NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis

Jaeseung Choi, Kangsu Kim, Daejin Lee, Sang Kil Cha
KAIST
{jschoi17, kskim0610, djlee1592, sangkilc}@kaist.ac.kr

***Abstract*—Although it is common practice for kernel fuzzers to leverage type information of system calls, current Windows kernel fuzzers do not follow the practice as most system calls are private and largely undocumented. In this paper, we present a practical static binary analyzer that automatically infers system call types on Windows at scale. We incorporate our analyzer to NtFuzz, a type-aware Windows kernel fuzzing framework. To our knowledge, this is the first practical fuzzing system that utilizes scalable binary analysis on a COTS OS. With NtFuzz, we found 11 previously unknown kernel bugs, and earned $25,000 through the bug bounty program offered by Microsoft. All these results confirm the practicality of our system as a kernel fuzzer.**

I. Introduction

Software vulnerabilities in kernel code cause serious security breaches. At the very least, malicious attackers can produce a Blue Screen of Death (BSoD) on a victim machine. At worst, attackers can gain unprivileged access to the kernel space, which entails information disclosure or privilege escalation. For these reasons, the two biggest OS makers, i.e., Apple and Microsoft, are offering rewards up to $15,000 and $30,000, respectively, for reporting a critical vulnerability in their kernels [3], [57].

Therefore, there has been growing research interest on kernel fuzzing in both industry and academia [18], [30], [34], [40], [41], [48], [76]–[78], [84], [91].

One of the key strategies in kernel fuzzing research is to utilize types and dependencies of system calls (syscalls). Since syscall arguments are typically nested and dependent on each other, fuzzers often fail to generate meaningful test cases without recognizing types of syscalls.

Such an approach is easily achieved by Linux kernel fuzzers [18], [34], [76], [91] due to its open nature. However, Windows syscalls are widely unknown and undocumented. Furthermore, their convention frequently changes over time [38]. Although ReactOS [79] partially provides such information, it does not account for the latest syscalls.

To our knowledge, there is no existing Windows kernel fuzzer that generally infers type information from ever-changing syscalls of Windows. Instead, current fuzzers mitigate the challenge by (1) focusing on a small subset of the attack surface, such as font-related APIs [37] and the IOCTL interface [41], [74], or (2) relying on user-provided knowledge or harness code [25], [47], [84].

Therefore, we present NtFuzz, a Windows kernel fuzzer that leverages static binary analysis to automatically infer syscall types. At a high level, it runs in two steps. First, it statically analyzes Windows system binaries—`kernel32.dll`, `ntdll.dll`, and etc.—that invoke syscalls, and infers their argument types. Then, it uses the inferred types to fuzz the kernel by performing type-aware mutation on syscall arguments.

The key intuition behind our approach is that even though syscalls are largely undocumented on Windows, known (documented) API functions often call those syscalls through a chain of internal function calls. This means, we can bridge the information gap between documented and undocumented interfaces with static analysis by propagating the knowledge from documented functions to undocumented syscalls.

Unfortunately, designing a scalable static analyzer for Windows system binaries is challenging due to their huge size and interdependency. To statically infer syscall types by analyzing them, one needs to track both register and memory states while considering data flows between functions that are located in multiple different binaries. While there are a number of public tools focusing on CFG recovery [28], [32], [69] and single-binary analyses [13], [23], [42], [86], [92], we are not aware of any practical binary analysis solution that performs an inter-binary and inter-procedural analysis, in a scalable manner.

We overcome this challenge with a modular analysis, *a.k.a.* compositional analysis [2], [15], by designing a novel abstract domain. At a high level, our analyzer constructs a parameterized summary of each function, which describes the semantics of a function. When an analyzed function is later invoked from another function, we instantiate the parameterized summary to figure out the behavior of the function call. This way, we can efficiently analyze data flows and syscall types in an inter-procedural fashion, while avoiding redundant computations.

With the inferred syscall type information, NtFuzz then runs a type-aware fuzzing that launches a user application and intercepts syscalls to perform mutation on their argument values. This way, NtFuzz can automatically fuzz a Windows kernel with minimal manual effort from the user; NtFuzz does not require a user to write a harness code for the syscalls to test, as in kAFL [84] or pe-afl [47].

We evaluate our system on Windows 10, and show that type information obtained from our static analysis indeed helps find more kernel crashes. Moreover, NtFuzz found 11 unique bugs from the latest version of Windows 10, and four CVEs were assigned at the time of writing. A total of $25,000 bounty from Microsoft was awarded to the bugs we reported, which highlights the practical impact of our work.

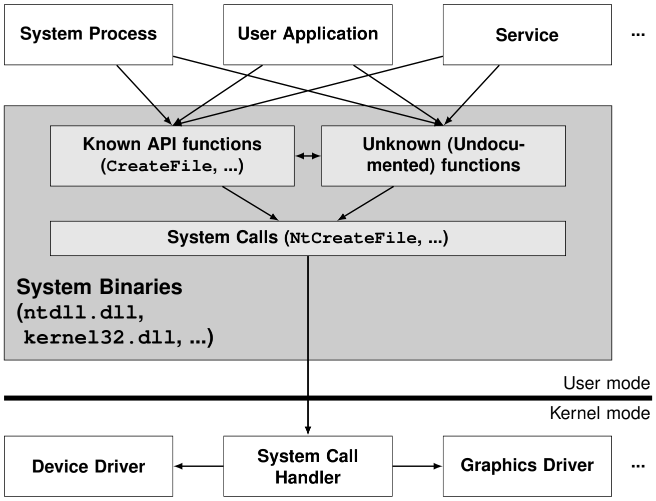


Fig. 1. Simplified architecture of Windows OS.

The contribution of this paper is as follows.

1) We integrate a static binary analysis technique with Windows kernel fuzzing for the first time.
2) We present a scalable static analyzer on Windows system binaries to infer the types of Windows syscalls.
3) We present NTFUZZ, a type-aware fuzzing framework to test Windows kernel with minimal manual effort.
4) We evaluate NTFUZZ on the latest Windows 10, and discuss 11 unique kernel bugs and four CVEs found.

II. BACKGROUND

In this section, we provide backgrounds required to understand our methodology for Windows kernel fuzzing.

A. Windows Architecture

Figure 1 illustrates a simplified architecture of Windows OS. User applications access system resources, such as I/O devices, through a system call (syscall). Typically, those applications do not directly invoke syscalls by themselves. Instead, they call a high-level *API function*, which will internally request a syscall. For example, to invoke the `NtCreateFile` syscall, a user would normally call the `CreateFile` function located at `kernel32.dll`, instead of directly invoking the syscall. Therefore, NTFUZZ aims at automatically figuring out syscall types by analyzing call chains from known API functions to undocumented functions, and to syscalls.

There are more than 1,600 syscalls in Windows 10, and the majority of them are *not* documented. Windows API functions are documented [62], [65], and their actual implementation is present in built-in system DLL files [55] that we refer to as *system binaries* in this paper.

Our technique statically analyzes these system binaries in order to infer the types of arguments passed to syscalls. Since there are numerous API functions and DLL files on Windows, we focus only on the core system libraries that we manually identified (see Table I). In Windows 10 17134.1, which was used for our evaluation (§VII-B), 80.4% of the existing syscalls were invoked at least once from these binaries. Note, we can extend the list if needed, although it was enough for our purposes (see §VIII).

TABLE I
LIST OF SYSTEM BINARIES THAT OUR STATIC ANALYSIS TARGETS.

Binary	Description
`ntdll.dll`	Syscalls and APIs for native applications [56]
`kernel32.dll`	Management of core system resources (e.g. file)
`kernelbase.dll`	——————"——————
`win32u.dll`	Syscalls for graphic user interface (GUI)
`gdi32.dll`	Graphic device interface to control video displays
`gdi32full.dll`	——————"——————
`user32.dll`	Management of UI components (e.g. window)
`dxcore.dll`*	Interface for DirectX functionalities

* This file presents on Windows 10 starting from the build 18362.

B. Static Program Analysis

Static program analysis (static analysis in short) refers to a methodology for automatically predicting the behavior of programs without running them [82]. While there is a wide spectrum of techniques, static analyses can be described using a general theoretical framework named abstract interpretation [20], [21]. In abstract interpretation, a concrete program state is approximated with an *abstract domain*, and a program is analyzed with *abstract semantics*, which subsumes the concrete semantics of the program.

In the field of static analysis, we say an analyzer is *sound*, if the analyzer has no false negative; if a sound analyzer reports a program as bug-free, then the safety of the program should be guaranteed. Similarly, we say an analyzer is *precise* (or complete), if the analyzer is free from false positives [67]. We note that both the terms are used for different meanings in other fields [86]. In this paper, however, we will follow the traditional convention in static analysis. If the analysis result happens to be both sound and precise, we will describe the analysis to be *accurate*, or *correct*.

III. MOTIVATION

In this section, we motivate our research by showing a code snippet taken from one of the CVEs we found (see §VII-E).

Figure 2 presents a simplified version of CVE-2020-0792. The bug exists in the `NtUserRegisterWindowMessage` syscall, which takes in a `UNICODE_STRING` pointer as input. The function validates the input fields in Line 11 in order to make sure that the user-provided pointer (`buf`) accesses the right memory region. However, the function skips the entire check when the length of the buffer (`len`) has an odd value in Line 6. The problem is that the `LogError` function in Line 7 does not abort the syscall upon execution. Therefore, one can effectively circumvent the safety check in Line 11 by specifying an odd length (`Length`), and an invalid memory pointer (`Buffer`). This vulnerability allows an attacker to access the kernel memory and gain escalated privilege by carefully crafting the input fields.

From this example, we can make the following important observations.

```
1  // Syscall in kernel-mode.
2  NtUserRegisterWindowMessage(UNICODE_STRING* arg) {
3    ... // Sanitize 'arg'.
4    unsigned short len = arg->Length;
5    wchar_t* buf = arg->Buffer;
6    if ( len & 1 ) {
7      LogError(...); // Does not abort.
8    }
9    else {
10     tmp = ((char*)buf) + len + 2;
11     if (tmp >= 0x7fff0000 || tmp <= buf || ... ) {
12       return;
13     }
14   }
15   ... // Access 'buf'.
16 }
17 // API function in user-mode.
18 RegisterWindowMessage(char* s) {
19   UNICODE_STRING str;
20   str.Buffer = malloc(2 * strlen(s) + 2);
21   str.Length = 2 * strlen(s);
22   ...
23   NtUserRegisterWindowMessage(&str);
24 }
```

Fig. 2. Simplified pseudo-code of CVE-2020-0792 found by NtFuzz.

First, it is difficult for fuzzers to trigger the bug without recognizing the type of `NtUserRegisterWindowMessage`. A fuzzer needs to know that the input argument of the syscall is a pointer, and that it should point to a structure of type `UNICODE_STRING`. Let us assume that the fuzzer blindly generates a value without knowing the type information. It is unlikely for the generated value to have a desired `UNICODE_STRING` structure where the `Length` field is odd and the `Buffer` field points to an invalid memory region.

Second, undocumented syscalls are often related to documented API functions. `RegisterWindowMessage` in our example is documented in Microsoft Docs [58], while `NtUserRegisterWindowMessage` is not. However, we can infer the type of this syscall via known type information of the documented API function. For instance, we can observe how `RegisterWindowMessage` initializes the local `UNICODE_STRING` structure, using its `char*` argument.

Third, API-function-level fuzzing may *not* trigger critical bugs, even if API functions eventually invoke syscalls. Note that the caller of `RegisterWindowMessage` cannot fully control the input to `NtUserRegisterWindowMessage` because `RegisterWindowMessage` always sets the `Length` field with an even number in Line 21. As a result, the bug in the syscall will never be triggered if we only fuzz the API function. This highlights the importance of direct syscall fuzzing.

IV. Overview

In this section, we first describe the overall architecture of NtFuzz (§IV-A). We then present an overview of our modular analysis (§IV-B) and our running example (§IV-C).

A. NtFuzz *Architecture*

Figure 3 illustrates the overall architecture of NtFuzz, which comprises two core components: (1) the static analyzer, and (2) the kernel fuzzer. At a high level, the static analyzer takes in a set of system binaries and API specifications as input, and outputs syscall type information by analyzing the binaries. The kernel fuzzer then repeatedly runs a seed application[1] while mutating the arguments of the invoked syscalls based on the type information gathered.

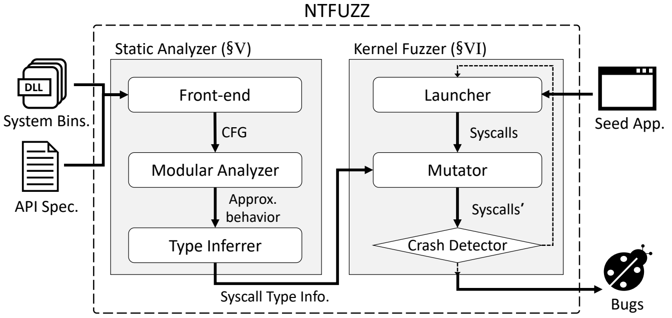


Fig. 3. Architecture of NtFuzz.

1) Static Analyzer: The static analyzer consists of three major components. First, the front-end lifts system binaries into intermediate representations and constructs Control-Flow Graphs (CFGs). It also parses the given API specification—the type information of documented API functions—and turns it into a suitable form for analysis (see §V-A).

Next, the modular analysis engine traverses the interprocedural CFGs and observes how the system binaries construct syscall arguments. Specifically, it analyzes how each argument of documented API functions flows into syscalls. We sketch the key idea of modular analysis in §IV-B, and present the detailed design in §V-B.

Finally, the type inferrer decides the argument types of each syscall based on the analyzed behavior of the binaries. To improve the accuracy of analysis, we aggregate the information obtained from multiple callsites of syscalls, and decide the final argument types (see §V-C).

2) Kernel Fuzzer: This module runs by repeating the following three steps. First, the launcher prepares for syscall hooking and runs a given seed application. The mutator then mutates the argument values of each syscall during the execution of the application. This mutation process is based on the type information obtained from the static analyzer. Finally, the crash detector checks if the kernel crashes, and if so, it retrieves the corresponding memory dump. Otherwise, it does nothing. These steps iterate until a timeout is reached. We describe the design details of each component in §VI.

B. *Modular Analysis Algorithm*

The key aspect of NtFuzz is the use of modular analysis [2], [15] to infer syscall types. At a high level, modular analysis computes the whole program behavior by (1) splitting the target program into multiple modules, and (2) assembling the analysis results of each component. NtFuzz divides a program into multiple functions. For each function, the

[1] Note our seed is different than seed inputs used in regular fuzzers [80].

Algorithm 1: Modular Analysis Algorithm.

```
1 function Analyze(CFGs, callGraph, APISpec)
2     summaries ← {}
3     typeInfo ← ∅
4     for $f$ in TopoSortReverse(callGraph) do
5         $s$ ← Summarize(CFGs[$f$], summaries, APISpec)
6         summaries[$f$] ← $s$
7         typeInfo ← typeInfo ∪ CollectType($s$)
8     return DecideType(typeInfo)
```

analyzer investigates its semantics and constructs a summary to capture its behavior. When a summarized function is later called by another function, we refer to this summary instead of analyzing the callee again. This way, the behavior of the whole program is composed in a bottom-up style.

Algorithm 1 describes the pseudo-algorithm of our modular analysis for syscall type inference. The analysis takes in CFGs (`CFGs`), an inter-binary call graph (`callGraph`), and a parsed API specification (`APISpec`) as input, and returns inferred syscall type information. The algorithm first topologically sorts the call graph in Line 4, and traverses it starting from the leaf nodes. Each function in the call graph is then summarized with the abstract interpretation framework (see §V-B) in Line 5. A summary of a function f captures (1) which syscalls are invoked by f with which arguments, and (2) how the memory state varies by running f. Note that we run this analysis from the leaf nodes in order to reuse their function summaries in the callers. In Line 8, `DecideType` emits the final syscall types using the information accumulated from each function within the loop (see §V-C).

There are several benefits of using the modular analysis. First, our analysis is naturally inter-procedural and context-sensitive [73], [85] as we utilize function summaries. Second, our analysis operates on each function only once. This can greatly reduce the cost of analysis compared to traditional global analyses [9], [72], which can potentially analyze the same function multiple times [71].

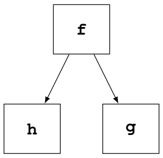
Of course, such benefits come at a price. For example, as modular analysis requires callees to be analyzed before the caller, it cannot soundly handle recursive calls or indirect calls. This is an inherent limitation of this technique, and other modular analysis tools, e.g., Infer [15], suffer from the same issue. In our implementation, recursive calls or unresolved indirect calls are unsoundly ignored as NOPs. However, the evaluation in §VII-B indicates that our analyzer can still collect meaningful information and yield a fair degree of accuracy despite such problems.

C. Running Example

We now present a running example to give a brief overview of our modular analysis. Figure 4 shows our running example, which consists of three functions. We assume that the function `f` is the only documented API function. That is, we know the exact type of `f`. For simplicity, we consider `malloc` and `syscall` as built-in primitive functions for memory allocation and syscall invocation, respectively. Note that we show this example in C for ease of explanation, but our actual analysis runs on a binary. Therefore, the types of the arguments and variables are not known. To indicate this, we deliberately omit type notations for `g` and `h`.

```
1  // Documented API.
2  void f(HANDLE x) {
3    p = (char *)malloc(8);
4    h(p, x);
5    h(p + 4, 10);
6    g(p);
7  }
8  void g(y) {
9    syscall(20, y);
10 }
11 void h(a, b) {
12   *a = b;
13 }
```

(a) Example in C.



(b) Call Graph.

```
/* struct begin */
+0: HANDLE
+4: int
/* struct end */
```

(c) Inferred struct `y`.

Fig. 4. Running example for modular analysis.

Our goal here is to figure out the types of the syscall arguments in Line 9.

1) Walk-Through: NtFuzz starts by topologically sorting the call graph in Figure 4b and traverses the sorted nodes in a reverse order starting from leaf nodes. There are two possible orders: either `h` → `g` → `f` or `g` → `h` → `f`. Assume that NtFuzz selects the former. It will firstly analyze `h` and produce a summary including the information about the side effect "`*a = b`", which means the memory location pointed to by the first argument (`a`) is updated with the value of the second argument (`b`).

Next, NtFuzz produces a summary for `g`, which contains a syscall with a constant `10` as its first argument and `y` as its second argument. In this case, there is no side effect to summarize for `g`. Note that our function summaries are *parameterized*. That is, a summary can be concretized later into different instances based on the provided arguments.

We now compute the summary of `f` by reusing the existing summaries for `h` and `g`. First, we obtain the type of `f` from the given API specification. We then instantiate and apply the summarized side effect of `h`, and find out how the heap object generated in Line 3 is updated via function calls (Line 4–5). When `p` is passed as an argument to `g` in Line 6, we instantiate the summarized syscall information of `g`, which reveals that the second argument of the syscall, in Line 9, is a pointer to a structure. The structure has a `HANDLE` field at the offset zero (Line 4), and an integer field at the offset four (Line 5). Therefore, we can identify the type of the second syscall argument as in Figure 4c. Note that the type of the first argument is already known during the analysis of `g`. Thus, at this point, we have complete type information for the syscall.

2) Challenges: Although the running example is deliberately simplified, accurately inferring the syscall type is *not* trivial for the following reasons.

First, our analysis should be able to trace data flows across function boundaries. In particular, it should understand data flows from `f` to `g` in order to correctly identify the syscall type: we should realize that the argument `y` of `syscall` is from p in Line 3.

Furthermore, our analysis should be able to track the memory states along program executions in an inter-procedural manner. For example, it should know how the heap object, which is pointed to by p, is allocated and how its contents are set via h. In this case, the function h has an assignment to a pointer, which produces a memory side effect. Therefore, binary analysis tools that only support register data flows or intra-procedural analyses cannot handle this.

V. Static Analyzer Design

This section presents our design of the static analyzer module, which is responsible for syscall type inference.

A. Front-End

The primary role of the front-end is to parse Windows system binaries and documented API specifications for our modular analysis.

We use B2R2 [36] to parse and lift binaries, which is fast enough to deal with large system binaries. B2R2 lifts binary code into intermediate representations (IRs), which describe semantics of binary code using only a few primitive operations. Figure 5 shows a simplified syntax of B2R2 IR. Note, we deliberately omitted many expressions for ease of explanation. For example, we removed unary operations and even branch statements except for call statements as their semantics are straightforward. The lifting process runs recursively by following jump targets and constructs a CFG for each function. With these CFGs, we build an inter-binary call graph.

To reduce the size of the call graph, we filter out unnecessary nodes, i.e., functions, from it. First, we identify syscall stub functions that execute a sysenter instruction, and then traverse back the call graph from the identified stub functions up to a documented API function. Note we stop the traversal at a documented API function because we already have complete type information for documented functions, and there is no benefit from further analyzing their callers. We accumulate all the functions encountered, including the stub functions, and denote them as S_1.

Next, we collect all reachable functions from S_1 and let the resulting set be S_2. This is to fully capture side effects incurred by the functions called from S_1. Finally, we prune the original call graph by leaving only the functions that belong to $S_1 \cup S_2$.

The front-end is also responsible for parsing Windows API specifications. Currently, the specifications are obtained from the header files in Windows 10 SDK [65]. We parse the function declarations in these files and obtain type information for each function. The declarations also include useful annotations written in Source Annotation Language (SAL) [63]. For example, SAL can denote what is the size of an array type argument. We parse these annotations too and use them for array type inference (see §V-C2).

B. Modular Analyzer

Recall from §IV-B, our modular analysis summarizes the behavior of each function with abstract interpretation [20].

$$\begin{array}{lcll}
exp & ::= & reg & \texttt{// Register} \\
 & | & [\ exp\ ] & \texttt{// Memory load} \\
 & | & int & \texttt{// Number} \\
 & | & exp \Diamond_b exp & \texttt{// Binary operation} \\
stmt & ::= & \texttt{Put}\ (reg, exp) & \\
 & | & \texttt{Store}\ (exp, exp) & \\
 & | & \texttt{Call}\ (f) &
\end{array}$$

Fig. 5. Simplified subset of B2R2 IR syntax for explanation.

$$\begin{array}{llcl}
\text{(Abstract Integer)} & \mathbb{I} & = & a * symbol + b \mid \bot \mid \top \\
\text{(Abstract Location)} & \mathbb{L} & = & \texttt{Global}(\mathbb{Z}) \\
 & & | & \texttt{Stack}(function \times \mathbb{Z}) \\
 & & | & \texttt{Heap}(allocsite \times \mathbb{Z}) \\
 & & | & \texttt{SymLoc}(symbol \times \mathbb{Z}) \\
\text{(Type Constraint)} & \mathbb{T} & = & \tau \\
 & & | & \texttt{SymTyp}(symbol) \\
\text{(Abstract Value)} & \mathbb{V} & = & \mathbb{I} \times 2^{\mathbb{L}} \times 2^{\mathbb{T}} \\
\text{(Register Map)} & \mathbb{R} & = & reg \to \mathbb{V} \\
\text{(Memory)} & \mathbb{M} & = & \mathbb{L} \to \mathbb{V} \\
\text{(Abstract State)} & \mathbb{S} & = & \mathbb{R} \times \mathbb{M}
\end{array}$$

Fig. 6. Our abstract domains.

At a high level, we perform a flow-sensitive analysis on each function, which computes an abstract program state for each point in the CFG. With these abstract states, we summarize each function by observing (1) which values are passed as a syscall argument, and (2) how the state changes between the entry and exit node of the function.

1) Abstract Domain: We define our abstract domains in Figure 6, and present their join operation in Appendix A. $\mathbb{Z}$ denotes the integer set, and *symbol* is a new symbol introduced at each argument of a function. Since we do not know the value of a function argument at its entry, we initialize every argument value with a fresh symbol.

Our abstract value $\mathbb{V}$ is a triple of an abstract integer, a set of abstract locations, and a set of type constraints.

First, abstract integers represent a numeric value that a register or a memory cell can hold. We use a linear expression with a symbol to represent an abstract integer. It can be either concrete, when $a = 0$, or symbolic, when $a \neq 0$.

Second, abstract locations present a potential location of a value. We have four different kinds of locations: Global, Stack, Heap, and SymLoc. Global(a) means a global variable location at the address a, Stack(f,o) represents a local variable located in the stack frame of f at the offset o, and Heap(a,o) represents a memory cell located at the offset o of a heap object allocated from the address a. We coalesce the heap locations based on their allocation site [39]. SymLoc(s,o) indicates a symbolic pointer with the pointer symbol s and the offset o from the pointer. Note that in static analysis, a pointer can point to multiple locations due to over-approximation. Thus, our abstract value entails $2^{\mathbb{L}}$.

Third, we embed type constraints in our abstract domain as in [19]. A type constraint can have either a concrete type τ, or a symbolic type SymTyp(α), where alpha is a symbol.

For example, consider a function that takes in a single argument as input. The argument can either be an integer or a pointer, and we take both cases into account. We first assume that the argument has an abstract integer $1 * \alpha_1 + 0$

$$\mathcal{V}(reg)(S) = S[0](reg)$$

$$\mathcal{V}([e])(S) = \bigsqcup\{S[1](l) \mid l \in \mathcal{V}(e)(S)[1]\}$$

$$\mathcal{V}(i)(S) = \begin{cases} \langle 0, \phi, \phi \rangle & \texttt{if } i = 0 \\ \langle \bot, \{\texttt{Global}(i)\}, \phi \rangle & \texttt{if } i \texttt{ in data section} \\ \langle i, \phi, \{integer\} \rangle & \texttt{otherwise} \end{cases}$$

$$\mathcal{V}(e_1 \Diamond_b e_2)(S) = binop(\Diamond_b, e_1, e_2, S)$$

(a) Evaluation of expressions (exp).

$$\mathcal{F}(\texttt{Put}(r, e))(S) = \langle R[r \mapsto \mathcal{V}(e)(S)], M \rangle, \ \texttt{where } S = \langle R, M \rangle$$

$$\mathcal{F}(\texttt{Store}(e_1, e_2))(S) = \langle R, update(\mathcal{V}(e_1)(S)[1])(\mathcal{V}(e_2)(S))(M) \rangle, \quad \texttt{where } S = \langle R, M \rangle$$

$$update(L)(v)(M) = \begin{cases} M[l \mapsto v] & \texttt{if } L = \{l\} \\ M[l_1 \overset{w}{\longmapsto} v] ... [l_n \overset{w}{\longmapsto} v] & \texttt{if } L = \{l_1, ..., l_n\} \end{cases}$$

$$\mathcal{F}(\texttt{Call}(f))(S) = \begin{cases} apply(\delta, S) & \texttt{if } f \texttt{ has side effect } \delta \\ S & \texttt{otherwise} \end{cases}$$

(b) Evaluation of statements ($stmt$).

Fig. 7. Abstract semantics of our static analyzer.

(or simply α_1), where α_1 is a fresh symbol. We also consider the abstract location of the argument as it can be a pointer. Thus, we introduce a new symbol α_2 for representing a symbolic location $\texttt{SymLoc}(\alpha_2, 0)$. Since we do not know the type constraint of the argument, we create a symbolic type constraint $\texttt{SymTyp}(\alpha_3)$. Finally, we subsume all the above information to initialize an abstract value $\mathbb{V}$ for the argument as $\langle \alpha_1, \{\texttt{SymLoc}(\alpha_2, 0)\}, \{\texttt{SymTyp}(\alpha_3)\} \rangle$. Once we finish analyzing the function, we will obtain a summary *parameterized* with these symbols.

Note our abstract domain differs from that of Value Set Analysis (VSA) [5], [6]. VSA uses a variant of interval domain to trace offsets of abstract locations, which enables a sound analysis of memory access range, while making it prone to imprecision. Since our focus is not on the soundness, we seek for more precise results by giving up tracing complex offsets. Instead, we focus on tracking constant offsets with $\mathbb{Z}$. We further justify our design in §V-B2.

2) Abstract Semantics: We now define our abstract semantics on the B2R2 IR shown in Figure 5. Recall that the IR is largely simplified for ease of explanation, and our actual implementation of NtFuzz handles the complete syntax.

First, we define $\mathcal{V} : exp \rightarrow \mathbb{S} \rightarrow \mathbb{V}$ that evaluates the given expression exp within the provided abstract state $\mathbb{S}$ to return an abstract value $\mathbb{V}$. Figure 7a presents the semantics. Here, we use $X[n]$ to denote the $(n+1)$-th element of a tuple. For example, for an abstract state $S \in \mathbb{S}$, $S[0]$ returns the register map ($\mathbb{R}$) of the abstract state.

Register read and memory load expressions are trivial to evaluate: we simply look up the given abstract state $\mathbb{S}$ and return the corresponding value. Note that the four kinds of abstract locations defined in Figure 6 are naturally distinguished as separate regions during the memory load.

Evaluating a number expression involves range checks. When a given number is zero, we safely ignore its type as we cannot distinguish between a NULL pointer and a constant zero. When a number is within the range of data sections, we consider it as a global pointer. For the rest of the cases, we consider the number as an integer.

Binary operation *binop* is formally defined in Appendix B, and intuitively it performs arithmetic operations with the abstract values and generates type constraints if possible. For example, a multiplication of two values should have an integer type. When computing a binary operation, we also calculate a new offset of the abstract location $\mathbb{L}$. To explain this semantics, let us consider the following x86 snippet of a function `f`.

```
mov esi, ebp
sub esi, 40            # esi: array base
lea edi, [esi+4*ecx]  # ecx: array index
```

This code computes the address of an array element indexed by `ecx`. We will assume that `ebp` carries the initial stack pointer at the entry of `f`, which will be converted as an abstract location $\texttt{Stack}(\texttt{f}, 0)$ in our domain. Ideally, traditional VSA will try to capture the exact range of `ecx` with its congruence interval domain, and calculate the accurate memory range pointed to by `edi`. However, if the range of `ecx` is imprecisely approximated to $\top$, i.e., $[-\infty, \infty]$, `edi` will be considered as pointing to any arbitrary memory location.

To mitigate such problems, we trace only the constant offsets, and ignore array index terms. For the `sub` instruction in this example, we calculate the abstract location assigned in `esi` as $\texttt{Stack}(\texttt{f}, -40)$. However, for the `lea` instruction, we ignore the added index term, and simply assign the abstract locations of `esi` to `edi`. Intuitively, ignoring array index means coalescing all the array elements into a single abstract location. Such an array-insensitive design is indeed common for static analyzers [72], [88].

We now define $\mathcal{F} : stmt \rightarrow \mathbb{S} \rightarrow \mathbb{S}$ that evaluates the given statement $stmt$ within the given abstract state and returns a new abstract state. Figure 7b shows the definition of $\mathcal{F}$. We use $m[k \mapsto v]$ to denote a *strong* update of m with a new mapping from k to v. Meanwhile, $m[k \overset{w}{\longmapsto} v]$ means a *weak* update: m is updated with a new mapping from k to $m(k) \sqcup v$.

The abstract semantics of `Put` and `Store` are straightforward: they simply update the given abstract state with an evaluated value. For the rationale behind distinguishing strong and weak update, see [22].

The core semantics of `Call` is to apply the summarized side-effect δ of the called function (see Appendix B for more details of function *apply*). If the callee f is a documented function, we can also update the type constraints of its arguments accordingly. Additionally, we manually encode the semantics for several core functions to improve the accuracy of our analysis. For example, we encode the memory allocation semantics of `RtlAllocateHeap` instead of analyzing the function. Note writing custom semantics for core functions is a common practice in static analysis [26], [83].

3) Comparison against VSA: As we discussed in §V-B2, VSA can soundly analyze memory accesses by tracking location offsets with interval domain. Unfortunately, it is notoriously difficult to achieve precise interval analysis in practice, even with source code [46], [72]. When the interval analysis yields imprecise results like $[-\infty, \infty]$, pointer values also lose precision in VSA. Meanwhile, our semantics design is unsound, but it can always track a concrete offset for abstract locations. That is, we trade-off soundness for precision.

Note, unlike our domain, the VSA domain does not suit modular analysis well. To enable modular analysis with VSA, we must extend its domain to support symbolic intervals. At the entry of a function, we must initialize its argument to have a symbolic location with symbolic boundaries, such as $\texttt{SymLoc}(s, [\alpha, \beta])$. However, when the offset interval is symbolic, it becomes extremely difficult to handle memory operations in the abstract semantics. In Appendix C, we discuss more details about possible strategies to cope with this challenge, and explain why they are imperfect.

4) The Trade-Off: Since our goal is not at sound program verification, we can compromise the soundness of our analysis to improve its scalability and precision. We briefly summarize some of the design choices we made for the trade-off.

During the intra-procedural analysis, we perform loop unrolling, which is a popular technique that many static analyzers adopt to control the soundness [31], [50]. Also, we deliberately allow strong updates for locations that should be always weakly updated (e.g. heap locations). Note that $\mathcal{F}$ in Figure 7b only checks whether the update is on a single location or not.

We also found that functions in real-world binaries often take in nested pointers as an argument, which involves numerous memory updates. As a result, we frequently observe an explosion in the number of side effect entries. According to our experience, soundly summarizing all these side effects made the analyzer unscalable.

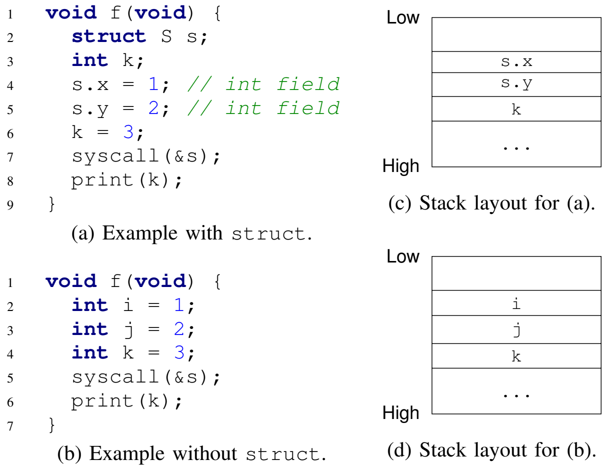
Therefore, we introduce N_{SE} parameter, which sets the maximum bound on the number of update entries to store for a function summary. That is, we unsoundly prune out the entries over this number. In §VII-B, we evaluate the impact of N_{SE} on the accuracy and scalability of the analysis.

C. Type Inferrer

The type inferrer decides syscall types based on the results obtained from the modular analyzer.

1) Structure Inference: If a syscall argument is not a pointer, we can trivially know its type from our abstract domain in Figure 6. However, if the argument is a pointer, we have to carefully inspect the analyzed memory state in order to infer the type of the pointee. Let us revisit the running example in Figure 4. At the syscall in Line 9, the abstract value of the second argument should be $\langle \bot, \{\texttt{Heap(Line 3}, 0)\}, \phi\rangle$. Therefore, we look for the abstract memory cells, i.e. abstract locations, that correspond to the heap object allocated in Line 3. That is, we search for any abstract memory cell that is in the form of $\texttt{Heap(Line 3}, *)$, where $*$ indicates any offset. This way, we can obtain two abstract memory cells $\texttt{Heap(Line 3}, 0)$ and $\texttt{Heap(Line 3}, 4)$, and their abstract values to obtain the structure in Figure 4c.

On the other hand, it is more challenging to infer a structure type allocated on the *stack*. Consider an example in Figure 8a, where the function `f` allocates a structure `s` on the stack. At the binary level, the initialization of the two `int` fields cannot be distinguished from the initialization of two local variables. That is, the binaries obtained from compiling the two different sources in Figure 8a and Figure 8b will be identical. Therefore, if a syscall argument is a pointer to the stack, we cannot decide whether it is a pointer to a structure (e.g. `&s`) or a singleton variable (e.g. `&i`). If we conservatively consider it as a structure, we have to determine its *boundary*. We may assume that the structure continues until the end of the stack frame, but we will end up having too many spurious fields.



```
1 void f(void) {
2   struct S s;
3   int k;
4   s.x = 1; // int field
5   s.y = 2; // int field
6   k = 3;
7   syscall(&s);
8   print(k);
9 }
```

(a) Example with `struct`.

```
1 void f(void) {
2   int i = 1;
3   int j = 2;
4   int k = 3;
5   syscall(&s);
6   print(k);
7 }
```

(b) Example without `struct`.

(c) Stack layout for (a).

(d) Stack layout for (b).

Fig. 8. Example for the inference of structure allocated on stack.

We heuristically address this problem by observing memory access patterns of a function. First, when an adjacent stack variable is defined but never used, we consider it as a structure field passed to the syscall. Second, if such a variable is used without any definition, we consider it as a structure field initialized by the syscall.

Assume that we are inferring the syscall type at Line 7 of Figure 8a. To decide the boundary of the structure `s`, we first examine the most adjacent next location of it, which is `s.y` in this case. Since this location is defined at Line 5 but not used anywhere within the function, we conclude that this location must be a part of the structure `s`. In contrast, for the next adjacent location (`int k`), we can observe its use at Line 8. Therefore, we decide that this location does not necessarily belong to the structure. To avoid including spurious fields, we conclude that the structure has only two fields. For finding undefined/unused locations, we use the standard reaching definition analysis and liveness analysis [1].

2) Array Inference: There are mainly two ways to infer array types. First, a known array type from a documented API may flow into a syscall. We found that an array size is often declared with a variable, which means that the array size is passed to a function as a separate argument. Let us consider the example in Figure 9. The declaration of the `Data` structure is annotated with the SAL annotation `_Field_size_`, which describes that the size of the array `buf` is specified by the field `n`. When a `Data` structure flows from a documented API

```
1  void f(struct Data* d) {
2    syscall_1(d->buf, d->n);
3  }
4
5  void g(x) {
6    p = malloc(x);
7    memset(p, 0, x);
8    syscall_2(p, x);
9  }
```

(a) Example snippet in C.

```
struct Data {
  _Field_size_(n)
  int* buf;
  size_t n;
};
```

(b) The Data structure.

Fig. 9. Example for the array inference.

function to the function f, we know that the first argument of the syscall_1 is a pointer to an int array, and the second argument of it is the size of the array.

Second, NtFuzz can also directly infer an array type by observing the memory allocation pattern. Let us consider function g in Figure 9. Recall from §V-B, we initialize the argument x with a symbolic value. In Line 8, we can figure out that the size of the heap object pointed to by p is always the same as x. Therefore, we can infer that the first argument of the syscall is an array pointer, whose size is given by the second argument of the syscall.

3) Resolving Conflicts: Due to the over-approximating nature of static analysis, we inevitably encounter contradictory type constraints. For example, our analysis can return an abstract value for a syscall argument as $\langle \alpha, \phi, \{handle, integer\} \rangle$. When such contradictory type constraints are encountered, a sound type system like TIE [45] will output a $\top$ type. However, since the goal of our analysis is to help in type-aware kernel fuzzing, we try to emit more precise type information instead of returning too many $\top$ types.

To this end, we aggregate the type information obtained from different call sites of a syscall. The key intuition here is that since the same syscall is often invoked from multiple different functions, we can have a majority vote from them. While some of them may suffer from imprecise results, we expect the remainders to yield accurate type information. Therefore, for each syscall argument, we collect type constraints from every call site of the syscall, and select the majority type as its final type. Note that such a decision has to be recursively performed in certain cases. For example, when a syscall argument is a pointer type, we should determine the type of the pointee again with the majority decision.

D. Implementation

To implement the binary front-end, we imported the B2R2 project [36], and wrote 739 source lines of our own F# code. For the API specification front-end, we wrote 1,854 source lines of F# code. We also wrote 1,239 lines of Lex/Yacc rules and used FsLexYacc [89] package to automatically generate the parser code for SDK header files. Finally, the core engine for static analysis and type inference is implemented in 6,227 source lines of F# code. Although our analyzer currently targets Windows binaries only, the idea is general enough to be extended to other OSes. The main engineering challenge is on modifying the front-end.

VI. Kernel Fuzzer Design

In this section, we present the design details of the kernel fuzzer module, which utilizes type information obtained from the static analyzer and runs type-aware fuzzing on syscall interfaces to find kernel vulnerabilities.

A. Launcher

Our kernel fuzzer runs by intercepting syscalls requested by a seed application. Specifically, it mutates syscall arguments encountered during the execution of the program. This type of kernel fuzzing technique is often referred to as hooking-based fuzzing [8], [74]. Hooking-based fuzzers can easily explore deep kernel states by means of valid syscall sequences generated from a regular program execution.

It is crucial for hooking-based fuzzers to provide proper user inputs to the seed application in order to observe various syscalls. Since most Windows applications require GUI interactions, such as clicking buttons or dragging icons, hooking-based fuzzers typically run with a proxy script that performs GUI interaction. NtFuzz has the same requirements. Therefore, we manually wrote a Python script for each seed application listed in §VII-A for our evaluation. Although such manual effort is inevitable, writing a script for GUI interaction does not require domain-specific knowledge and expertise as in template-based fuzzers that require writing harness code, e.g., kAFL [84]. In our experiment, the size of the scripts used for each application was only 29 SLoC on average.

We implement syscall hooking by directly modifying the System Service Descriptor Table (SSDT) [10]. Note that on x86-64 Windows, Kernel Patch Protection [59] prevents this hooking mechanism. Therefore, we have to use Windows debugging APIs [54] to hook syscalls in x86-64 Windows. Currently, NtFuzz targets x86 Windows, since hooking SSDT incurs less overhead than relying on debugger APIs.

B. Mutator

We address two technical challenges in designing our syscall mutator. First, our mutator should be aware of type information obtained from the static analyzer (type-aware mutation in §VI-B1). Second, our mutation should not get stuck by syscall error handlers (lazy mutation in §VI-B2).

1) Type-Aware Mutation: Our mutator changes its mutation strategy based on the type of a target syscall argument.

- **Integer types (int, long, ...).** We adopt the mutation strategies employed by AFL [99], which are proven to be effective for finding bugs. More specifically, we randomly choose one of the four operations: (1) bit flipping, (2) arithmetic mutation, (3) trying extreme values such as 0 or INT_MAX, (4) generating a completely random value. During these mutations, we consider the width of an integer, as well.
- **String types (char*, wchar_t*, ...).** We perform three different mutations at random: (1) randomly choose a character and replace it with a random character, (2) extend the string with a random string, or (3) randomly truncate the string.

Algorithm 2: Hooking-based Fuzzing Algorithm.

```
1  function Fuzz(seedApp, typeInfo)
2      cntList ← []
3      for $i$ in 1 to ITER_N do
4          proc ← Launch(seedApp)
5          cntList.Add(CountSyscall(proc))
6      avgSysCnt ← Average(cntList)
7      while true do
8          n ← RandInt(avgSysCnt)
9          proc ← Launch(seedApp)
10         MutateSyscall(proc, typeInfo, n)
```

- **Handle type (`HANDLE`).** We do not perform mutation for `HANDLE` because passing incorrect handle values often makes syscall handlers prematurely return with an error.
- **Struct types.** We simply perform type-aware fuzzing on every field in a structure.
- **Array types.** If we know the size of the target array (from our analysis in §V-C2), we simply iterate through each element of an array and perform type-aware mutation. Otherwise, we consider the target array as a singleton array, and mutate only the first element.
- **Pointer types.** We mutate both pointee value and pointer value. To mutate pointee values, we recursively follow the pointee type and perform type-aware mutation. For example, when a pointer points to a structure, we recursively traverse every field of the structure until we meet a non-pointer type and mutate them. To mutate pointer values, we use the similar strategies used for integer values.

Of course, mutating every syscall argument will likely lower the chance of finding bugs; the syscall sequences will be likely invalid. Therefore, NtFuzz controls the degree of mutation with a configuration parameter called *mutation probability*. For every chance of mutation, we decide whether to mutate the argument or not based on this probability. Let us denote the mutation probability by p. Our mutator will uniformly sample a value between 0 and 1 for every mutation candidate, and perform a mutation only when the sampled value is below p.

2) Lazy Mutation: Although hooking-based fuzzing is effective in generating meaningful syscall sequences, there is a caveat. When performing hooking-based fuzzing, the syscall handlers in kernel may return an error, and user-level code often terminates its execution upon detecting such errors. As a result, syscalls requested earlier during an execution of a seed program have more chance to be mutated, leading to a considerable bias. Error handling routines help in writing secure code, but they necessarily impede hooking-based fuzzing.

To mitigate this problem, we perform a novel mutation strategy that we call *lazy mutation*. The key intuition is to hold mutation off until a random point is reached. Algorithm 2 presents the idea. In Line 3-6, we first estimate the number of syscalls invoked by a seed application (`seedApp`) when there is no mutation. Particularly, we measure the average count of syscalls over `ITER_N` executions of `seedApp`, where `ITER_N` $= 3$ in our current implementation. In Line 7-10, we randomly choose a number between 0 and `avgSysCnt` for every iteration, which determines the number of syscalls to skip. The `MutateSyscall` function starts mutation only after the first `n` syscalls are executed.

C. Crash Detector

Unlike user-level fuzzers, kernel fuzzers should deal with system reboots because the entire system will shut off with BSoD whenever we find a kernel crash. Therefore, we configured our Windows VMs to create a memory dump when the system crashes. When the system reboots, the fuzzer will discover the crash dump file and send it to the host machine. Also, we made our fuzzer to store the recent syscall payloads in memory, so that memory dumps provide useful information to analyze and reproduce the crashes.

D. Implementation

We implemented the fuzzer's top-level logic (`Fuzz` of Algorithm 2) with 196 source lines of python code. The hooking and mutation logics (`MutateSyscall`) are implemented as a kernel driver, which is written in 2,724 source lines of C/C++ code. This driver is loaded in kernel space, and installs syscall hookers as described in §VI-A. It also identifies the syscalls invoked from the seed application, and applies the type-aware mutations described in §VI-B1.

VII. Evaluation

In this section, we address the following research questions.

RQ1 How accurate and scalable is our static analysis? How is it affected by N_{SE}? (§VII-B)

RQ2 How does the mutation ratio affect the effectiveness of kernel fuzzing? (§VII-C)

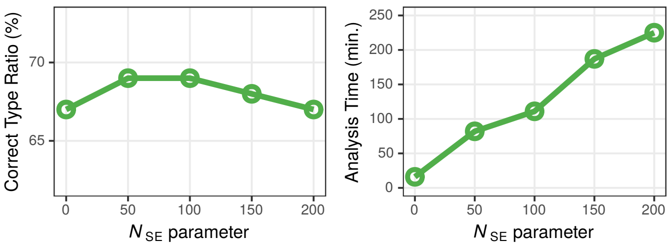
RQ3 Does the type-aware fuzzing strategy indeed help in finding more bugs? (§VII-D)

RQ4 Can NtFuzz find previously unknown bugs from the latest version of Windows? How does it compare with existing fuzzers? (§VII-E)

A. Experimental Setup

*1) **Runtime Environment**:* For the evaluation of the static analyzer (**RQ1**), we used a desktop machine with an Intel i7-6700 3.4GHz CPU and 64GB of memory. For the rest of the experiments (**RQ2–RQ4**), we used server machines by assigning two cores of Intel Xeon E5-2699 2.2GHz CPU and 4GB of memory to each VM running under VirtualBox-6.1.0 [75].

*2) **Windows Versions**:* We used two versions of x86 Windows 10 for the evaluation. For the self-evaluation with different parameters (**RQ2** and **RQ3**), we used an old Windows 10 17134.1 build, released in April 2018, because we can easily evaluate the bug-finding ability of NtFuzz with more number of confirmed bugs. For the real-world evaluation (§VII-E), we used Windows 10 18362.592 released in January 2020, which was the latest version at the time of evaluation.



(a) Accuracy of analysis. (b) Time consumption of analysis.

Fig. 10. The accuracy and speed of static analyzer with different N_{SE} values.

*3) **Seed Applications**:* Recall from §VI that NtFuzz requires a seed application to operate. Therefore, we manually collected eight user applications from various categories: AdapterWatch 1.05, Chess Titans [96], DxDiag, PowerPoint 2019 10361.20002, SpaceSniffer 1.3.0.2, SumatraPDF 3.2, Unity Sample [90], and WordPad. We used these seed applications for all our fuzzing experiments.

B. Performance of Static Analyzer

Does our static analyzer output accurate syscall type information within a reasonable amount of time? To answer this question, we ran our static analyzer on the system binaries (Table I) obtained from Windows 10 17134.1.

1) Accuracy of the Analysis: To evaluate the accuracy of our static analysis, we first had to find documented syscalls for establishing ground truth data. We collected documented syscalls from Microsoft Docs [60], [61], [64], [66], and filtered out syscalls that are not called from the system binaries. As a result, we obtained 64 syscalls and their 326 arguments as the ground truth.

Figure 10a describes how accurately our static analyzer performs. The Y-axis denotes the percentage of correctly identified syscall argument types. With N_{SE} from 50 to 100, our analyzer was able to correctly infer 69% of the syscall arguments. Recall from §V-B4 that N_{SE} parameter decides the degree of soundness of our analysis. When N_{SE} is too low, the analyzer ignores side effects and yields a lower accuracy. When N_{SE} is too high, the analyzer soundly captures side effects, but it becomes prone to over-approximation.

To further understand the inaccuracy result of our analyzer, we examined why our analyzer returns incorrect types for 31% of the cases when $N_{SE} = 50$. The most significant cause was the use of a NULL pointer. If every call site of a syscall we analyzed passes a NULL value for a pointer-type argument, we have no means to infer that type by just looking at the binary. Another major cause we found was C structures located on the stack. As discussed in §V-C1, this is another inherent limitation of binary analysis.

We note that a 69% accuracy is already high enough for the purpose of fuzzing because this means our fuzzer can perform type-aware mutation on 69% of the syscall arguments it encounters. As we show in §VII-D, our type analysis result indeed helps NtFuzz find 1.7× more unique crashes on Windows kernel.

TABLE II
BASIC STATISTICS OF OUR TARGET SYSTEM BINARIES.

Binary	Size (KB)	# Funcs	# BasicBlks	# Instrs
ntdll.dll	1,582	3,303	63,432	241,012
kernelbase.dll	1,942	2,599	52,483	193,436
kernel32.dll	622	971	19,771	73,290
win32u.dll	101	1,244	2,448	3,672
gdi32.dll	132	1,135	3,797	12,648
gdi32full.dll	1,432	1,716	36,572	140,454
user32.dll	1,496	2,012	35,833	128,480
Total	7,307	12,980	214,336	792,992

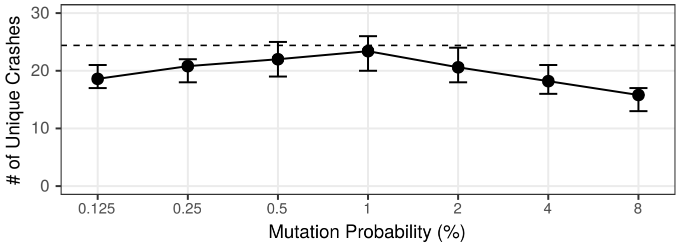


Fig. 11. The number of unique crashes found with each mutation probability.

2) Scalability of the Analysis: Although our static analysis can be considered as a pre-processing step, it should be efficient enough to handle multiple Windows system binaries. To evaluate the scalability of our static analyzer, we first computed several statistics about the system binaries that our static analyzer had to deal with. We then computed how long it took to run our analyzer on those binaries.

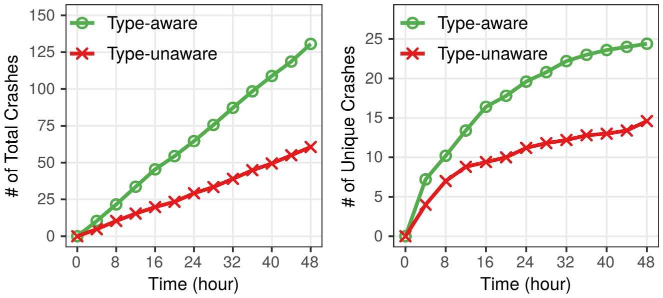
Table II summarizes the size of code analyzed by our system. Recall from §V-A that NtFuzz selectively analyzes functions that can affect syscall arguments in order to reduce the analysis cost. Still, our analyzer had to deal with the semantics of more than 12K functions and 792K instructions in total. These numbers indeed confirm the need for scalable binary analysis.

Figure 10b describes how long it takes for our analyzer to infer syscall types with each N_{SE} parameter. As N_{SE} gets higher, we naturally spend longer time on the analysis. Surprisingly, though, the total analysis time was only within a few hours for all cases. This result highlights that our design choices described in §IV-B and §V indeed enabled a scalable binary analysis for syscall type inference.

In the following fuzzing experiments, we use the type information obtained with $N_{SE} = 50$ as the experimental results imply that this configuration strikes a good balance between the scalability and accuracy of the analysis.

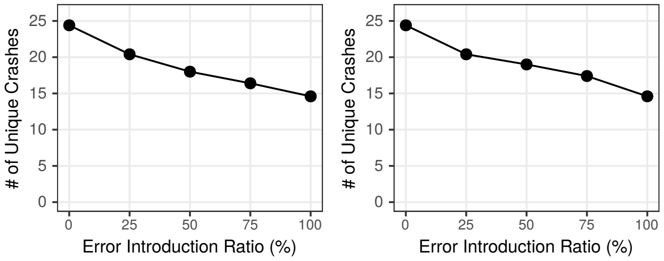
C. Deciding Mutation Parameter for Fuzzing

Previous works [16], [30] have shown that mutation configuration greatly influences the effectiveness of fuzzing. Therefore, we also evaluate the impact of mutation configuration on our system. As we described in §VI-B1, NtFuzz employs a user-configurable parameter p, the mutation probability.



(a) The number of total crashes. (b) The number of unique crashes.

Fig. 12. The number of kernel crashes found over time, respectively with type-aware fuzzing and type-unaware fuzzing.



(a) Impact of unsoundness. (b) Impact of imprecision.

Fig. 13. The impact of type inaccuracy on fuzzing effectiveness.

To observe the impact of the mutation probability, we ran NtFuzz on the Windows 10 17134.1 with seven different p values: $p = 0.01 \times 2^n$, where $n \in \{-3, -2, ..., 3\}$. For each p, we ran NtFuzz for 48 hours with each of the eight seed applications. This sums up to a total of 384 (= 48 × 8) hours of fuzzing for each p. We repeated the experiments five times and reported the average numbers with ranges in Figure 11.

When mutation probability is too low, NtFuzz has less chance to mutate syscall arguments, and thus, it finds fewer bugs. On the other hand, when the mutation probability is too high, the seed program terminates too early due to syscall errors even before it reaches a meaningful program state. In our experiments, NtFuzz with mutation probability $p = 0.01$ found the greatest number of crashes on average.

Next, we further investigated the crashes found with each mutation probability, and noticed that different p parameters report different sets of crashes. For example, although $p = 0.01$ produced the most crashes, one of the crashes we observed was found only with $p = 0.08$, but not with $p = 0.01$. This is *not* surprising, because the optimal mutation probability can be different for each different bug [16].

Based on this observation, we ran an additional experiment with *variable* mutation probability. That is, for each execution of a seed application, we randomly chose one of the seven mutation probabilities, instead of using a fixed one. The dashed line in Figure 11 presents the average number of unique crashes found with this strategy. While it found slightly more crashes than $p = 0.01$, the difference was not significant.

Nonetheless, we decided to use the variable mutation probability for the rest of the fuzzing experiments. This is to avoid potential overfitting of our system. For example, if we use a different set of seed applications, the optimal mutation parameter may change accordingly. By using this strategy, we expect our system to adapt flexibly to such changes.

D. Impact of Type Information On Fuzzing Effectiveness

We now evaluate whether our static analysis is indeed helpful for effective fuzzing. To confirm the impact of type information on NtFuzz, we ran NtFuzz and compared the number of crashes found with and without type information. For type-aware fuzzing, we simply ran NtFuzz with the type information obtained in §VII-B. For type-unaware fuzzing, we modified NtFuzz to disable our type-aware mutation, and made it simply use the integer mutation strategy only.

Figure 12 shows the number of crashes found over time. Again, we ran NtFuzz for 48 hours with each seed application, and repeated the experiments for five times. On average, NtFuzz found 130.6 crashes (with standard deviation $\sigma = 15.2$) and 24.4 unique crashes ($\sigma = 2.1$) with type information. Without type information, however, it found only 60.6 total crashes ($\sigma = 5.2$) and 14.6 unique crashes ($\sigma = 0.9$). With a Mann-Whitney U test, we conclude that type-aware fuzzing finds significantly more unique crashes than type-unaware fuzzing (p-value = 0.011).

Next, we further evaluate how the type accuracy affects the fuzzing capability. To this end, we deliberately introduce errors to the type information obtained from §VII-B, and measure the number of unique crashes found. Specifically, we emulate both unsoundness and imprecision of our type analysis as follows. To emulate the unsoundness, we randomly select syscall arguments and replace their types with integer types. To emulate the imprecision, we randomly select pointer type arguments, and inflate their contents twice. For example, when the pointee type is a structure, we replicate its fields to double the structure size.

Figure 13 presents the average number of unique crashes found with different error introduction ratios. We ran fuzzing for 48 hours per each seed application per each configuration, and repeated the experiments for five times. When we make type information unsound, the fuzzer will lose chances to mutate the contents pointed to by the arguments. On the other hand, if we make the types imprecise, the fuzzer will waste its resource in mutating irrelevant data, and even make the seed application to abort due to corrupted parameters. Figure 13 confirms that fuzzing effectiveness indeed decreases as we introduce more errors to the syscall types.

E. Real-World Bug Finding

We now discuss the practical impact of NtFuzz by evaluating it on the latest Windows 10 (see §VII-A). For this experiment, we reran the static analyzer on the new version of the system binaries to obtain the syscall types. We also manually analyzed and triaged all the bugs found.

1) Comparison Against Other Fuzzers: First, we compared the effectiveness of NtFuzz against existing Windows kernel fuzzers. Note, while there are several open-source projects,

TABLE III
LIST OF UNIQUE BUGS NTFUZZ FOUND FROM THE LATEST WINDOWS 10.

No.	Module	Description	Security Impact	Status	CVE
1	win32kfull.sys	Arbitrary memory access due to sanitization error	Privilege escalation	Fixed	CVE-2020-0792
2	ntoskrnl.exe	Use of uninitialized heap memory in kernel-space	Privilege escalation	Fixed	CVE-2020-1246
3	dxgkrnl.sys	Indirect call to arbitrary address due to memory corruption	Privilege escalation	Fixed	CVE-2020-1053
4	win32kfull.sys	Out-of-bound buffer read in kernel-space memory	Information disclosure	Fixed	CVE-2020-17004
5	ntoskrnl.exe	Invalid kernel-space memory access due to sanitization error	Denial-of-service	Confirmed	
6	win32kfull.sys	Invalid user-space memory access due to sanitization error	Denial-of-service	Confirmed	
7	ntoskrnl.exe	Termination of critical system process	Denial-of-service	Confirmed	
8	tcpip.sys	NULL pointer dereference	Denial-of-service	Confirmed	
9	-*	NULL pointer dereference	Denial-of-service	Unknown	
10	-*	Floating point error	Denial-of-service	Unknown	
11	-*	Floating point error	Denial-of-service	Unknown	

* Redacted for reponsible disclosure.

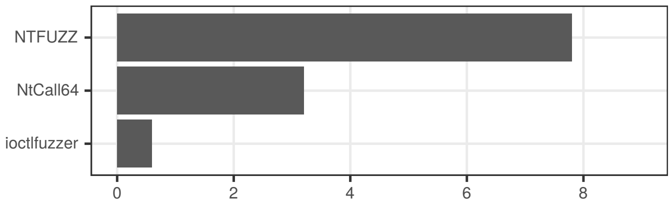


Fig. 14. Comparison of the bugs found by NTFUZZ, ioctlfuzzer, and NtCall64.

```
1  NtTraceEvent(struct S* arg) {
2    ...
3    size_t n = arg->field_0x1c;
4    char* p = arg->field_0x20;
5    if (n && (p + n > 0x7fff0000 || p + n > p)) {
6      return;
7    }
8    if (p + 4 < p + n) {
9      if (*p) { ... } // Crashes here.
10   }
11   ...
```

Fig. 15. Pseudo-code of one of the bugs found by NTFUZZ.

most of them were *not* usable. For example, kAFL [84] and pe-afl [47] require users to manually write harness code for fuzzing. Similarly, KernelFuzzer [25] requires users to manually encode generation rules. BrokenType [37] targets font parsing APIs, but these APIs do *not* execute kernel code since 2015 [53].

Syzkaller [91] has limited Windows support because **the current implementation can only fuzz API functions but not syscall functions**. This makes the comparison against Syzkaller pointless because syscall-level fuzzing is largely different from API-level fuzzing as noted in §III.

This leaves us two Windows kernel fuzzers at hand: ioctlfuzzer [74] and NtCall64 [33]. To run the fuzzers on x86 Windows 10, we had to modify their source code: 34 lines of code of ioctlfuzzer as it does not support Windows 10, and 165 lines of code of NtCall64 as it does not support x86.

We ran NTFUZZ for 48 hours per each seed application. For ioctlfuzzer, which is a hooking-based fuzzer, we used the same seed applications and fuzzing hours. For NtCall64, which is a generation-based fuzzer, we simply ran it for the same amount of time (48 × 8 hours) without the seed applications. The experiments were repeated five times.

Figure 14 presents the number of unique bugs found by each fuzzer. On average, NTFUZZ found 7.8 ($\sigma = 0.8$) unique bugs, while NtCall64 and ioctlfuzzer respectively found only 3.2 ($\sigma = 0.4$) and 0.6 ($\sigma = 0.5$) unique bugs. With Mann-Whitney U tests, we confirmed that the differences between NTFUZZ and the other fuzzers are significant (p-value = 0.009 for NtCall64 and p-value = 0.010 for ioctlfuzzer). The result makes sense as ioctlfuzzer only focuses on IOCTL syscall, and NtCall64 is type-unaware. This highlights the importance of type-aware fuzzing on general syscall interfaces.

2) Found Bugs: Next, we collected all the unique bugs found by NTFUZZ during the fuzzing experiment in §VII-E1. Table III presents the 11 new bugs found by NTFUZZ, along with short descriptions and relevant module names. To understand the security impact of each bug we found, we manually analyzed them and noted them in the table. All the bugs in the table at least had the impact of denial-of-service, which allows an unprivileged user to shut down the Windows system. Moreover, four of the bugs had a more severe security impact than denial-of-service. We note that ioctlfuzzer and NtCall64 only found denial-of-service bugs in the meantime. At the time of writing, four of the bugs found by NTFUZZ were assigned CVEs, and we won $25,000 bug bounty from Microsoft. This result indeed highlights the practical impact of NTFUZZ.

3) Case Study: We present a case study on one of the bugs we found to show how the design of NTFUZZ helped in finding a new bug. In Figure 15, we provide the pseudo-code of the syscall handler related to the fifth entry of Table III. We simplified the code for ease of explanation.

This bug is caused by an error in the pointer sanitization logic. First, an attacker provides a pointer to a C structure as an argument to the syscall. This structure carries a buffer pointer (`field_0x20`) as well as the size of the buffer (`field_0x1c`). The kernel code sanitizes these two fields at Line 5, to ensure that `p` does not point to a kernel-space address. However, one can bypass this check when `n` is zero.

Therefore, if we provide `n = 0` and `p = UINT_MAX` as input, the kernel will crash at Line 9, while accessing an invalid pointer value `UINT_MAX`.

Thanks to the syscall type information inferred by the static analyzer, NtFuzz knows that `arg` is a pointer to a structure of `0x28`-byte size, which allows it to perform type-aware mutations on each field of the structure. Therefore, NtFuzz was able to trigger this bug while mutating the fields of the structure, particularly at the offset `0x1c` and `0x20` with extreme values (see §VI-B1). Note, without such type information, NtFuzz would suffer to trigger this bug.

VIII. Discussion

Scope of Static Analysis. Our static analysis infers syscall types by analyzing carefully chosen system binaries listed in §II-A. Although those binaries cover a significant portion of syscalls, we can potentially extend the number of target system binaries to analyze more syscalls. An alternative solution is to analyze the kernel code to figure out syscall types, which is beyond the scope of this paper.

Generation-based and Coverage-based Fuzzing. While hooking-based fuzzing has its advantage, generation-based fuzzing can greatly reduce the dependency on seed applications. Since generation-based fuzzing can also benefit from syscall type information, one may adopt our analyzer to a generation-based fuzzer. Another promising direction is to leverage code coverage feedback to effectively evolve test cases, as in [11], [17], [52], [76], [84], [91], [95].

Union Type Handling. Windows syscalls can accept a *union type* argument, which can be interpreted as different types according to the context (e.g. IOCTL). We leave it as future work to extend our system to support union types. Currently, NtFuzz can support a certain degree of flexibility by handling such types as an array whose size is specified by a different argument (see §V-C).

IX. Related Work

Since the early black-box fuzzing work [35], [43], [44], kernel fuzzing has lately evolved to various extents. Many kernel fuzzers adopt grey-box fuzzing with coverage feedback [47], [51], [70], [84], [91], [97], or utilize knowledge about the syscall dependencies [25], [30], [91].

Static analysis has been another popular technique for finding kernel bugs. Dr.Checker [50] aimed at making an effective trade-off between soundness and precision to develop a practical bug finder. K-Miner [27] proposed a method to partition kernel code into relevant segments to enhance its analysis capabilities. Other tools [93], [94] aim at finding a specific class of bug. All of them attempt to analyze kernel source code to find bugs, while our goal is on improving fuzzing effectiveness with analyzed type information.

There is another line of work [4], [12], [29], which leverages static analysis to enhance software testing. SymDrive [81] relies on static analysis to decide efficient path scheduling for symbolic execution. DEADLINE [98] statically analyzes the memory access of kernel code to collect candidates for double-fetch bug, and then checks these points with symbolic execution. Razzer [34] uses static analysis to spot potential data race points, and then runs hypervisor-assisted fuzzing on these points. Moonshine [76] syntactically analyzes memory access of kernel code and find implicit dependencies between syscalls. This information is used to minimize the sequence of seed syscalls. DIFUZE [18] leverages static analysis to infer the type of syscall interface. While DIFUZE analyzes kernel code to infer the types of `IOCTL` handlers, we analyze user-space code and infer the types of all the observable syscalls. All the above fuzzers report successful integration of static analysis with kernel testing, but none of them runs on binary code. To the best of our knowledge, our work is the first to use static binary analysis to enhance COTS OS fuzzing.

There has been plenty of research on binary-level type inference [14], but only a few of them consider memory access [68]. DIVINE [7] and TIE [45] utilize VSA for type inference, but their scalability to large binaries have not been confirmed. Notably, SecondWrite [24] achieves scalability by choosing a flow- and context-insensitive analysis. In contrast, we do *not* sacrifice the sensitivities to obtain more precise syscall types. Meanwhile, types in binary code can be also inferred with dynamic analysis [49], [87], which is complementary to our work. While dynamic analysis does not suffer from over-approximation as it traces a single execution path, static analysis can benefit from observing the code that is not executed in the runtime. For example, we can aggregate type information from the call sites not executed by the seed application (§V-C3), or analyze the data access pattern of the whole function to infer stack structure (§V-C1).

X. Conclusion

In this paper, we presented NtFuzz, a type-aware Windows kernel fuzzer. Figuring out syscall types is challenging for Windows due to its closed nature and large syscall interface. Therefore, NtFuzz analyzes Windows system binaries to fathom what type of arguments are used to invoke syscalls. To the best of our knowledge, NtFuzz is the first system-wide binary analyzer for Windows, which is both inter-procedural and context-sensitive. We evaluated NtFuzz on the latest Windows kernel and found 11 unique kernel bugs, including four CVEs. Our effort has been well appreciated by the industry: currently, we have earned $25,000 of bug bounty.

Acknowledgement

We thank anonymous reviewers and our shepherd Brendan Dolan-Gavitt for their comments and suggestions. We also thank Minyeop Choi and DongHyeon Oh for their help during the background study. This work was partly supported by (1) Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security), (2) the "Binary-level analysis and vulnerability discovery on IoT" grant funded by the LG Electronics, and (3) the Office of Naval Research grant N00014-18-1-2024.

APPENDIX A
JOIN OPERATION FOR ABSTRACT DOMAIN

This section describes the join operation of our abstract domain presented in Figure 6.

First, we define the join operation for the abstract integer domain ($\mathbb{I}$) as follows. Join operation with $\bot$ or $\top$ is trivial. For non-symbolic integers, join is performed as in the flat integer domain. For symbolic integers, the two integers should exactly match each other in order to spawn a non-$\top$ integer.

$$
\begin{aligned}
i \sqcup \bot &= i, \bot \sqcup i = i \\
i \sqcup \top &= \top, \top \sqcup i = \top \\
(a_1 s_1 + b_1) \sqcup (a_2 s_2 + b_2) &= \begin{cases} b_1 & (a_1 = a_2 = 0, b_1 = b_2) \\ a_1 s_1 + b_1 & (a_1 = a_2, s_1 = s_2, b_1 = b_2) \\ \top & (\texttt{otherwise}) \end{cases}
\end{aligned}
$$

Now we define the join of abstract values ($\mathbb{V}$), abstract memories ($\mathbb{M}$), and abstract states ($\mathbb{S}$). We use the standard join operations [67], [82] for product domain ($\mathbb{V}$, $\mathbb{S}$), power set domain ($2^{\mathbb{L}}$, $2^{\mathbb{T}}$), and map domain ($\mathbb{M}$). Since the register map domain ($\mathbb{R}$) has the same join operation with abstract memory domain, it is omitted below.

$$
\begin{aligned}
v_1 \sqcup v_2 &= \langle v_1[0] \sqcup v_2[0], v_1[1] \cup v_2[1], v_1[2] \cup v_2[2] \rangle \\
m_1 \sqcup m_2 &= \bot_m[k_1 \mapsto m_1(k_1) \sqcup m_2(k_1)] ... [k_n \mapsto m_1(k_n) \sqcup m_2(k_n)] \\
& \quad \texttt{where } k_i \texttt{ is contained in } m_1 \texttt{ or } m_2 \\
s_1 \sqcup s_2 &= \langle s_1[0] \sqcup s_2[0], s_1[1] \sqcup s_2[1] \rangle
\end{aligned}
$$

APPENDIX B
DETAILS OF ABSTRACT SEMANTICS

A. Binary Operation

In this section, we present the formal definition of binary operation semantics (*binop*) we described in §V-B2. For simplicity, we present the semantics of the two representative binary operations, addition and multiplication. We note that the semantics of subtraction is defined similarly to addition, and other operations like logical AND are defined similarly to multiplication.

First, we define binary operations for abstract integers. If the result cannot be represented in a linear expression of a single symbol, we conservatively return $\top$.

$$
\begin{aligned}
i \mathbin{\hat{+}} \bot &= \bot, \bot \mathbin{\hat{+}} i = \bot \\
i \mathbin{\hat{+}} \top &= \top, \top \mathbin{\hat{+}} i = \top \\
(a_1 s_1 + b_1) \mathbin{\hat{+}} (a_2 s_2 + b_2) &= \begin{cases} (b_1 + b_2) & (a_1 = a_2 = 0) \\ a_1 s_1 + (b_1 + b_2) & (a_2 = 0) \\ a_2 s_2 + (b_1 + b_2) & (a_1 = 0) \\ (a_1 + a_2) s_1 + (b_1 + b_2) & (s_1 = s_2) \\ \top & (\texttt{otherwise}) \end{cases} \\
i \mathbin{\hat{\times}} \bot &= \bot, \bot \mathbin{\hat{\times}} i = \bot \\
i \mathbin{\hat{\times}} \top &= \top, \top \mathbin{\hat{\times}} i = \top \\
(a_1 s_1 + b_1) \mathbin{\hat{\times}} (a_2 s_2 + b_2) &= \begin{cases} b_1 b_2 & (a_1 = a_2 = 0) \\ (a_1 b_2) s_1 + b_1 b_2 & (a_2 = 0) \\ (a_2 b_1) s_2 + b_1 b_2 & (a_1 = 0) \\ \top & (\texttt{otherwise}) \end{cases}
\end{aligned}
$$

Next, we define an operation to add an offset to an abstract location. We use the same notation $\hat{+}$ again here, as we can decide which definition to use by looking at the operand types.

$$
\begin{aligned}
\texttt{Global}(n_1) \mathbin{\hat{+}} n_2 &= \texttt{Global}(n_1 + n_2) \\
\texttt{Stack}(f, n_1) \mathbin{\hat{+}} n_2 &= \texttt{Stack}(f, n_1 + n_2) \\
\texttt{Heap}(a, n_1) \mathbin{\hat{+}} n_2 &= \texttt{Heap}(a, n_1 + n_2) \\
\texttt{SymLoc}(s, n_1) \mathbin{\hat{+}} n_2 &= \texttt{SymLoc}(s, n_1 + n_2)
\end{aligned}
$$

Finally, we define the binary operation semantics (*binop*) as follows. As we described in §V-B2, we trace only the constant offsets and ignore array index terms. This is achieved by considering only the syntactic constants during the addition to abstract locations. Also, note that multiplication results in an empty location set, and generates an integer type constraint. In the actual implementation, we also consider the width of the integer types, which is not shown here for the conciseness.

$$
\begin{aligned}
\mathit{binop}(+, e_1, e_2, S) &= \mathit{add}(e_1, e_2, S) \\
\mathit{binop}(\times, e_1, e_2, S) &= \mathit{mul}(e_1, e_2, S) \\
& ... \\
\mathit{add}(e, n, S) &= \langle v[0] \mathbin{\hat{+}} n, \{ l \mathbin{\hat{+}} n | l \in v[1] \}, \phi \rangle \\
& \quad \texttt{where } v = \mathcal{V}(e)(S) \\
\mathit{add}(n, e, S) &= \mathit{add}(e, n, s) \\
\mathit{add}(e_1, e_2, S) &= \langle v_1[0] \mathbin{\hat{+}} v_2[0], v_1[1] \cup v_2[1], \phi \rangle \\
& \quad \texttt{where } v_i = \mathcal{V}(e_i)(S) \\
\mathit{mul}(e_1, e_2, S) &= \langle v_1[0] \mathbin{\hat{\times}} v_2[0], \phi, \{ \mathit{integer} \} \rangle \\
& \quad \texttt{where } v_i = \mathcal{V}(e_i)(S)
\end{aligned}
$$

B. Side Effect Application

In this section, we present the formal definition of *apply* function described in §V-B2.

First, we define a side effect as a pair of an *argument map* and an *update set*. The argument map is a mapping from an argument index to its corresponding symbolic value. Meanwhile, the update set captures the pairs of an updated location and its updated value. Recall from §V-B4 that N_{SE} limits the size of this set for scalability.

$$
\begin{aligned}
\texttt{(Argument Map) } \mathbb{A} &= \mathbb{Z} \to \mathbb{V} \\
\texttt{(Update Set) } \mathbb{U} &= 2^{\mathbb{L} \times \mathbb{V}} \\
\texttt{(Side Effect) } \Delta &= \mathbb{A} \times \mathbb{U}
\end{aligned}
$$

For instance, let us assume that function `f` updates its first argument location with zero (i.e. "`*arg1 = 0`"). Then, the argument map of `f` will record that its first argument was initialized as $\langle s_1, \{\texttt{SymLoc}(s_2, 0)\}, \{\texttt{SymTyp}(s_3)\} \rangle$, and its update set will be expressed as $\{ \langle \texttt{SymLoc}(s_2, 0), \langle 0, \phi, \phi \rangle \rangle \}$.

To apply side effect $\delta = \langle A, U \rangle$ to state S, we should first construct a substitution Γ, by matching each $A(i)$ with $\mathit{getArg}(S, i)$, where i is an argument index contained in A.

Here, function *getArg* obtains the $(i+1)$-th argument value from the state S, according to the ABI specification.

Let us resume the previous example, and assume that a new function `g` calls `f` with $\langle \bot, \{\texttt{Stack}(\texttt{g}, -40)\}, \phi\rangle$ as the first argument. That is, `g` passes its local variable address as an argument to `f`. Then, the substitution Γ is constructed by matching $\langle s_1, \{\texttt{SymLoc}(s_2, 0)\}, \{\texttt{SymTyp}(s_3)\}$ with this value. As a result, Γ will contain a mapping that substitutes symbolic location $\texttt{SymLoc}(s_2, 0)$ into $\texttt{Stack}(\texttt{g}, -40)$.

With the constructed substitution Γ, we can finally define *apply* as follows. We first substitute the update entries in U and accumulate these instantiated updates to the memory.

$$apply(\delta, S) = \langle R, update(\Gamma(l_n))(\Gamma(v_n))(...update(\Gamma(l_1))(\Gamma(v_1))(M)...)\rangle$$
$$\texttt{where}\ \ \delta = \langle A, U\rangle, S = \langle R, M\rangle, \langle l_i, v_i\rangle \in U$$

In the actual implementation, a side effect includes additional information like the return value of a function, which is omitted here. It is straightforward to extend the definition of a side effect in such a way.

C. IR-level Example

Now we illustrate how the abstract semantics is actually applied on an IR-level example in Figure 16. This code snippet corresponds to Line 4 of our high-level example in Figure 4. The function call `h(p,x)` in C code is translated into the IR statements in Figure 16. We first prepare argument `x` (Line 1-3) and argument `p` (Line 4-5), and then call `h` (Line 6).

We will assume that `esp` is initialized to have a location $\texttt{Stack}(\texttt{f}, -20)$ at Line 1. Meanwhile, `ebp` will contain the initial value of `esp` at the function prologue[2], namely $\texttt{Stack}(\texttt{f}, -4)$. Also, let us assume that the heap location $\texttt{Heap}(a, 0)$ returned from `malloc` is contained in `esi`, where a is the allocation site.

For the `Put` statement in Line 1, we first evaluate the memory load expression, `[ebp+8]`. According to the binary operation semantics in Appendix B-A, we add 8 to the offset of the abstract location in `ebp`, obtaining $\texttt{Stack}(\texttt{f}, 4)$ as a singleton location to load from. Since this location corresponds to the argument of `f`, load from this location returns a symbolic value, such as $\langle s_1, \{\texttt{SymLoc}(s_2, 0)\}, \{\texttt{SymTyp}(s_3)\}\rangle$. For conciseness, let us name this value as v_x. Based on the semantics of `Put` in Figure 7b, this value is assigned to `ebx`.

In Line 2, we first apply *update* in Figure 7b, using location $\texttt{Stack}(\texttt{f}, -20)$ and the symbolic value in `ebx`. Then, we assign $\texttt{Stack}(\texttt{f}, -24)$ to `esp` in Line 3. Next, in Line 4, we update location $\texttt{Stack}(\texttt{f}, -24)$ with the value of `esi`, which is $\langle \bot, \{\texttt{Heap}(a, 0)\}, \phi\rangle$. Lastly, we assign $\texttt{Stack}(\texttt{f}, -28)$ to `esp` at Line 5.

```
1  Put(ebx, [ebp+8])
2  Store(esp, ebx)
3  Put(esp, esp-4)
4  Store(esp, esi)   # esi: return of malloc()
5  Put(esp, esp-4)
6  Call(g)
```

Fig. 16. IR-level example code.

At Line 6, we finally call a summarized function, `h`. Recall from §IV-C that the side effect of `h(a, b)` was summarized as "`*a = b`". As we explained in Appendix B-B, this can be represented as an update set $U = \{\langle \texttt{SymLoc}(\alpha_2, 0), v_b\rangle\}$, where an argument map A records that `a` is initialized as $v_a = \langle \alpha_1, \{\texttt{SymLoc}(\alpha_2, 0)\}, \{\texttt{SymTyp}(\alpha_3)\}\rangle$, whereas `b` is initialized as $v_b = \langle \beta_1, \{\texttt{SymLoc}(\beta_2, 0)\}, \{\texttt{SymTyp}(\beta_3)\}\rangle$.

To apply this side effect, we first have to construct a substitution Γ. As we described in Appendix B-B, we match v_a with the argument $\langle \bot, \{\texttt{Heap}(a, 0)\}, \phi\rangle$, and v_b with the argument v_x. Consequently, we obtain Γ that substitutes $\texttt{SymLoc}(\alpha_2, 0)$ with $\{\texttt{Heap}(a, 0)\}$ and v_b with v_x. By applying the substituted update entries, Line 6 will update location $\texttt{Heap}(a, 0)$ with v_x, which is $\langle s_1, \{\texttt{SymLoc}(s_2, 0)\}, \{\texttt{SymTyp}(s_3)\}\rangle$.

Appendix C
VSA and Our Modular Analysis

Since our analysis should initialize function arguments with symbolic values, the abstract domain has to deal with symbolic locations. In VSA, such a symbolic location has the form of $\texttt{SymLoc}(s, [\alpha, \beta])$, where s is a symbolic pointer, and $[\alpha, \beta]$ is a symbolic offset.

Suppose we perform a memory update on the above symbolic location. The first plausible strategy is to soundly coalesce all possible locations into one. This means we consider the symbolic pointer as an access to $\texttt{SymLoc}(s, *)$, where $*$ means any offsets. All the updates with this symbolic pointer will then be accumulated to a single location, but this will make the analysis too imprecise.

Another strategy is to unsoundly ignore the update when the offset has symbolic boundaries. While this strategy can mitigate the imprecision problem, it can lose interesting data flows. For example, any memory accesses using a pointer argument will be ignored in the analysis as an argument is always initialized to have symbolic boundaries.

The last alternative is to split the abstract memory based on the given symbolic boundaries. That is, we subdivide the memory into three: $\texttt{SymLoc}(s, [-\infty, \alpha])$, $\texttt{SymLoc}(s, [\alpha, \beta])$, and $\texttt{SymLoc}(s, [\beta, \infty])$. However, when there is a subsequent update with another location $\texttt{SymLoc}(s, [\gamma, \delta])$, we must consider the overlap between the two symbolic offsets $[\alpha, \beta]$ and $[\gamma, \delta]$. Thus, the memory state has to grow exponentially to the number of memory updates.

For these reasons, it is not straightforward to integrate modular analysis with the VSA domain. While this is an interesting challenge, we leave it as future work. Instead, we enable modular analysis for binary code by designing a novel abstract domain that we describe in §V-B1.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[2] Precisely speaking, after the first instruction (`push ebp`) of the prologue.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, 2007, pp. 43–48.

[3] Apple Inc., "Apple security bounty," https://developer.apple.com/security-bounty/.

[4] D. Babic, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[5] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "CodeSurfer/x86—a platform for analyzing x86 executables," in *Proceedings of the International Conference on Compiler Construction*, 2005, pp. 250–254.

[6] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of the International Conference on Compiler Construction*, 2004, pp. 5–23.

[7] ——, "DIVINE: Discovering variables in executables," in *Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2007, pp. 1–28.

[8] I. Beer, "pwn4fun spring 2014–safari–part ii," http://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html, 2014.

[9] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2003, pp. 196–207.

[10] B. Blunden, *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Publishers, 2012.

[11] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2020, pp. 678–689.

[12] F. Brown, D. Stefan, and D. Engler, "Sys: A static/symbolic tool for finding good bugs in good (browser) code," in *Proceedings of the USENIX Security Symposium*, 2020, pp. 199–216.

[13] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the International Conference on Computer Aided Verification*, 2011, pp. 463–469.

[14] J. Caballero and Z. Lin, "Type inference on executables," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1–35, 2016.

[15] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *Proceedings of the NASA Formal Methods Symposium*, 2015, pp. 3–11.

[16] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 725–741.

[17] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 736–747.

[18] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[19] P. Cousot, "Types as abstract interpretations," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1997, pp. 316–331.

[20] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1977, pp. 238–252.

[21] ——, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, 1992.

[22] I. Dillig, T. Dillig, and A. Aiken, "Fluid updates: Beyond strong vs. weak updates," in *Proceedings of the ACM European Conference on Programming Languages and Systems*, 2010, pp. 246–266.

[23] A. Djoudi and S. Bardin, "BINSEC: Binary code analysis with low-level regions," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 212–217.

[24] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2013, pp. 51–60.

[25] F-Secure LABS, "KernelFuzzer," https://github.com/FSecureLABS/KernelFuzzer.

[26] Facebook, Inc., "Infer," https://github.com/facebook/infer/tree/master/infer.

[27] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-Miner: Uncovering memory corruption in linux," in *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[28] Google LLC, "BinNavi," https://github.com/google/binnavi.

[29] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 49–64.

[30] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 2345–2358.

[31] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proceedings of the International Conference on Software Engineering*, 2017, pp. 519–529.

[32] Hex-Rays SA., "IDA Pro," https://www.hex-rays.com/products/ida/.

[33] hfiref0x, "NtCall64," https://github.com/hfiref0x/NtCall64.

[34] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019, pp. 754–768.

[35] D. Jones, "Trinity," https://github.com/kernelslacker/trinity.

[36] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[37] M. Jurczyk, "BrokenType," https://github.com/googleprojectzero/BrokenType.

[38] ——, "Windows system call tables updated, refreshed and reworked," https://j00ru.vexillium.org/2016/08/windows-system-call-tables-updated-refreshed-and-reworked/.

[39] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Computing Surveys*, vol. 49, no. 2, pp. 1–47, 2016.

[40] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the linux kernel," in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[41] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical concolic testing techniques for COTS operating systems," in *Proceedings of the USENIX Annual Technical Conference*, 2017, pp. 689–701.

[42] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proceedings of the International Conference on Computer Aided Verification*, 2008, pp. 423–427.

[43] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proceedings of the Symposium on Reliable Distributed Systems*, 1997, pp. 72–79.

[44] T. Le, "tsys," http://groups.google.com/groups?q=syscall+crashme&hl=en&lr=&ie=UTF-8&selm=1991Sep20.232550.5013%40smsc.sony.com&rnum=1, 1991.

[45] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium*, 2011, pp. 251–268.

[46] W. Lee, W. Lee, and K. Yi, "Sound non-statistical clustering of static analysis alarms," in *Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2012, pp. 299–314.

[47] L. Leong, "Make static instrumentation great again: High performance fuzzing for Windows system," in *BlueHat*, 2019.

[48] M. Li, "Active fuzzing as complementary for passive fuzzing," in *PacSec*, 2016.

[49] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[50] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr. Chekcer: A soundy analysis for linux kernel drivers," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 1007–1024.

[51] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[52] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the International Conference on Software Engineering*, 2020, pp. 1024–1036.

[53] Microsoft Corporation, "ADV200006: Type 1 font parsing remote code execution vulnerability," https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/adv200006.

[54] ——, "Debugging reference," https://docs.microsoft.com/en-us/windows/win32/debug/debugging-reference.

[55] ——, "Dynamic-link libraries," https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries.

[56] ——, "Inside native applications," https://docs.microsoft.com/en-us/sysinternals/learn/inside-native-applications.

[57] ——, "Microsoft bug bounty program," https://www.microsoft.com/en-us/msrc/bounty.

[58] ——, "Microsoft docs," https://docs.microsoft.com/.

[59] ——, "Microsoft security advisory 932596," https://docs.microsoft.com/en-us/security-updates/securityadvisories/2007/932596.

[60] ——, "Ntddk.h," https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/.

[61] ——, "Ntifs.h," https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/.

[62] ——, "Programming reference for the Win32 API," https://docs.microsoft.com/en-us/windows/win32/api/.

[63] ——, "Understanding SAL," https://docs.microsoft.com/en-us/cpp/code-quality/understanding-sal?view=vs-2019.

[64] ——, "Wdm.h," https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/.

[65] ——, "Windows 10 software development kit," https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk/.

[66] ——, "Winternl.h," https://docs.microsoft.com/en-us/windows/win32/api/winternl/.

[67] A. Møller and M. I. Schwartzbach, "Static program analysis," https://cs.au.dk/ amoeller/spa/, 2019.

[68] A. Mycroft, "Type-based decompilation," in *Proceedings of the ACM European Conference on Programming Languages and Systems*, 1999, pp. 208–223.

[69] National Security Agency, "Ghidra," https://github.com/NationalSecurityAgency/ghidra.

[70] NCC Group, "Triforce linux syscall fuzzer," https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[71] H. Oh, "Large spurious cycle in global static analyses and its algorithmic mitigation," in *Proceedings of the Asian Symposium on Programming Languages and Systems*, 2009, pp. 14–29.

[72] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and implementation of sparse global analyses for c-like languages," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2012, pp. 229–238.

[73] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective context-sensitivity guided by impact pre-analysis," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2014, pp. 475–484.

[74] D. Oleksiuk, "Ioctl fuzzer," https://github.com/Cr4sh/ioctlfuzzer, 2009.

[75] Oracle, "VirtualBox," https://www.virtualbox.org/.

[76] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 729–743.

[77] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 149–165.

[78] A. Plaskett, "OSXFuzz," https://github.com/FSecureLABS/OSXFuzz.

[79] ReactOS Team, "ReactOS," https://reactos.org/.

[80] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 861–875.

[81] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing drivers without devices," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2012, pp. 279–292.

[82] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.

[83] ROPAS Lab, "Sparrow," https://github.com/ropas/sparrow.

[84] 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.

[85] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, Carnegie Mellon University, 1991.

[86] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2016, pp. 138–157.

[87] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[88] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis, "Set-based pre-processing for points-to analysis," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 253–270.

[89] D. Syme, "FsLexYacc," https://github.com/fsprojects/FsLexYacc.

[90] Unity Technologies, "Adventure sample game," https://assetstore.unity.com/packages/essentials/tutorial-projects/adventure-sample-game-76216.

[91] D. Vyukov, "syzkaller," https://github.com/google/syzkaller.

[92] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[93] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 1899–1913.

[94] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2012, pp. 163–177.

[95] D. Weston, "Keeping windows secure," in *BlueHat*, 2020.

[96] Winaero, "Windows 7 games for windows 10," https://winaero.com/blog/get-windows-7-games-for-windows-10/.

[97] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRACE: Data race fuzzing for kernel file systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2020, pp. 1643–1660.

[98] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018, pp. 661–678.

[99] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.