code snips. The trampoline technology to realize static binary rewriting, which means a 5-byte jump instruction will replace origin codes and redirect the control flow to a trampoline. This technology has obvious advantages: simpler, faster, more stable, more reliable, and lightweight. Figure 4 shows the PE file structures for non-instrumented and instrumented binaries.

PE structure (32 bit or 64 bit) auto detect: the SpotInstr back-end supports both PE32 and PE64, as shown in Figure 5, and there is no need for user intervention. In order to realize this detector, we build a PE file parser for both PE32 and PE64, with which the SpotInstr can recognize the PE structure before doing any instrument. This work greatly improves the usability of the tool.

3.4.1. Building the Trampoline Segment. The trampoline segment is used to store all the trampoline code snippets. According to our implementation, each memory point should have its own trampoline code snippet. The size of this segment should be calculated according to account of memory points and the flag of this segment should be set to EXECUTE_READ.

3.4.2. Building the Feedback Segment. The feedback segment is used to store feedback data (e.g., execution path bitmap), which will be used by the fuzzer. In this paper, we inherit the feedback data structure from WinAFL. In addition to this, the feedback segment also holds a size field which indicates the size of the extra feedback segment. We use the extra feedback segment for records linear basic block coverage information when user turns it on. The linear basic block coverage information can be used for lighthouse in IDA Pro.

3.4.3. Building the Local Storage Segment. The local storage segment or TLS segment is used to isolate storage between threads. That means each thread will maintain a TLS segment for local data storage. We use this segment to hold the last basic block address and the jump back address for resume the origin control flow. So, even if the target is multi-thread program, the execution paths for each thread will not be confused.

3.4.4. Updating the Relocation Table. The relocation table is very important for PE file to calculate the right addresses. The instrumentation moves the original code to trampoline which make the original relocation information is no longer correct. After all memory points have been processed by the SpotInstr, the relocation table should be updated to fix all relative addresses in trampolines. The trampolines locate in the new segment, that means the virtual address may exceed the relocation table. So, the simplest way to correct the relocation table is to add some new entries at the end of the table. The old entries for addresses in replaced instructions must be deleted to avoid relocation breaking the jump instructions. In summary, updating the relocation table should have 2 processing stages: the cleaning stage and the inserting stage.

3.4.5. Updating Global Fields and Checksum. After all the processes above have been finished, we should update some global fields in PE header, such as BaseRelocRva and BaseRelocSize. Before updating the checksum, the old one must be reset to 0.

3.5. Fuzzing Framework with the Static Instrument. The latest version of WinAFL supports instrumenting a binary via syzygy statically, but syzygy only provides a framework able to decompose PE32 binaries with full PDB. That is useless for most COTS software, even the Windows components rarely have a private symbols file. So, we have to abandon syzygy and replaced it with our SpotInstr.

3.5.1. General Fuzzing. If the target binary can be loaded normally, SpotFuzzer will use the general fuzzing framework. First, SpotInstr instruments the target binary statically. Then, we use a helper program to load the instrumented module, and it collects and sends feedback to SpotFuzzer. Finally, the fuzz engine generates new test cases based on the feedback. This general fuzzing framework should be suitable for most software. Figure 6 shows the general fuzzing framework.

3.5.2. Agent-Based Fuzzing. If the target is a service on Windows or cannot be loaded normally, SpotFuzzer will inject an agent into the target process. The agent use named pipe to communicate with SpotFuzzer, and once injected it will register an exception handler to catch crash information. Figure 7 shows the agent-based fuzzing framework.

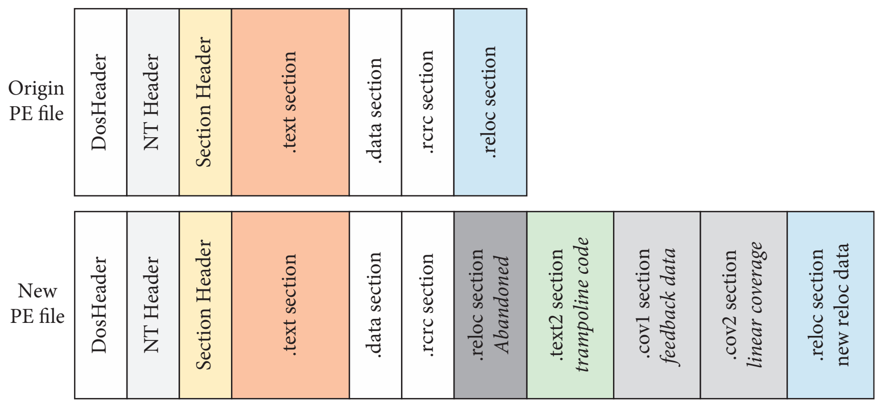


Figure 4: Windows PE file structures.

```
NT header   50  45  00  00  4C  01  05  00  1A  6A  65  60  00  00  00  00
for PE32    00  00  00  00  E0  00  02  01  0B  01  0E  1C  00  32  00  00

NT header   50  45  00  00  64  86  0A  00  4D  81  65  60  00  00  00  00
for PE64    00  00  00  00  F0  00  22  00  0B  02  0E  1C  00  7C  00  00
```

🟦 Machine type
🟥 Image magic

Figure 5: Flags in NT header for PE32 and PE64.

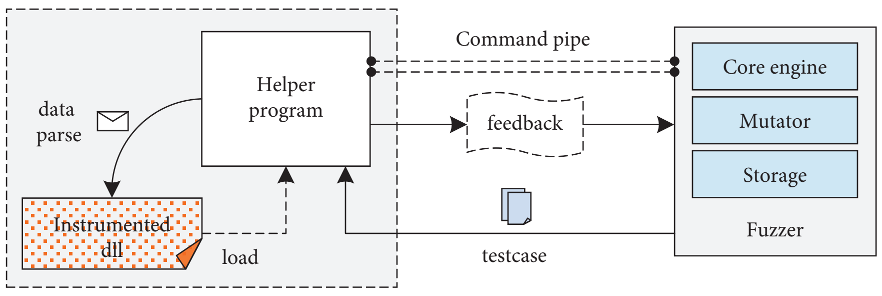


Figure 6: Framework of general fuzzing.

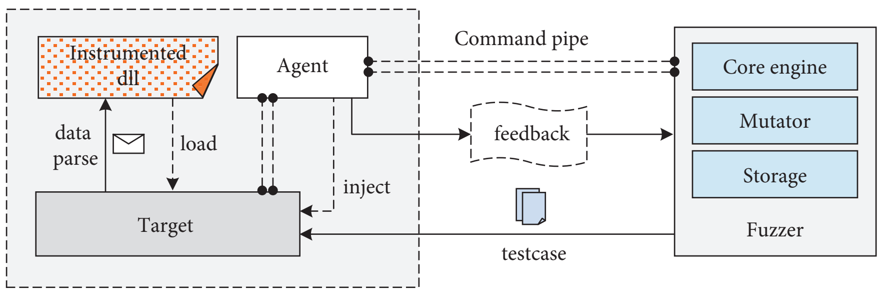


Figure 7: Framework of agent-based fuzzing.

4. Evaluation

In this section, we evaluate scalability of SpotInstr, speed, and overhead compare to pe-afl. We also evaluate performance of SpotFuzzer on some Windows COTS software.

4.1. Instrument Scalability. We evaluate SpotInstr on several widely used software packages on Windows, such as 7z, notepad++, WinRAR, and 010editor. We also choose some system component additionally. Table 1 shows a list of all successfully instrumented binaries on Windows.

Table 1: Binaries list for the scalability test.

Binary	Vender	Architecture	SpotInstr	pe-afl	syzygy
7za.exe	7-Zip	PE32	✓	✓	✗
notepad++.exe	Notepad++	PE32	✓	✗ (crash)	✗
rar.exe	RarLab	PE32	✓	✗ (crash)	✗
010Editor.exe	SweetScape	PE32	✓	✗ (crash)	✗
cmake.exe	CMake	PE32	✓	✓	✗
jscript.dll	Windows	PE32	✓	✓	✗
imagingengine.dll	Windows	PE32	✓	✓	✗
gdi32.dll	Windows	PE32	✓	✓	✗
mpengine.dll	Windows	PE32	✓	✗ (failed)	✗

The results show that our tool can correctly instrument all these executable program or dynamic libraries, while pe-afl can work on a part of them and syzygy can instrument none of them. The main problem for pe-afl is that it is not reliable enough for some real-world programs. Syzygy need private pdb file for the target. That means syzygy supports only targets with source code, one can recompile it and generate the *.pdb* file.

Besides, we also compare some usable features, such as instrument mode, target architecture, thread mode, *.pdb* file dependence, and selective instrumentation. Jump mode is more light weight than inline mode, it will be more stable and efficiency to instrument a huge target with selective Instrumentation. For programs that contain only one parsing thread, single-thread mode can reduce runtime overhead significantly than multi-thread mode. Selective Instrumentation make researchers able to focus on more interesting areas. Table 2 shows the features supported by SpotInstr and other tools.

Table 2: Instrumentation features.

Binary	SpotInstr	pe-afl	syzygy
Inline mode	✓	✓	✓
Jump mode	✓	✗	✗
Support 32 bit	✓	✓	✓
Support 64 bit	✓	✗	✗
Single-thread mode	✓	✓	✗
Multi-thread mode	✓	✓	✓
PDB file independent	✓	✓	✗
Selective instrumentation	✓	✗	✗

4.2. Instrumentation Performance. We try to compare SpotInstr with other tools, such as pe-afl and syzygy. As mentioned before, pe-afl supports a part of PE32 binaries, and syzygy supports only PE32 binaries with private symbols. We can hardly find COTS software that meets the requirements. We have to remove syzygy from the performance evaluation. In order to make the comparison more meaningful, we only choose PE32 binaries which can be both instrumented successfully by SpotInstr and pe-afl. Table 3 shows the binaries chosen for testing. The smallest one is archive.dll with about 176 KB, and the biggest is mpengine.dll with about 11 MB.

To evaluate the instrumentation performance, we design three tests: output size, time cost, and execution overhead. Before testing, we first measure the size of each binary and write a plugin for IDA Pro to calculate the number of basic blocks in each binary. Table 2 also lists the basic block counts of PE binaries chosen for testing. The smallest one named archive.dll, which has less than 9,000 basic blocks. The mpengine.dll is the core engine of Microsoft Malware Protection service, which is the biggest and contains more than 590,000 basic blocks.

First, we compare the count of basic blocks instrumented by SpotInstr and pe-afl. Table 4 shows the count of basic blocks instrumented by different tools. On all test programs, SpotInstr can instrument about 20% more basic blocks with its inline mode than pe-afl. But for jump mode, SpotInstr instruments a little fewer basic block than pe-afl. That is because jump mode supports only a basic block with more than 5 bytes to hold the jump instruction.

Second, we compare the size of the output binaries of SpotInstr and pe-afl. We use SpotInstr and pe-afl to instrument all these programs with their default setting and collect the size of instructed binaries. Figure 8 shows the sizes of programs instrumented by different tools or with different mode. We find that our tool did a little better than pe-afl on most binaries. While working on some small binaries, SpotInstr and pe-afl performed almost the same. We find that different instrumentation modes have a significant impact on the size of the instrumented file. Jump mode always generates smaller output files than inline mode, and the average size reduction is about 10%. Compare to the raw inline mode, if we turn on selective instrumentation, the average size reduction is about 42%. For jump mode, the same selective instrumentation will cause the reduction to 57%.

Third, we compare the efficiency of the tools. Figure 9 shows the time of instrumentation spent by SpotInstr and pe-afl. Obviously, SpotInstr spends much less time than pe-afl in all tests. Pe-afl took 5x~10x more time than SpotInstr on some small binaries, such as archive.dll, 7za.dll, gdi32.dll, and eqnedit32.exe. Pe-afl took 30x~100x more time than SpotInstr on some bigger binaries, such as rar.exe, jscrip.dll, 7za.exe, and cmake.exe. It is worth noting that pe-afl spend more than 1 hour and finally result in an error when instrumenting on mpengine.dll.

At last, we compare the execution time between original programs, static instrumented ones, and dynamic instrumented ones. To figure out the execution overhead caused by instrumenting, we select some typical programs for testing.

Table 3: Binaries list for performance test.

Binary	Size (KB)	Architecture	GUI	Description
archive.dll	176	PE32	No	Library for libarchive
7za.dll	269	PE32	No	Library for 7-zip
gdi32.dll	304	PE32	No	Library for Windows GDI
eqnedit32.exe	524	PE32	Yes	Formula editor used by MS Word
rar.exe	568	PE32	No	Command line tool for WinRAR
jscript.dll	670	PE32	No	Microsoft javascript engine
7za.exe	723	PE32	No	Command line tool for 7-zip
imagingengine.dll	1810	PE32	No	Microsoft image engine
winrar.exe	2433	PE32	Yes	WinRAR GUI program
notepad++.exe	3005	PE32	Yes	Notepad++ GUI program
jscript9.dll	3779	PE32	No	Microsoft javascript engine
cmake.exe	7865	PE32	No	Command line tool for CMake
mpengine.dll	11281	PE32	No	Microsoft malware protection engine

Table 4: Basic blocks instrumented by different tools.

Binary	Basic blocks count				
	pe-afl	SpotInstr	-select	-jump	-jump-select
archive.dll	7660	8652(↑13%)	880	6645	826
7za.dll	9452	11951(↑26%)	2117	8454	1953
gdi32.dll	14003	16781(↑20%)	1202	12569	1154
eqnedit32.exe	10040	14448(↑44%)	947	12255	920
rar.exe	21744	25455(↑17%)	2816	18731	2701
jscript.dll	29444	34988(↑19%)	4964	27421	4856
7za.exe	30946	38326(↑24%)	6531	26892	6137
imagingengine.dll	70224	81110(↑16%)	8738	62914	8479
winrar.exe	54317	64553(↑19%)	7311	48382	6972
notepad++.exe	64147	82095(↑28%)	7594	61113	7214
jscript9.dll	138731	172194(↑24%)	26519	130506	24301
cmake.exe	261180	312399(↑20%)	28683	234899	27510
mpengine.dll	346443	594481(↑72%)	43284	315238	40766

For the convenience of comparison, we try to make the baseline parsing time of input data close to each other. So, we choose the appropriate input data to feed to the instrumented software. In this test, we also add DynamoRIO to show the dynamic instrumentation's overhead. Figure 10 shows the average execution time of 10 runs with different instrumentation type. According to the result, the overhead of the static instrumentation is much less than the dynamic one. Specifically, the average execution overhead of SpotInstr-inline is about 17%, whereas the overhead of pe-afl is about 13%. The main reason for this small gap is that SpotInstr-inline instrument more basic blocks than pe-afl. When we use selective instrumentation, as shown by SpotInstr-inline-select in the picture, the overhead reduces to 2.6%.

Since static instrumentation avoids the translation of instructions, the runtime overhead is significantly lower than dynamic instrumentation. The runtime overhead for static instrumentation depends mainly on the number of instrumented basic blocks. As a result, SpotInstr-inline, which instruments more points, has a slightly higher runtime overhead than pe-alf. However, SpotInstr-inline-select, which instruments fewer basic blocks by selective instrumentation, has a significantly lower runtime overhead.

4.3. Fuzzing Performance. We measured the fuzzing performance from three aspects, including fuzzing speed, execution paths, and unique crashes. The WinAFL was used as our baseline method. The SpotFuzzer leverages SpotInstr to make instrumentation on target program. To look closely at the effect of different instrumentation options on fuzzing, we built SpotFuzzer with different instrumentation modes. As a result, we got four tools, namely SpotFuzzer-inline, SpotFuzzer-inline-select, SpotFuzzer-jump, and SpotFuzzer-jump-select. In which, "inline" and "jump" stand for the instrumentation modes, "select" means the tool uses selective instrumentation.

As the use of WinAFL with syzygy is limited, it cannot work on most COTS software. We use the dynamic mode for WinAFL instead. In order to test WinAFL, we choose 7za.dll as the fuzzing target, which can run correctly under all these fuzzers.

To compare the fuzzing speed between the fuzzers, we observe the number of execution samples within a certain period. Figure 11 shows the total samples tested over time and fuzzing speed for SpotFuzzer and WinAFL. There is no doubt that all the static instrumentation methods have better performance than WinAFL. We can see that target instrumented with inline mode run much faster than jump mode

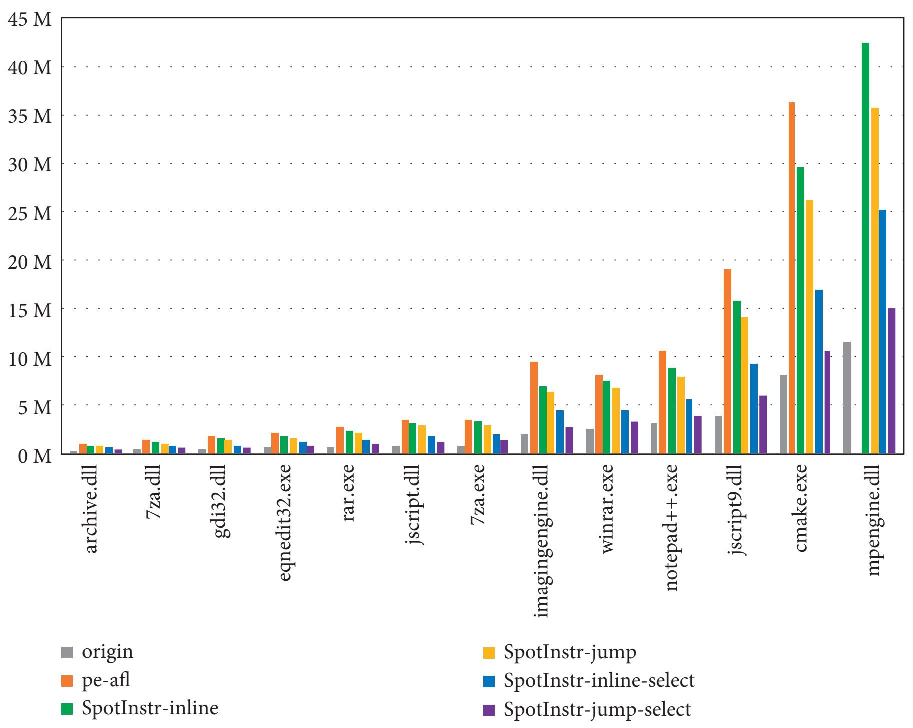


FIGURE 8: Size of original and instrumented binaries.

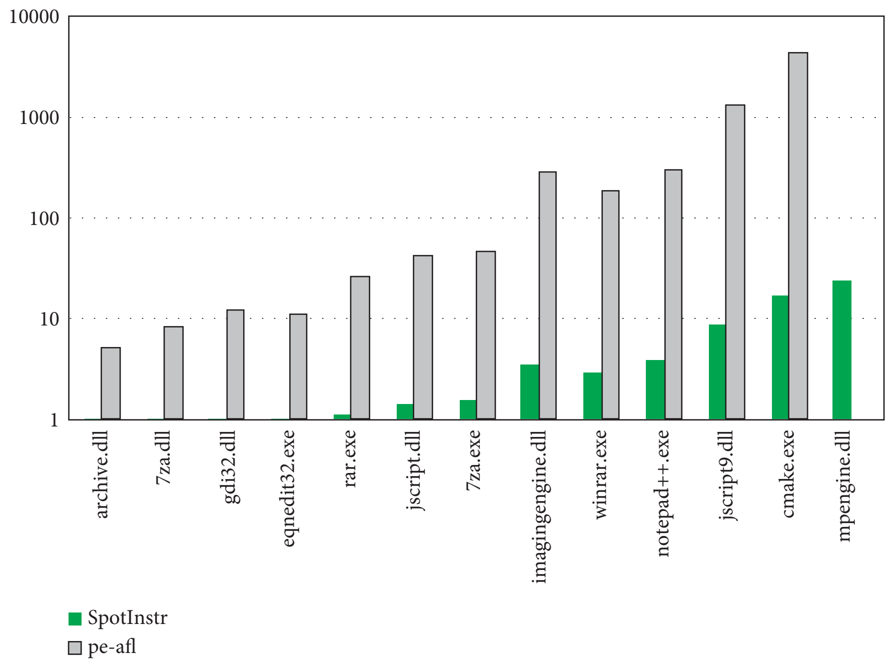


FIGURE 9: Time cost of instrumentation.

as expected, which is because jump mode introduces a large number of additional call instructions. However, the interesting thing is that the combination of jump mode and selective instrumentation makes the target surprisingly fast.

Figure 12 shows the total paths and unique crashes discovered by the fuzzers. As we can see, all SpotFuzzers discovered more paths than WinAFL profit from its high instrumentation rate and fast fuzzing speed. Not surprisingly, tools use selective instrumentation discover less paths than full instrumentation, that is because fewer basic blocks mean fewer paths.

The most important performance for a fuzzer is its ability to discover vulnerabilities. Figure 12 shows that all SpotFuzzers find more crashes than WinAFL, especially in the

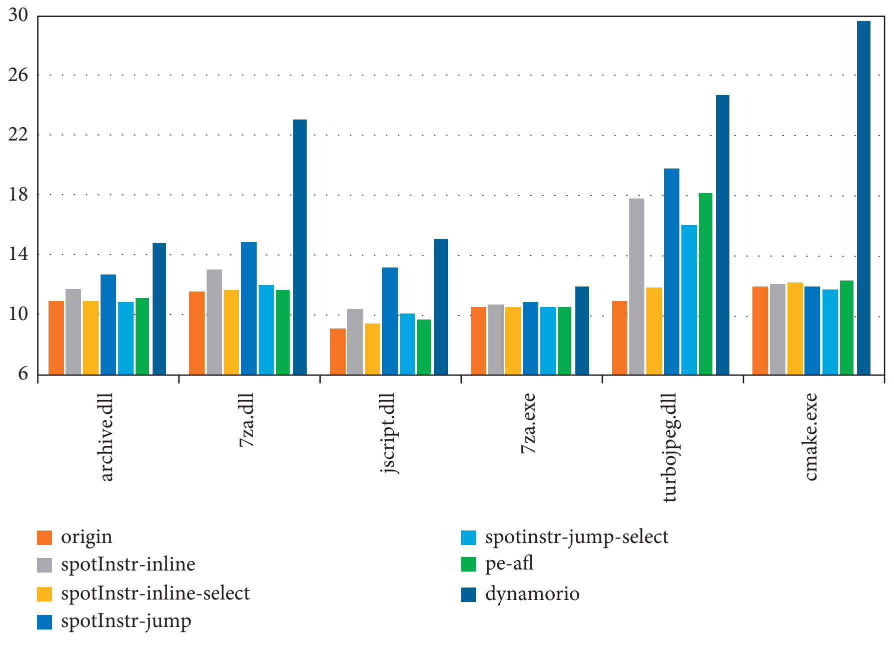


Figure 10: Execution time with different instrumentation types.

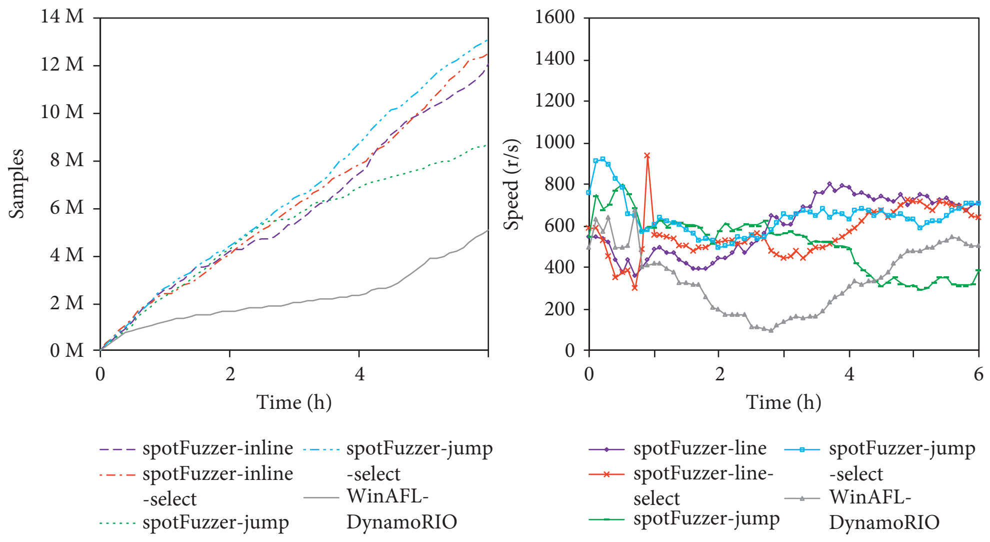


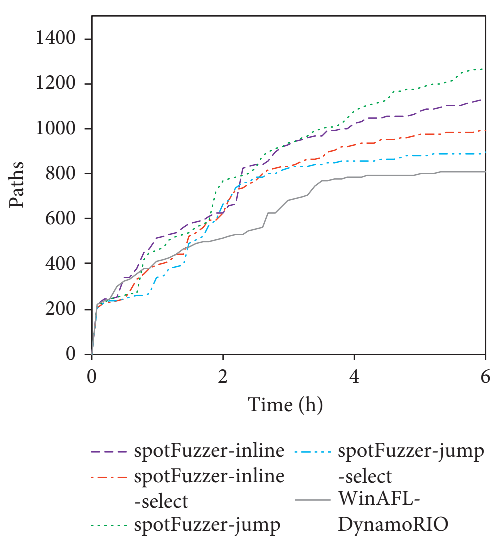
Figure 11: Number of total samples and fuzzing speed.

early time of fuzzing. The main reason is that SpotFuzzer has faster execution speed than WinAFL and discovers more execution paths. Although selective instrumentation has much fewer instrumented basic blocks than full instrumentation, it still achieved good fuzzing performance thanks to the faster execution speed and the memory-related selective instrumentation.

The unique crashes are not equal to unique vulnerabilities. After some analyze, we find that a high proportion of the unique crashes cause by the same vulnerable. This problem becomes more serious when more basic blocks selected for instrumentation. As shown in Table 5, all fuzzers can find a lot of unique crashes, but among which only a few are unique vulnerabilities. Despite this, our tools found more unique crashes than WinAFL.

5. Discussion

5.1. Instrument Basic Block Coverage. In this paper, when SpotInstr works on trampoline mode, a 5-byte jump instruction is chosen to fill the memory point for instrumentation, which means the points contains room less than 5 bytes will not be instrumented. According to our test data, we find such points will less than 10% of the total. We use the neighbor instruction to expand the point's room, which alleviates the problem to a certain degree. But there are still several ones left and cannot be instrumented. We notice that e9patch [21] try to reuse the instruction's origin bytes to construct a valid jump instruction. But that may cause a high virtual memory usage and make the process of instrumentation much more complex.



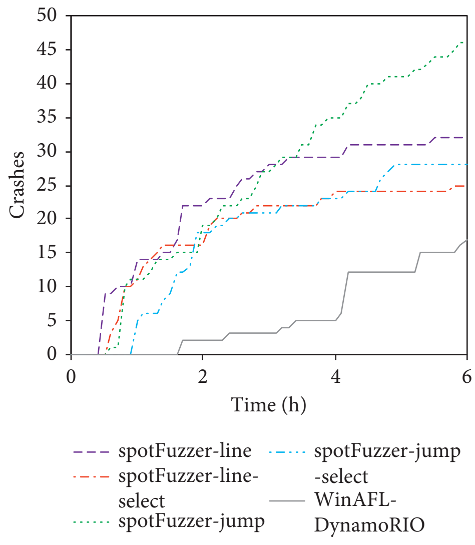


FIGURE 12: Number of total paths and unique crashes.

TABLE 5: Unique crashes versus unique vulnerabilities.

Target	SpotFuzzer				WinAFL
	inline	inline + select	jump	jump + select	
Instrument BBs	11951	2117	8454	1953	-
Execution paths	1127	993	1271	894	813
Unique crashes	32	25	46	28	17
Unique vulns	3	4	3	4	2

5.2. Static Instrumentation on Windows Kernel. The static instrumentation technology introduced in this paper should work with all Windows binaries. In theory, SpotInstr can instrument the Windows kernels, with the help of which, researchers can analyze or fuzzing the Windows kernel more efficiently.

5.3. Expanding the Usage of Static Instrument. Lots of researches [24, 25] have indicated that program instrumentation plays a very important role in program analysis. Program instrumentation can be used to memory access analysis [26], program behavior analysis [27], data structure recovery [28], and vulnerability mining [22], etc. But most of them are target to Linux or open-source software, there is little research on Windows binaries. We believe that a simple, stable, and usable static instrumentation tool is a good start for Windows binary analysis.

SpotFuzzer suffers from several limitations. First, it is currently restricted to Windows since we have only implemented the PE parsing module. Second, as the static instrumentation is designed for x86 binaries, SpotFuzzer is not applicable for across architectures. Third, the static instrumentation can only be used for binary fuzzing for now.

6. Related Works

In this section, we discuss the related work that are both complementary and orthogonal to our efforts in binary rewriting and fuzzing.

6.1. Binary Rewriting. The rewriting technology can be traced back to 1990s. At that time, the binary rewriting was mainly used to analyze or optimize the performance of programs, and almost all the tools like ATOM [29], QPT [30], EEL [31], and Etch [32] relied on static rewriting. After 2000, dynamic rewriting has become the mainstream research direction. A lot of successful tools appeared one by one: Dyninst [33], Vulcan [34], Vulgrind [35], DynamoRIO, PIN [35], QEMU, etc. Static rewriting has become a hot research direction again since 2010. At that time, new technology like reassembling was used to regenerate a binary. Tools like PEBIL [36], SecondWrite [37], BISTRO [38], Uroboros [39], Ramblr [40], Multiverse [41], RetroWrite, and E9Patch did a lot of work in theory and observation of static rewriting. Throughout all the static rewriting tools, we find that most of them are for Linux or Unix-like system. Only Etch are designed for Windows binaries, but it is too old for the modem operating system.

6.2. Fuzzing. Fuzzing is currently the most popular vulnerability discovery technique. Fuzzing was first proposed by Barton Miller at the University of Wisconsin in 1990s [42]. We find that AFL made Coverage-based grey-box fuzzing so popular and almost created a new fuzzing area. Lots of fuzzing tools developed upon AFL like AFL++ [43], AFLGo [9], AFLPIN [44], AFLSmart [45], FastAFLGo [10], and StFuzzer [46]. But for Windows the picture was very different: AFL first released in 2013, while WinAFL released in 2016. WinAFL use DynamoRIO to fetch feedback during execution, and its static mode based on syzygy almost unusable. Then, pe-afl was released for fuzzing Windows binaries, but only for PE32. In our experiments, pe-afl may cause some problem errors and made the target crash abnormally. We can hardly find a tool that can fuzzing Windows COTS software with static instrument.

7. Conclusion and Future Work

In this paper, we design two handy tools for instrumentation and fuzzing Windows binaries. The SpotInstr is a static instrumentation tool for Windows binaries without source code. It provides trampoline and inline mode for different usage scenario, and supported both PE32 and PE64. In other words, SpotInstr can instrument almost any binary on Windows. The SpotFuzzer was designed for fuzzing Windows COTS software. For general program, SpotFuzzer provides general fuzzing mode just like WinAFL. But for abnormal targets, like system service or kernel module, SpotFuzzer can switch to agent mode, and inject an agent to the target for fuzzing. What is more, we develop a memory-related selective instrumentation method for SpotInstr, which can reduce execution overhead and locate vulnerabilities faster.

All the algorithms used in SpotInstr are also applicable for other platforms such as Linux. In the future, we will try to support other platforms. As SpotInstr can instrument arbitrary code into the target binary, we plan to investigate the possibility of applying SpotInstr to different analysis works such as taint analysis, binary debugging, and software behavior identification for binaries.

Data Availability

The binaries used for testing in this paper are open-source software, commercial software, or operating system modules that are available from public sources.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the present study.

Acknowledgments

This work was supported by the National Key Research and Development Project (2019QY1305).

References

[1] L. Chen, C. Yang, F. Liu, D. Gong, and S. Ding, "Automatic mining of security-sensitive functions from source code," *Computers, Materials & Continua*, vol. 56, no. 2, pp. 199–210, 2018.

[2] V. J. Manès, H. Han, C. Han et al., "The art, science, and engineering of fuzzing: a survey," 2018, https://arxiv.org/abs/1812.00140.

[3] M. Zalewski, "American fuzzy lop," 2013, http://lcamtuf.coredump.cx/afl/.

[4] Peach.tech, "Peach fuzzer platform," 2015, https://www.peach.tech/wp-content/uploads/Peach-Fuzzer-Platform-Primer-DataSheet-Oct2015.pdf.

[5] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "KAFL: hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the USENIX Security Symposium 2017*, pp. 167–182, Vancouver, BC, Canada, August 2017.

[6] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: a greybox fuzzer for network protocols," in *Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification ICST*, pp. 460–465, Porto, Portugal, October 2020.

[7] X. Li, X. Pan, and Y. Sun, "Ps-fuzz: efficient graybox firmware fuzzing based on protocol state," *Journal of Artificial Intelligence*, vol. 3, no. 1, pp. 21–31, 2021.

[8] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *Proceedings of the USENIX Security Symposium 2019*, pp. 1099–1114, Santa Clara, CA, USA, August 2019.

[9] M. Böhme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas TX USA, November 2017.

[10] C. Du, T. Jin, Y. Guo, B. Jia, and B. Li, "FastAFLGo: toward a directed greybox fuzzing," *Computers, Materials & Continua*, vol. 69, no. 3, pp. 3845–3855, 2021, 3.

[11] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, Anaheim, CA, USA, April 2005.

[12] I. Fratric, "WinAFL," 2016, https://github.com/googleprojectzero/winafl.

[13] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," 2004, https://github.com/DynamoRIO/dynamorio.

[14] Hex-rays, "IDAPro," 2021, https://hex-rays.com/ida-pro/.

[15] Intel, "Intel® 64 and IA-32 architectures software developer's manual," 2021, https://www.intel.com/content/dam/.

[16] S. Porst, "PeLib," 2004, https://github.com/sporst/PeLib.

[17] L. Leong, "Pe-afl," 2018, https://github.com/wmliang/pe-afl.

[18] C. Hamilton, S. Ásgeirsson, and S. Marchand, "Syzygy," 2014, https://github.com/google/syzygy.

[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 *Proceedings of the Network and Distributed System Security Symposium (NDSS) 2021*, February 2021.

[20] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO-37 2004)*, pp. 81–92, IEEE Computer Society, Portland, OR, USA, December 2004.

[21] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, London UK, June 2020.

[22] D. Sushant, N. Burow, D. Xu, and M. Payer, "RetroWrite: statically instrumenting COTS binaries for fuzzing and sanitization," in *IEEE Symposium on Security and Privacy*, pp. 1497–1511, San Francisco, CA, USA, May 2020.

[23] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: sanitizer-guided greybox fuzzing," in *Proceedings of the USENIX Security Symposium*, Boston, MA, USA, August 2020.

[24] A. Ashish and J. Aghav, "Automated techniques and tools for program analysis: Survey," in *Proceedings of the Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pp. 1–7, Tiruchengode, India, July 2013.

[25] A. Gosain and G. Sharma, "A survey of dynamic program analysis techniques and tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications FICTA*, Bhubaneswar, Odisa, India, November 2014.

[26] P. Godefroid and J. Kinder, "Proving memory safety of floating-point computations by combining static and dynamic program analysis," in *Proceedings of the 19th international symposium on Software testing and analysis ISSTA '10*, Trento Italy, July 2010.

[27] D. Boardman, G. Greene, V. Khandelwal, and A. P. Mathur, "LISTEN: a tool to investigate the use of sound for the analysis of program behavior," in *Proceedings of the Nineteenth Annual International Computer Software and Applications Conference*, pp. 184–189, Dallas, TX, USA, August 1995.

[28] A. Slowinska, T. Stancescu, and H. Bo, "Howard: a dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium NDSS*, San Diego, CA, USA, February 2011.

[29] A. Srivastava and A. Eustace, "Atom: a system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pp. 196–205, Orlando Fl USA, June 1994.

[30] J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Software: Practice and Experience*, vol. 24, no. 2, pp. 197–218, 1994.

[31] J. R. Larus and E. Schnarr, "EEL: machine-independent executable editing," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 291–300, 1995.

[32] T. Romer, G. Voelker, D. Lee et al., "Instrumentation and optimization of win32/intel executables using Etch," in *Proceedings of the USENIX Windows NT Workshop*, pp. 1–7, Seattle, WA, USA, August 1997.

[33] B. R. Buck and J. K. Hollingsworth, "An api for runtime code patching," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[34] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: binary transformation in a distributed environment," Technical Report MSR-TR-2001-50, Microsoft Research, Redmond, Washington, 2001.

[35] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation PLDI*, San Diego, CA, USA, June 2007.

[36] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: efficient static binary instrumentation for Linux," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 175–183, White Plains, NY, USA, March 2010.

[37] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in COTS software with binary rewriting," in *Proceedings of the 26th IFIP TC International Information Security Conference. (SEC)*, pp. 154–172, Copenhagen, Denmark, June 2011.

[38] Z. Deng, X. Zhang, and D. Xu, "BISTRO: binary component extraction and embedding for software security applications," in *Proceedings of the 18th European Symposium on Research in Computer Security*, Egham U.K, September 2013.

[39] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the USENIX Security Symposium 2015*, pp. 627–642, Austin, TX.USA, August 2015.

[40] R. Wang, Y. Shoshitaishvili, A. Bianchi et al., "Ramblr: making reassembly great again," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS 2017*, San Diego, CA, USA, March 2017.

[41] E. Bauman, Z. Lin, and K. W. Hamlen, "Superset disassembly: statically rewriting x86 binaries without heuristics," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018*, Diego, CA, USA, February 2018.

[42] 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.

[43] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Boston, MA, USA, August 2020.

[44] Mothran, "Aflpin," 2015, https://github.com/mothran/aflpin.

[45] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1980–1997, 2019.

[46] J. Yang, X. Zhang, H. Lu, M. Shafiq, and Z. Tian, "StFuzzer: contribution-aware coverage-guided fuzzing for smart devices," *Security and Communication Networks*, vol. 2021, Article ID 1987844, 15 pages, 2021.