Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay

Mengchen Cao*
mengchen.cmc@alibaba-inc.com
Orion Security Lab, Alibaba Group

Xiantong Hou*
xiantong.houxianto@alibaba-inc.com
Orion Security Lab, Alibaba Group

Tao Wang*
wt124724@alibaba-inc.com
Orion Security Lab, Alibaba Group

Hunter Qu
fuping.qfp@alibaba-inc.com
Orion Security Lab, Alibaba Group

Yajin Zhou†
yajin_zhou@zju.edu.cn
School of Cyber Science and Technology, Zhejiang University

Xiaolong Bai
baiqiu.bxl@alibaba-inc.com
Orion Security Lab, Alibaba Group

Fuwei Wang
fuwei.wfw@alibaba-inc.com
Orion Security Lab, Alibaba Group

ABSTRACT

The use of uninitialized variables is a common issue. It could cause kernel information leak, which defeats the widely deployed security defense, i.e., kernel address space layout randomization (KASLR). Though a recent system called `Bochspwn Reloaded` reported multiple memory leaks in Windows kernels, how to effectively detect this issue is still largely behind.

In this paper, we propose a new technique, i.e., *differential replay*, that could effectively detect the use of uninitialized variables. Specifically, it records and replays a program's execution in multiple instances. One instance is with the vanilla memory, the other one changes (or poisons) values of variables allocated from the stack and the heap. Then it compares program states to find references to uninitialized variables. The idea is that if a variable is properly initialized, it will overwrite the poisoned value and program states in two running instances should be the same. After detecting the differences, our system leverages the *symbolic taint analysis* to further identify the location where the variable was allocated. This helps us to identify the root cause and facilitate the development of real exploits. We have implemented a prototype called `TimePlayer`. After applying it to both Windows 7 and Windows 10 kernels (x86/x64), it successfully identified 34 new issues and another 85 ones that had been patched (some of them were publicly unknown.) Among 34 new issues, 17 of them have been confirmed as zero-day vulnerabilities by Microsoft.

*Authors contributed equally to this research.
†Corresponding author.

ACM Reference Format:
Mengchen Cao, Xiantong Hou, Tao Wang, Hunter Qu, Yajin Zhou, Xiaolong Bai, and Fuwei Wang. 2019. Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3345654

1 INTRODUCTION

In modern operating systems such as Windows and Linux, values of variables are usually undetermined until being explicitly initialized. These uninitialized variables could compromise the security of a system, especially when they are crossing different privilege domains. For instance, if a variable holds the address of a kernel object and flows into user space, the kernel address will leak to the (untrusted) user program. This defeats the widely deployed kernel address space layout randomization (KASLR) mechanism. A real-world example is a kernel exploit found in July 2015 [1], which takes advantage of a kernel data leak from the heap (CVE-2015-2433) to get the randomized base address of the win32k.sys driver, and then uses it to exploit another vulnerability to escalate its privilege. Previous research has shown that the use of uninitialized data [28, 34, 40, 48, 62] is among the most severe vulnerabilities in C and C++ language, and it accounts for more than 1/3 (147) of all memory disclosure CVEs (388) from 2000 to 2015 [33]. What's worse, a recent research has demonstrated the way to automatically perform privilege escalation attack, by exploiting uninitialized variables [35].

Unfortunately, the detection of such an issue is not easy, since it generally does not cause a crash or other perceivable effects. One possible way is using dynamic taint analysis to track the flow of variables inside the system. Specifically, it sets the newly allocated memory regions from the stack and the heap as `taint sources`, and propagates taint tags. Tags will be removed if tainted memory regions are being written with new values (being initialized). If a tainted memory region is referred by the program, a use of uninitialized variables is detected.

Though the proposed method can work, it has a serious limitation in practice. Specifically, leveraging the dynamic taint analysis in a full system software stack, including the kernel, system components and user programs, will inevitably introduce a high performance overhead. As such, Bochspwn Reloaded [1] [28], the state-of-the-art tool that focuses on kernel memory disclosure bugs, only propagates taint tags for specific instructions to reduce performance overhead. However, it introduces false negatives. As stated in the paper, "this means that every time data in memory is copied indirectly through a register, the taint information is discarded in the process, potentially yielding undesired false-negatives."

Our approach In this paper, we propose a method with two key techniques to detect *kernel information leak* due to the use of uninitialized variables. Specifically, we leverage the first key technique, i.e., *differential replay*, to quickly spot the use of uninitialized variables, without performing time-consuming full system dynamic taint analysis. Then we use the second key technique, i.e., *symbolic taint analysis*, to determine locations where uninitialized variables were allocated.

First of all, our system adopts differential replay to record the execution of kernels and user programs. Then we replay two different instances, with vanilla and poisoned values of the stack and heap memory, respectively. We compare differences of program states, e.g., addresses and contents of memory operation instructions. As poisoned variables will be overwritten during initialization, the two instances should be exactly the same if newly allocated variables have been properly initialized. In other words, the difference in program states could indicate the occurrence of an uninitialized variable is being used.

After that, our system will conduct a *symbolic taint analysis* to determine the *exact* location where the variable was allocated. To this end, our system performs offline taint analysis on the recorded execution trace. It sets the new variable as the taint source, and allocates a symbolic value for the variable. Then it propagates the symbolic taint tag and generates symbolic expressions along the trace. This process stops until reaching the instruction that uses the uninitialized variable. Finally, we obtain the complete symbolic expression of the variable. By using such an expression, an analyst is able to determine the location whether the variable was allocated. To speed up this process, symbolic expressions are packed if necessary.

Note that a recent system kMVX [70] uses the concept of multi-variant execution (MVE) to detect the kernel information leaks in Linux. The idea of MVE is similar with differential replay by executing multiple instances. However, it requires the kernel source code, and thus cannot be applied to closed-source Windows kernels. On the contrast, `TimePlayer` is a non-intrusive system that can work towards closed-source systems. Its effectiveness has been demonstrated by detecting new vulnerabilities in both Windows 7 and Windows 10 kernels.

Prototype and evaluation We have implemented a prototype system called `TimePlayer`. The differential replay is implemented based on the PANDA system [18], and the *symbolic taint analysis* leverages the SimuVEX [59] library. In order to evaluate its effectiveness, we applied it to both Windows 7 and Windows 10 kernels in a period of seven months. It successfully detected 34 issues of information leak from kernel space to user space. Among them, 17 have been confirmed by the Microsoft Security Response Center with CVE numbers. For the remaining 17 ones, at the time of writing this paper (August 2019), we are still collaborating with them to assess potential security consequences.

To further evaluate the capability of our system to detect known vulnerabilities, we used public test cases (exploits) released by Google Project Zero in our system. The result is encouraging. We have detected 85 vulnerabilities in Windows 7 and Windows 10. Among them, 55 are publicly known vulnerabilities with CVE numbers. However, there exist 30 ones that do not have CVE numbers. Our manual analysis confirmed that they are indeed kernel information leaks.

To evaluate the efficiency of differential replay used by `TimePlayer`, we implemented a reference system that purely leverages the taint analysis to track uninitialized variables. We ran the same test cases and logged the time when the kernel information leak is detected. The result shows that our system can detect 34 new issues in around 47 hours, while the reference system can only detect 7 of them in around 66 hours. This demonstrated the efficiency of `TimePlayer` to detect new vulnerabilities.

In summary, this paper makes the following contributions.

- We propose a technique called differential replay, which can quickly detect the use of uninitialized variables in Windows kernels without the need of the source code.
- We propose symbolic taint analysis to locate the sources of uninitialized variables, and present two optimizations to speed up this process.
- We have implemented a prototype and applied it to both Windows 7 and Windows 10 in a time period of seven months. It reported 34 new issues, and 17 of them have been confirmed as zero-day vulnerabilities by Microsoft.

To engage the community, we have released the test cases, recorded program traces, differences of program states during the replay that leads to the discovery of new vulnerabilities in the following link: https://github.com/AlibabaOrionSecurityLab/TimePlayer.

The rest of this paper is structured as follows: we introduce the background and a motivating example in Section 2, and illustrate the overall design of our system in Section 3. We illustrate the two key techniques of our system in Section 4 and Section 5, respectively. We then present the evaluation result in Section 6 and discuss the potential limitation of our work in Section 7. At last, we describe the related work in Section 8 and conclude our work in Section 9.

2 BACKGROUND AND A MOTIVATING EXAMPLE

2.1 Background

Deterministic record and replay of PANDA PANDA is an open source dynamic analysis platform with many unique features, which make it a powerful platform for analyzing complicated software. Our system leverages the record and replay feature offered by

[1] In this paper, if not otherwise specified, we use the name Bochspwn to denote the latest version of the tool, i.e., Bochspwn Reloaded that focuses on the kernel memory disclosure detection.

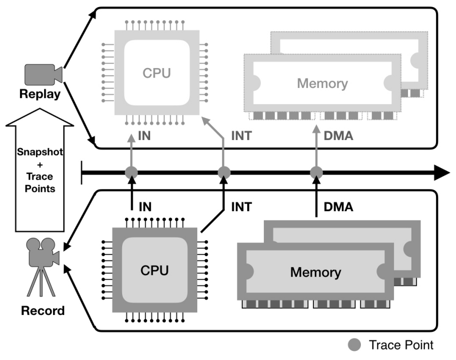


Figure 1: The overview of deterministic record and replay of PANDA. IN: input from port; INT: hardware interrupts; DMA: DMA events.

PANDA to record the execution first, and then replay the program with poisoned memory values of the kernel stack and heap.

In order to detect uninitialized variables by comparing differences of two replay instances, the record and replay should be deterministic. PANDA solves this challenges in the following way [18]. When starting recording, it first takes a snapshot of machine states, including registers and memory values. Then it records three kinds of non-deterministic events, including input events (through the `IN` instruction), hardware interrupts, and DMA events. When any event happens, it logs the *trace point* information, which consists of the program counter (PC value), the instruction count since record began, and the ECX counter value used in the x86 `loop` instruction. The information is sufficient to distinguish one trace point from another [19].

During the replay, PANDA first restores system states based on the saved snapshot. Then it executes the program and feeds non-deterministic events into the system, when the current trace point is identical to the logged one. By doing so, it ensures that non-deterministic events are generated in a same (virtual) timeline as in the recording process. Figure 1 shows the overview of this process. Note that, the multi-threading will not cause any issue, since the execution trace of the multi-threaded program will be deterministically replayed.

It is worth mentioning that, during the replay, there is no device emulation code executed, and the replay cannot *go live* to accept new inputs. The purpose of PANDA is to analyze the recorded trace. This design choice makes the implementation of record and replay really simple and clean.

Taint analysis Taint analysis [46, 57] is a data flow analysis technique with wide usage. The basic idea is that it marks certain types of data as the taint sources, assigns taint tags to them and then propagates the tags when the program executes. When the program reaches certain locations, namely taint sinks, rules could be enforced by checking the tags. Our system uses the symbolic taint analysis to identify the location where the uninitialized data was allocated.

```
// nt!IopXxxControlFile (Entry: nt!NtDeviceIoControlFile):
/* Code 01:
IO_STATUS_BLOCK  localIoStatus;//And other stack vars
*/
// a new kernel stack is allocated (and uninitialized)
01: 0x83c6f838 sub esp, eax
...
/* Code 02-03:
if (fastIoDispatch->FastIoDeviceControl(&localIoStatus){
   *IoStatusBlock = localIoStatus; //Last 4 bytes leaked
}*/
// move value from the kernel stack into ecx
02: 0x83e244fe mov ecx, dword ptr [ebp - 0x70]
// move from ecx to a user space memory (eax + 4)
03: 0x83e24501 mov dword ptr [eax + 4], ecx
```

Figure 2: The code snippet of a kernel information leak (CVE-2018-8408).

2.2 A Motivating Example

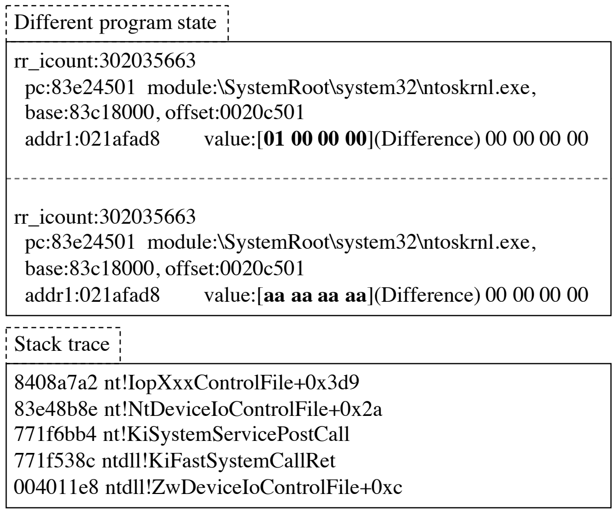
Before presenting the detailed system design and implementation, we first use a new kernel information leak vulnerability (CVE-2018-8408) detected by our system to show limitations of existing tools, which motivate our work. Figure 2 shows the code snippet of this vulnerability that leaks the kernel data from the stack.

A kernel information leak vulnerability This vulnerability exists in both Windows 7 and Windows 10 kernels. The system call `NtDeviceIoControlFile` can be used to set the UDP socket with a flag `FIONBIO` for the non-blocking I/O mode. This routine is invoked by the user-mode function `WSARecvFrom`. If this function is invoked without any incoming UDP data, a special status value `0xc00000a3` (`STATUS_DEVICE_NOT_READY`) will be assigned to the variable `IoStatusBlock.Status`. Four uninitialized bytes on the kernel stack will be leaked to a user-space variable named `IoStatusBlock.Information`, in the condition that the function `FltpFastIoDeviceControl` returns a nonzero value.

Why it cannot be detected by Bochspwn Bochspwn [28] is the state-of-the-art tool to detect kernel memory disclosure bugs. The core idea is using the *double-tainting* technique to trace the data flow of the uninitialized variable.

However, the tool cannot detect this vulnerability due to the limitation of its taint analysis capability. For instance, to reduce performance overhead, the tool only propagates the taint tag for the instructions of memory to memory operations. In other words, the tag will be lost when the tainted data flows into a register (which is indeed the case of this vulnerability.) Also, as stated by the author, its bug detection module is activated only if certain conditions are satisfied, i.e., when the `esi` is in kernel-mode and `edi` is in user-mode. However, in this case, the uninitialized kernel stack memory is first passed to the `ecx` register and then leaked to user space memory (Line 3 in Figure 2). This will cause the tool to generate a wrong taint state, and miss this vulnerability.

Why our system can detect this vulnerability Our system applies the differential replay technique to quickly spot the occurrence of the use of uninitialized data, without the need to dynamically



```
Different program state

rr_icount:302035663
  pc:83e24501  module:\SystemRoot\system32\ntoskrnl.exe,
  base:83c18000, offset:0020c501
  addr1:021afad8        value:[01 00 00 00](Difference) 00 00 00 00

rr_icount:302035663
  pc:83e24501  module:\SystemRoot\system32\ntoskrnl.exe,
  base:83c18000, offset:0020c501
  addr1:021afad8        value:[aa aa aa aa](Difference) 00 00 00 00

Stack trace

8408a7a2 nt!IopXxxControlFile+0x3d9
83e48b8e nt!NtDeviceIoControlFile+0x2a
771f6bb4 nt!KiSystemServicePostCall
771f538c ntdll!KiFastSystemCallRet
004011e8 ntdll!ZwDeviceIoControlFile+0xc
```

Figure 3: The observed different memory states of two replay instances, one is the normal replay instance, and the other one is the replay instance with the poisoned stack memory value `0xaa`.

trace data flow at runtime. In the following, we will describe how our tool detects this vulnerability with a high-level description.

First, we leverage the first key technique of our system to replay the recorded execution with poisoned kernel stack. During this process we find a difference of the program state as shown in Figure 3. Specifically the value at the memory location `0x021afad8` is different in two instances. That means an uninitialized data from the kernel stack is used. Moreover, we get the stack trace by using the function symbols and stack information that comes with `TimePlayer`. This stack trace helps the manual analysis of the vulnerability.

After that, we leverage the second key technique of our system to locate the source of the uninitialized variable, i.e., where this data was allocated. To this end, we perform a symbolic taint analysis starting from the last N (we used `500` in the experiment) kernel stack frames, and find the source of the leaked kernel data by applying the taint analysis on the execution trace, as shown in Figure 5 on page 7.

From Figure 5, we can see that the kernel frame is allocated at the instruction `0x83c6f838` (`sub esp, eax`). Our system sets the memory region (`[0x9e12fc44:0x9e12fcbf]`) as the taint source with a symbolic taint tag `<BV992 TAINT_s_0_992>` [2]. Then, with the program execution, the taint tag will be removed from the memory region if it is initialized (line 2), or propagated to other memory regions (line 3) and registers (line 4). At last, the use of the uninitialized variable is detected (line 5). In this example, the taint tag is first propagated to the `ecx` register (line 4), and then leaked to the user space memory (`[0x21afad8:0x21afadb]`) through the `ecx` register at the instruction address `0x83e24501` (`nt!IopXxxControlFile+418`) (line 5). Note that, the symbolic taint tag `<BV32 TAINT_s_0_992[256:287]>` denotes part of the kernel stack (from bit 256 to 287, 4 bytes in total) has been leaked.

Takeaway Instead of using the traditional dynamic taint tracking technique, `TimePlayer` leverages the differential replay to detect the use of the uninitialized data, which could be missed by other tools. Moreover, the symbolic taint analysis further helps us find the location where the uninitialized data was allocated, and the exact portion of the memory region that has been leaked with a bit-level granularity.

3 SYSTEM OVERVIEW

Detecting the use of uninitialized variables is not as easy as one may think. A variable could be allocated from multiple locations (stack and heap), and frequently used that may cross different privilege domains. If we leverage the dynamic taint analysis to track the variable in the whole system, including the kernel and user programs, it may introduce a high performance overhead, which makes the system impractical. Hence, it is a common practice to only partially track the instructions, which introduces false negatives [28].

Our system proposed a new technique called differential replay to quickly detect such an issue. Figure 4 shows the overall architecture of our system. Specifically, we use a full system emulator to record the execution of the operating system kernel and user programs (❶). Then we replay the execution, but with poisoned values for variables allocated from the stack and the heap (❷). If variables are properly initialized before being used, then two replay instances should have the same program states, since the initialization will overwrite the poisoned value. However, if the variables are used without being initialized, it will cause differences of program states. Thus our system can detect the use of uninitialized variables by detecting the differences during replay (❸), *without* performing the time-consuming *whole* system taint tracking.

After that, we further leverage the symbolic taint tracking to help us identify sources of uninitialized variables (❹). Our system performs a forward symbolic taint tracking along the program execution trace (❺). By doing so, we can identify the exact location where the variable was allocated (❻), and the uninitialized memory region that has been leaked.

In the following two sections, we will illustrate the two key techniques, i.e., differential replay and symbolic taint tracking, of our system.

4 KEY TECHNIQUE I: DIFFERENTIAL REPLAY

Differential replay involves several steps, i.e., recording the execution to save the program's state, poisoning memory, replaying programs and comparing the differences. In the following, we will illustrate these steps one by one.

4.1 Recording Program Execution

The main purpose of recording program execution is to save the state for replaying. Though there are many frameworks, e.g., PIN [36], Valgrind [45], that could be used, our system uses PANDA, a full-system emulation and analysis tool based upon Qemu [7], due to the following reasons.

First, PANDA is a non-intrusive framework that does not change the program's state, e.g., the memory layout, when recording the

[2]BV992 denotes the tag that is a bit vector with a length of 992 bits, each one denotes one tainted memory bit.

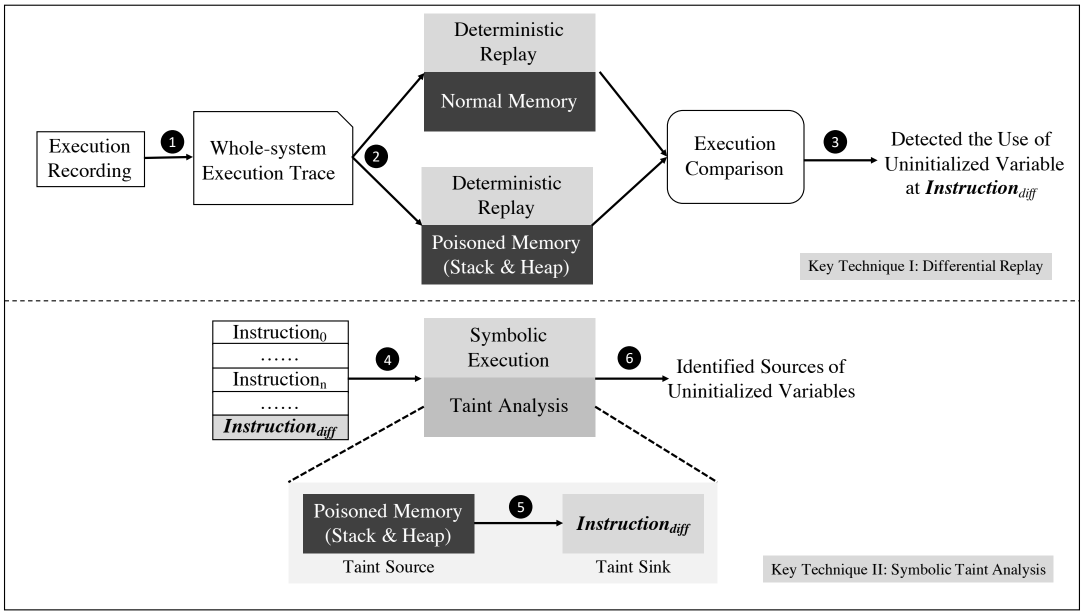

Execution Recording
1
Whole-system Execution Trace
2
Deterministic Replay
Normal Memory
Deterministic Replay
Poisoned Memory (Stack & Heap)
Execution Comparison
3
Detected the Use of Uninitialized Variable at Instruction_diff
Key Technique I: Differential Replay
Instruction_0
……
Instruction_n
……
Instruction_diff
4
Symbolic Execution
Taint Analysis
6
Identified Sources of Uninitialized Variables
Poisoned Memory (Stack & Heap)
5
Instruction_diff
Taint Source
Taint Sink
Key Technique II: Symbolic Taint Analysis

Figure 4: The overview of `TimePlayer`. It leverages the differential replay to detect the use of uninitialized variables, and the symbolic taint analysis to identify the sources of these variables.

execution. Other user-level tools like PIN and Valgrind will reside in the program's memory space, which could change the program's behaviors like memory allocation. This non-intrusive requirement to the target program (or the operating systems) is important to our system.

Second, our system needs to track the data flow inside the whole system, since variables could cross different privileged domains. Hence, we need a full system emulator that can run the OS kernel, system services and user-mode applications. This requirement excludes user-model instrumentation frameworks, e.g., PIN and Valgrind.

Third, PANDA is able to perform a system-wide *deterministic* replay. It ensures that the instructions, and other non-deterministic system events like interrupts, inputs can be replayed in a deterministic order. This capability enables us to reliably compare program states to find differences that are affected by the poisoned memory.

In our system, the recording functionality is built upon the PANDA system. During the implementation, we found and fixed several bugs in PANDA, such as missing some types of PCI DMA data, errors of handling `syscalls2` in the multi-threading scenario and etc.

4.2 Poisoning Memory

During the replay, we launch two instances. One instance does not change any memory, and another one is with poisoned memory of both the stack and the heap. We call the instance with vanilla memory as normal record-replay instance (`RRNormal` in short) in this paper. For another replay instance, memory regions are initialized with special values upon creation. Such an instance is called poisoned record-replay instance (`RRPoisoned` in short). To be specific, we change initial values of the memory allocated from the kernel stack and the heap. After that, if the program state is different between the `RRNormal` instance and the `RRPoisoned` instance, a use of an uninitialized variable variable is detected.

Poisoning timing To poison the memory allocated from the kernel stack and the heap, we need to find a way to monitor the creation of a new stack frame, and the allocation of new memory regions from the heap.

(1) *Stack frame creation:* When there is a subtraction operation on the stack pointer register (the `esp` register in x86 for instance), a new stack frame is created and its size could be obtained from the constant operator of the subtraction instruction. We can then poison the memory region inside the newly allocated stack frame accordingly.
(2) *Heap memory allocation:* When a function is invoked through a `call` instruction (or a similar one), we check the callee to determine whether a heap allocation occurs. For instance, if the callee is `ExAllocatePoolWithTag` in Windows, then a new space is being allocated from the heap, and the size of the memory can be retrieved from the parameters of these functions.

Poisoning policies Different scenarios have different requirements in terms of the granularity and poisoned values. To this end, `TimePlayer` supports different memory poisoning policies to fulfill these needs.

(1) *Granularity:* Our system supports different poisoning granularity, ranging from fine-grained byte level to coarse-grained word level. For instance, when using byte-level poisoning granularity, every poisoned byte will be different from one another. This is the default granularity of our system.
(2) *Poisoned value:* By poisoning the kernel memory and comparing the memory write operation to user space, our system could

identify kernel information leak. For instance, we can poison every byte allocated from the kernel stack with a special value `0xaa`.

`TimePlayer` keeps a record of all the poisoned memory into a list called `poisoning history`. This list will be used in the symbolic taint analysis in Section 5 to help locate the source of uninitialized variables.

4.3 Comparing Replay Instances

Our system compares the state of the `RRPoisoned` instance with the `RRNormal` instance. It simultaneously checks and compares the execution of some specific instructions, which are defined as `checking points` as follows.

Checking points Remember that `TimePlayer` is mainly designed to detect the use of uninitialized variables. Accordingly, we only perform the comparison at certain instructions, namely `checking points` in our system. They include memory read and write instructions. For instance, for the `mov rax, qword ptr [rsi]` instruction, our system compares both the address value in the register `rsi` and the memory value fetched from that address in the two replay instances. However, our system only considers a kernel information leak when *a memory write instruction is executed with the kernel privilege and the destination address is in user-space area, while at the same time there is a difference between two replay instances.*

Difference comparison We implement the difference comparison via the memory R/W callbacks in PANDA. Our system maintains a block of shared memory between two replay instances, which uses a data structure called Checking Points' Information Record (`CPIR`) to log detailed context information, e.g. accessed memory addresses and contents. All the `CPIR` entities are maintained using a linked list in the shared memory.

Specifically, the two replay instances interact in a producer-consumer manner. The `RRNormal` instance replays first. After a basic block is executed, a corresponding `CPIR` entity is pushed onto the top of `CPIR` list. Once the shared memory is full, the `RRPoisoned` instance is notified to start execution and pause the execution itself. The `RRPoisoned` instance fetches the `CPIR` from the shared memory and compares the accessed memory addresses and contents. It repeats this process until the shared memory is exhausted. Then it notifies the `RRNormal` instance to continue the execution and pauses itself.

Once a difference is found, our system logs the detailed context information. In this paper, we name the checking point with differences as `differential point`. The context information, along with the `poisoning history` will be used to identify the exact location where the uninitialized data was allocated (Section 5).

Continuation after the differential point After identifying one `differential point`, our system needs to continue the execution to find more. If an uninitialized variable does not affect the control flow of a program, it is straightforward to continue the replay. However, uninitialized variables may influence the control flow. For instance, the control flow of a program may depend on a comparing operation of the variable with a constant or another variable. Since the variable has been poisoned, it will lead to a change in the control flow compared to the original (recorded) one. This may confuse the replay functionality of the PANDA due to the misalignment of instruction count number. In this case, we need to find a solution to fix these side effects to let PANDA continue the replay process of the program.

Specifically, if we know which conditional branch instruction would use the uninitialized variable, we can dynamically feed the instruction at `differential point` with the data in `RRNormal` replay instance, instead of the data from the `RRPoisoned`. We can transparently do this since the execution of the `RRNormal` is before the `RRPoisoned`, and they are synchronized using the shared memory. Hence, we directly copy the data from the `RRNormal` instance and use it in the `RRPoisoned` instance.

Parallel replay The differential replay will incur high performance overhead, since we need to compare the program state at each `checking point`. To speed up this process, we introduce the concept of `parallel replay` in our system, based on the *scissors* plugin of PANDA. Our parallel replay works in the following way. It first performs a normal replay, but saves multiple snapshots (N1, N2 and etc.) and non-deterministic events (S1, S2 and etc.) with numbers of instruction count C1, C2 and etc. After that, we can replay the saved snapshots (along with the saved non-deterministic events) in parallel, with each one as a normal replay instance (Section 2 illustrates the background information of record and replay of PANDA.)

The parallel replay may lead to false negatives in theory. For instance, if an instruction in one piece of the snapshot uses a variable which was allocated from a preceding piece, it will be missed by our parallel replay, since the poisoning states in different pieces are separated. Although this problem could be mitigated with a trace slicing mechanism, which carefully chooses the points where variables would not be split, our system takes a more straightforward workaround. When splitting the whole snapshot into pieces, we expand the range of each piece so that it overlaps with adjacent ones. Though this mechanism cannot totally solve the problem, it reduces chances of occurrence. We will show the evaluation result of parallel replay in Section 6.3.

5 KEY TECHNIQUE II: SYMBOLIC TAINT ANALYSIS

When a `differential point` is detected, we need to further determine the source of the uninitialized variable, i.e., the location where the variable was allocated. The second key technique of our system, i.e., symbolic taint analysis, aims to fulfill this requirement.

5.1 Preparing Traces and Contexts for Taint Analysis

Our taint analysis applies to the trace of a program's execution. However, for performance concern, our system does not actively collect the context of the execution trace during the process of differential replay. Instead, when a differential point is detected, we look backward a number of instructions (this number is determined adaptively), and replay the execution from there to the differential point. We then collect the traces with detailed context information of each executed instruction accordingly.

We implemented a PANDA plugin to collect the context of execution traces. Specifically, for each instruction, our system logs the value of the program counter (`PC`), the stack pointer, and values of

	Operation	Taint Source	Taint Destination	Taint Expression	Instruction	Note
1	Taint Memory	/	[0x9e12fc44: 0x9e12fcbf]	<BV992 TAINT_S_0_992>	0x83c6f838 sub esp, eax	A new kernel stack frame is allocated. The taint expression is applied to the stack frame (992 bits)
......						
2	UnTaint Memory	/	[0x9e12fca4: 0x9e12fca4]	<BV8 1>	0x83e24116 mov byte ptr [ebp-0x2c], cl	The taint tag is removed from the memory range [0x9e12fca4: 0x9e12fca4] since a constant value (one byte) 0x1 is written into the memory.
3	Taint Memory	[0x9e12fca4: 0x9e12fca7]	[0x9e12fc28: 0x9e12fc2b]	<BV32 TAINT_S_0_992[672:695]..1#8>	0x83e24218 push dword ptr [ebp-0x2c]	The taint tag from [0x9e12fca4: 0x9e12fca7] is propagated to [0x9e12fc28: 0x9e12fc2b]. After that, the taint expression of the destination memory region comprises of three tainted bytes and one constant byte (valued 0x1).
......						
4	Taint Register	[0x9e12fc60: 0x9e12fc63]	ecx	<BV32 TAINT_S_0_992[256:287] >	0x83e244fe mov ecx, dword ptr [ebp-0x70]	The taint tag from [0x9e12fc60: 0x9e12fc63] is propagated to ecx register.
5	Taint Memory	ecx	[0x21afad8: 0x21afadb]	<BV32 TAINT_S_0_992[256:287] >	0x83e24501 mov dword ptr [eax+4], ecx	The taint tag is propagated from ecx to user-space memory [0x21afad8: 0x21afadb]. The source of the uninitialized variable is located (on kernel stack) and the exact position is also obtained (from bit 256 to bit 289).

Figure 5: One example of the symbolic taint analysis. The taint expression (the fifth column) in the table will be applied to the taint destination (the fourth column), i.e., the register or the memory region.

general purpose registers, the instruction sequence count, accessed memory addresses, etc. During this process, the semantics of each instruction is needed to guarantee that only the used operands (explicit and implicit) are inspected and logged. This is implemented with aids of Intel XED[23] and Capstone[51] for x86 and ARM respectively. Our system also supports the extension instruction sets, e.g., x87/MMX/SSE/AVX for x86.

However, how to set the code location where the trace starts, i.e. how far do we need to look backward, needs a further consideration. If the location is too far from the differential point, then it will take a long time to replay from that address and the size of the log file will be big. In contrast, if the location is too close, we could miss the location where the uninitialized variable was allocated.

In this paper, we propose a mechanism similar to the sliding window protocol. Specifically, we define the number of the stack frame as the window size, and set an initial value of the window size. Then we adaptively increase the size if the instructions inside the window do not cover the location where the variable is allocated. We continue this process until we successfully locate the allocation point, or when the window size reaches a threshold.

5.2 Symbolic Taint Analysis

After obtaining the trace and context information, we will perform the symbolic taint analysis. In our system, we define the memory regions that have been poisoned as taint sources (and assign corresponding symbolic expressions), while addresses and contents of the memory operands of the instruction at the `differential point` as taint sinks. After that, symbolic execution is carried out so that the taint expressions (tags) are propagated along the trace. Expressions are split or simplified if needed along the execution. In the following, we will use an example to elaborate this process.

Figure 5 illustrates the taint tag propagation process where an uninitialized variable gets leaked from the kernel stack into a user-space program (see the motivating example in Section 2.2). The main operation is adding or removing taint tags to or from memory regions and registers. Specifically, in the first line of the figure, a new kernel stack frame is allocated. Our system assigns a new taint tag to the whole stack frame memory region, with an expression `<BV992 TAINT_S_0_992>`. The symbol `TAINT_S_0_992` denotes this is a tainted stack with index 0 and the length of the tainted memory is 992 bits (124 bytes). Since we will have multiple stack frames during program execution, our system maintains a mapping table between the stack index to the concrete memory address. The second line of the operation is to remove the taint tag from the memory region `[0x9e12fca4: 0x9e12fca4]` (1-byte long) because the 1-byte constant value has been written into that memory. We continue this process until the fourth instruction in the figure that propagates the taint tag from memory into `ecx` register, and the fifth instruction that propagates the taint tag from `ecx` register to memory. It turns out that the destination memory address belongs to user space, which means a tainted memory value (an uninitialized variable) has been leaked to a user program. Moreover, with the taint expression, we can further locate the stack where the variable was allocated, and the length (32 bits) of the leaked data. In our system, the symbolic taint analysis is implemented by using SimuVEX [59] library with the symbolic execution engine on top of the VEX IR. Note that our system does not need to solve the expression.

5.3 Optimizations

Our system introduces two optimizations, i.e., selective execution and symbolic expression packing, to improve the performance of symbolic taint tracking.

```
;[0xb1daba2c:0xb1daba2f]-> EDX, <BV32 TAINT_544[96:127]>
win32k!lCvt+8  mov edx, [esp+0Ch]
win32k!lCvt+E  imul edx
win32k!lCvt+22 shrd eax, edx, 1Fh
win32k!lCvt+26 sar edx, 1Fh
win32k!lCvt+2C shrd eax, edx, cl
win32k!lCvt+2F adc eax, 0
……
win32k!GreGetCharABCWidthsW+186 mov ebx, eax
win32k!GreGetCharABCWidthsW+1AD sub edx, ebx
```

(a) The code snippet that propagates taint expression. Instructions in bold denote the ones that affect the register ECX.

```
<BV32 __add__(
0xffffffff * (
(
   0x40000000 * TAINT_544[120:120]#32 .. TAINT_544[120:127] ..
   TAINT_544[112:119] .. TAINT_544[104:111] .. TAINT_544[96:103]
)[32:63] >> 0x1f
)[0:2] .. (
   0x40000000 * TAINT_544[120:120]#32 .. TAINT_544 [120:127] ..
   TAINT_544[112:119] .. TAINT_544[104:111] .. TAINT_544[96:103]
)[34:62],
0x0#31 .. (
   if ((0x40000000 * ......])[33:33] == 0) then 0 else1), ......
)>
```

(b) The symbolic expression of the register ECX after win32k!GreGetCharABCWidthsW+1AD

Figure 6: The real example of a complicated symbolic expression that should be packed.

Selective execution Our system does not always use the symbolic execution engine to execute each instruction. If an instruction does not operate on the tainted memory or register, there is no necessary to symbolically execute it and we can safely skip it.

Specifically, we first parse the generated trace and translate each machine instruction into a VEX IR, if and only if the instruction is operating on the tainted memory or register. Otherwise, the instruction is unchanged. All the translated VEX IR and the native instructions are maintained inside a memory region, using a bit for each instruction (instruction mode bit) to denote the execution mode (symbolic or native). When we find a mode switch from the symbolic execution to the native execution, we can safely skip all the following native instructions and directly jump to the next symbolic instruction that operates the tainted value. Since the concrete context information of the instruction has been saved, we can restore the context and symbolically execute the following ones from there.

Symbolic expression packing With the growth of the complexity of symbolic expression, the time and space overhead to manipulate the new expression is also growing. Our system uses another optimization, i.e., expression packing, for better performance.

Specifically, we evaluate the complexity of each symbolic expression before propagating it. If the expression is too complex, we will split a new symbol as an alias to replace the old one. In our system, the complexity of an expression is measured by its length and depth. The length of the expression refers to the number of atomic symbols in the expression, while the depth means the number of steps to generate the expression. In practice, we set $1,024$ as the threshold for the length and 6 as the threshold for the depth. These threshold values are obtained through the experiments of multiple benchmark programs.

Table 1: The test cases used in the evaluation. We ran these programs in Windows with our system. User (remote) login means we record the process of (remote) user login (through the Microsoft remote desktop protocol.)

Name	Version	Name	Version
ReactOS Test Suite	0.4.9	Youku	7.6.8.12071
Firefox	64.0.2	Chrome 64 bits	70.0.3533.110
Chrome 32 bits	71.0.3571.98	User Login	-
IE	11.0.9600.19080IS	User Remote Login	-

Figure 6 shows a real example taken from the taint analysis in the latest version of Windows 7 kernel. In this example, we firstly set the kernel stack frame (`[0xb1dab99c:0xb1dab9df]`) as the taint source tagged `<BV544 TAINT_544>`. Then, after three copy instructions (`rep movs/push/mov`), 4 bytes from kernel stack are passed to `edx` register and remain uninitialized. After multiple arithmetic and bit-wise operations (Addition, Subtraction, Multiplication, Sign Extend, Bit Shift, etc.), the complexity of this symbolic expression will exceed the threshold. Our system packs the expression accordingly and generates a new simple expression used in the subsequent calculation. To preserve the taint information, our system records the relationship between the new expression and the original one.

6 EVALUATION

In the following, we present the evaluation result of our system. Our evaluation aims to answer the following questions.

Q1 - effectiveness: Can `TimePlayer` detect new vulnerabilities, and perform better than the state-of-the-art tool?
Q2 - efficiency: Can `TimePlayer` quickly detect the vulnerabilities?
Q3 - performance overhead: What is the performance overhead of the key techniques used by `TimePlayer` and whether proposed optimizations improve the performance of our system?

During the evaluation, all experiments were performed on a server with an Intel I7-7700K Quad-core 4.20 GHz processor and 32G bytes RAM, running the Ubuntu 14.04.1 system. `TimePlayer` is based on the full-system emulator, i.e., PANDA. Thus all operating systems evaluated are running as guest OSes, with 2G bytes RAM allocated to each one.

6.1 Effectiveness

We applied our system to multiple Windows 7 and 10 versions (both 32 and 64 bits) in a period of seven months (from July 2018 to January 2019). Specifically, we leveraged 8 test cases and ran them in windows 7 and 10 systems with the latest patches at that time, and used our system to record and replay the execution to detect the information leak from kernel space to user space. The test cases used in our evaluation are shown in Table 1. During the test, we poison the kernel memory data using the byte-level granularity.

New vulnerabilities: Table 2 shows the result of newly detected issues and vulnerabilities [3], and their detailed information. In total, our system identified 34 cases of kernel information leak in

[3]In this paper, we use issues to denote findings that have not been confirmed by Microsoft, and vulnerabilities to denote the ones that have been confirmed.

Table 2: The detected kernel information leaks in Windows operating systems. In total, our system detected 34 new issues, and 17 of them have been confirmed by Microsoft. We mask out part of the component name for the purpose of anonymity, since some of these issues have not been fixed.

	Component	System	stack/heap	Status	Discovery Date	Brief description
1	nt!Iop***File	win7/win10 32/64bit	stack	CVE-2018-8408	2018-07	0x04 bytes disclosure with IoStatusBlock
2	nt!Nt***Error	win7/win10 32/64bit	stack	CVE-2018-8477	2018-09	0x04 bytes disclosure with Response
3	nt!Nt***File	win7 32/64bit	stack	CVE-2018-8621	2018-08	0x04 bytes disclosure with IoStatusBlock
4	nt!Nt***Memory	win7 32/64bit	stack	CVE-2018-8622	2018-08	0x04 bytes disclosure with OldProtect
5	Ntfs!Nt***Journal	win7/win10 32/64bit	heap	CVE-2019-0569	2018-09	0x01-0x03 bytes disclosure with USN_RECORD structure
6	nt!Nt***Token	win7/win10 32bit	stack	CVE-2019-0621	2018-10	0x04 bytes disclosure with ReturnLength
7	nt!Nt***Memory#2	win7/win10 32/64bit	stack	CVE-2019-0767	2018-11	0x04 bytes disclosure with ReturnLength
8	nt!Nt***Port	win7 64bit	stack	CVE-2019-0775	2018-11	0x04 bytes disclosure with REMOTE_PORT_VIEW
9	nt!Alpc***Port	win7 64bit	stack	CVE-2019-0782	2018-11	0x04 bytes disclosure with REMOTE_PORT_VIEW
10	nt!Alpc***Attributes	win7/win10 64bit	stack	CVE-2019-0702	2018-11	0x10 bytes disclosure with ALPC message
11	nt!Alpc***Message	win10 64bit	stack	CVE-2019-0840	2018-12	0x04 bytes disclosure with ALPC message
12	nt!Nt***FileEx	win7/win10 64bit	stack	CVE-2019-0844	2018-12	0x04 bytes disclosure with IoStatusBlock
13	win32k!xxx***MsgEx	win7/win10 32bit	heap	CVE-2019-0628	2018-10	0x04 bytes disclosure with lpdwResult
14	nt!Nt***Control	win7 32bit	heap	CVE-2019-0661	2018-11	0x2140 bytes disclosure with OutputBuffer argument
15	nt!Nt***Information	win7 32bit	heap	CVE-2019-0663	2018-11	0x7f bytes disclosure with SystemInformation argument
16	win32kbase!RIM***Input	win10 64bit	stack	CVE-2019-0776	2019-01	0x04 bytes disclosure with input pointer
17	tcpip!Udp***Indication	win10 64bit	heap	CVE-2019-1039	2019-02	0x04 bytes disclosure with OutputBuffer argument
18	win32k!xxx***Terminal	win7/win10 32bit	stack	Discussion	2018-08	0x50 bytes disclosure
19	nt!Nt***Error#2	win7 32bit	stack	Discussion	2018-09	0x18 bytes disclosure
20	nt!Rtl***X86	win7 32bit	stack	Discussion	2018-09	0x04 bytes disclosure
21	nt!Ki***Apc	win7 32bit	stack	Discussion	2018-10	0x04 bytes disclosure
22	nt!Nt***Process	win7/win10 32bit	stack	Discussion	2018-10	0x04 bytes disclosure
23	mrxsmb!MRx***Transports	win7 32bit	stack	Discussion	2018-11	0x0a bytes disclosure
24	Ntfs!Ntfs***Extend	win10 64bit	stack	Discussion	2018-12	0x02 bytes disclosure
25	win32kbase!Check***Pointer	win10 64bit	stack	Discussion	2018-12	0x20 bytes disclosure
26	nt!Etwp***Space	win10 64bit	stack	Discussion	2019-01	0x04 bytes disclosure
27	nt!Pop***CleanUp	win7/win10 32bit	heap	Discussion	2018-10	0x1c bytes disclosure
28	nt!Pop***State	win7/win10 32bit	heap	Discussion	2018-11	0x04 bytes disclosure
29	nt!Exp***Info	win7 32bit	heap	Discussion	2018-11	0x01 byte disclosure
30	nt!Nt***Objects	win7 32bit	heap	Discussion	2018-11	0x02 bytes disclosure
31	nt!Etwp***Buffer	win7/win10 32bit	heap	Discussion	2018-11	0x04 bytes disclosure
32	nt!Iop***Request	win7 32bit	heap	Discussion	2018-11	0x01f0 bytes disclosure
33	nt!Etwp***Item	win7 32bit	heap	Discussion	2018-11	0x04 bytes disclosure
34	mpsdrv!Send***Notification	win7 32/64bit	heap	Discussion	2018-12	0x0f85 bytes disclosure

```
/* LocalResponse: uninitialized kernel stack
 * Response: user space memory
 */
Status = ExpRaiseHardError (ErrorStatus, NumberOfParameters,
                            UnicodeStringParameterMask, CapturedParameters,
                            ValidResponseOptions, &LocalResponse);

try {
    *Response = LocalResponse;  -> where the leak occurs
} except (EXCEPTION_EXECUTE_HANDLER) {
    NOTHING;
}
```

Listing 1: The code snippet of the function `NtRaiseHardError`

multiple Windows kernel components. We reported our findings to the Microsoft Security Response Center. Among them, 17 have been confirmed as vulnerabilities with assigned CVE numbers. For the rest ones, we are still in the process of communicating with Microsoft to evaluate potential security consequences, at the time of writing this paper (August 2019).

Case study: CVE-2018-8477 In the following, we will use one case study to demonstrate how our system detects the new vulnerability in the latest Windows kernel. This vulnerability exists in the `nt!NtRaiseHardError` module of the Windows 10 kernel. Part of the code snippet of the vulnerable module can be found in WRK(Windows Research Kernel) v1.2 project (Listing 1). This vulnerability is triggered by setting the value `ValidResponseOptions` to `OptionShutdownSystem`. In this case, the return value of the function `ExpRaiseHardError` will be `STATUS_PRIVILEGE_NOT_HELD`

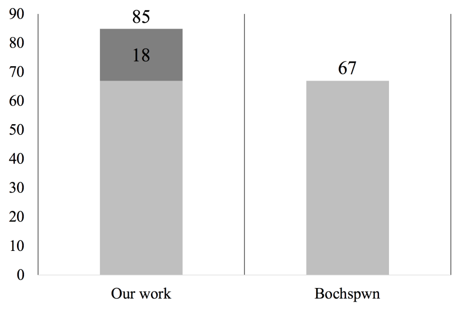


Figure 7: The comparison of our system with Bochspwn to detect known vulnerabilities. Our system can locate 85 vulnerabilities, while Bochspwn can detect 67 of them.

Table 3: The result to detect vulnerabilities in old versions of Windows kernels. These vulnerabilities (67 in total) can be detected by both our system and Bochspwn.

	Component	System	stack/heap	Status	Brief description
1	win32kfull!SfnINLPUAHDRAWMENUITEM	win7/win10 32/64bit	stack	Historical	CVE-2017-0167, 0x14 bytes disclosure (rep movsd)
2	win32k!xxxClientLpkDrawTextEx	win7 32/64bit	stack	Historical	CVE-2017-0245, 0x04 bytes disclosure (rep movsd)
3	nt!NtQueryInformationWorkerFactory	win7/win10 32/64bit	stack	Historical	CVE-2017-0300, 0x05 bytes disclosure (rep movsd)
4	win32k!xxxSendMenuSelect	win7/win10 32/64bit	stack	Historical	CVE-2017-11853, 0x0c bytes disclosure (rep movsd)
5	nt!NtGdiExtGetObjectW	win7/win10 32/64bit	stack	Historical	CVE-2017-8470, 0x04 bytes disclosure (memove)
6	win32k!NtGdiGetOutlineTextMetricsInternalW	win7/win10 32/64bit	stack	Historical	CVE-2017-8471, 0x04 bytes disclosure (movsd)
7	nt!NtGdiGetTextMetricsW	win7 32/64bit	stack	Historical	CVE-2017-8472, 0x07 bytes disclosure (rep movsd)
8	win32k!NtGdiGetRealizationInfo	win7/win10 32/64bit	stack	Historical	CVE-2017-8473, 0x08 bytes disclosure (rep movsd)
9	win32k!ClientPrinterThunk	win7/win10 32/64bit	stack	Historical	CVE-2017-8475, 0x14 bytes disclosure (rep movsd)
10	nt!NtQueryInformationProcess	win7/win10 32/64bit	stack	Historical	CVE-2017-8476, 0x04 bytes disclosure (rep movsd)
11	win32k!NtGdiMakeFontDir	win7/win10 32/64bit	stack	Historical	CVE-2017-8477, 0x68 bytes disclosure (rep movsd)
12	nt!NtQueryInformationJobObject	win7/win10 32/64bit	stack	Historical	CVE-2017-8478, 0x04 bytes disclosure (memove)
13	nt!NtQueryInformationJobObject#2	win7/win10 32/64bit	stack	Historical	CVE-2017-8479, 0x10 bytes disclosure (memove)
14	nt!NtQueryInformationTransaction	win7/win10 32/64bit	stack	Historical	CVE-2017-8480, 0x06 bytes disclosure (rep movsd)
15	nt!NtQueryInformationResourceManager	win7/win10 32/64bit	stack	Historical	CVE-2017-8481, 0x02 bytes disclosure (rep movsd)
16	nt!KiDispatchException	win7/win10 32/64bit	stack	Historical	CVE-2017-8482, 0x20 bytes disclosure (rep movsd)
17	nt!NtQueryInformationJobObject#3	win7/win10 32/64bit	stack	Historical	CVE-2017-8485, 0x04 bytes disclosure (rep movsd)
18	win32k!NtGdiGetPhysicalMonitorDescription	win7/win10 32/64bit	stack	Historical	CVE-2017-8681, 0x0100 bytes disclosure (rep movsd)
19	win32k!NtGdiGetFontResourceInfoInternalW	win7 32/64bit	stack	Historical	CVE-2017-8684, 0x50 bytes disclosure (memove)
20	win32k!NtGdiEngCreatePalette	win7 32/64bit	stack	Historical	CVE-2017-8685, 0x034c bytes disclosure (rep movsd)
21	win32k!NtGdiDoBanding	win7/win10 32/64bit	stack	Historical	CVE-2017-8687, 0x08 bytes disclosure (rep movsd)
22	nt!NtQueryInformationProcess#2	win10 32/64bit	stack	Historical	CVE-2018-0745, 0x04 bytes disclosure (rep movsd)
23	win32k!fnHkINLPMSLLHOOKSTRUCT	win7 32/64bit	stack	Historical	CVE-2018-0810, 0x04 bytes disclosure (rep movsd)
24	nt!RtlpCopyLegacyContextAmd64	win10 32/64bit	stack	Historical	CVE-2018-0832, 0x04 bytes disclosure (rep movsd)
25	nt!KiDispatchException#2	win7/win10 32/64bit	stack	Historical	CVE-2018-0897, 0x78 bytes disclosure (rep movsd)
26	nt!NtWaitForDebugEvent	win7/win10 32/64bit	stack	Historical	CVE-2018-0901, 0x04 bytes disclosure (rep movsd)
27	nt!NtQueryVirtualMemory	win10 32/64bit	stack	Historical	CVE-2018-0968, 0x04 bytes disclosure (memove)
28	nt!NtQueryAttributesFile	win7/win10 32/64bit	stack	Historical	CVE-2018-0969, 0x04 bytes disclosure (rep movsd)
29	nt!NtQuerySystemInformation	win7/win10 32/64bit	stack	Historical	CVE-2018-0971, 0x04 bytes disclosure (rep movsd)
30	nt!NtQueryVirtualMemory#2	win7/win10 32/64bit	stack	Historical	CVE-2018-0974, 0x08 bytes disclosure (rep movsd)
31	nt!NtQueryFullAttributesFile	win7/win10 32/64bit	stack	Historical	CVE-2018-0975, 0x38 bytes disclosure (rep movsd)
32	nt!NtQueryInformationToken	win7/win10 32/64bit	heap	Historical	CVE-2017-0258, 0x08 bytes disclosure (memove)
33	nt!NtTraceControl	win7/win10 32/64bit	heap	Historical	CVE-2017-0259, 0x3c bytes disclosure (memove)
34	nt!NtNotifyChangeDirectoryFile	win7/win10 32/64bit	heap	Historical	CVE-2017-0299, 0x02 bytes disclosure (rep movsd)
35	nt!NtQueryObject	win7/win10 32/64bit	heap	Historical	CVE-2017-11785, 0x37 bytes disclosure (rep movsd)
36	nt!NtQueryDirectoryFile	win7/win10 32/64bit	heap	Historical	CVE-2017-11831, 0x19 bytes disclosure (rep movsd)
37	nt!NtDeviceIoControlFile	win7 32/64bit	heap	Historical	CVE-2017-8469, 0x01e4 bytes disclosure (rep movsd)
38	win32k!NtGdiGetOutlineTextMetricsInternalW#2	win7/win10 32/64bit	heap	Historical	CVE-2017-8484, 0x05 bytes disclosure (rep movsd)
39	nt!NtDeviceIoControlFile#2	win7 32/64bit	heap	Historical	CVE-2017-8488, 0x1a bytes disclosure (rep movsd)
40	nt!WmipIoControl	win7/win10 32/64bit	heap	Historical	CVE-2017-8489, 0x48 bytes disclosure (rep movsd)
41	win32k!NtGdiEnumFonts	win7/win10 32/64bit	heap	Historical	CVE-2017-8490, 0x47 bytes disclosure (rep movsd)
42	nt!NtDeviceIoControlFile#3	win7/win10 32/64bit	heap	Historical	CVE-2017-8491, 0x08 bytes disclosure (rep movsd)
43	nt!IopXxxControlFile	win7/win10 32/64bit	heap	Historical	CVE-2017-8492, 0x04 bytes disclosure (memove)
44	NSI!NsiGetParameter	win7/win10 32/64bit	heap	Historical	CVE-2017-8564, 0x0d bytes disclosure (memove)
45	nt!NtGdiGetGlyphOutline	win7/win10 32/64bit	heap	Historical	CVE-2017-8680, arbitrary number of bytes (rep movsd)
46	nt!NtQuerySystemInformation#2	win10 32/64bit	heap	Historical	CVE-2018-0746, 0x0c bytes disclosure (memove)
47	nt!NtQueryVirtualMemory#3	win7/win10 32/64bit	heap	Historical	CVE-2018-0894, 0x04 bytes disclosure (rep movsd)
48	nt!NtQueryInformationThread	win7/win10 32/64bit	heap	Historical	CVE-2018-0895, 0x04 bytess disclosure (rep movsd)
49	nt!NtQuerySystemInformation#3	win7/win10 32/64bit	heap	Historical	CVE-2018-0973, 0x04 bytes disclosure (rep movsd)
50	win32k!NtUserGetSystemMenu	win7 32bit	stack	Historical	0x0c bytes disclosure (rep movsd)
51	win32k!NtUserCreateWindowEx	win7 32bit	stack	Historical	0x0c bytes disclosure (rep movsd)
52	win32k!SfnINLPCREATESTRUCT	win7 32bit	stack	Historical	0x0c bytes disclosure (rep movsd)
53	win32k!NtUserRealInternalGetMessage	win7 32bit	stack	Historical	0x10 bytes disclosure (rep movsd)
54	win32k!xxxClientLoadMenu	win7 32bit	stack	Historical	0x02 bytes disclosure (rep movsd)
55	win32k!xxxMNGetBitmapSize	win7 32bit	stack	Historical	0x14 bytes disclosure (rep movsd)
56	nt!NtCallbackReturn	win7 32bit	stack	Historical	0x14 bytes disclosure (rep movsd)
57	win32k!NtUserBeginPaint	win7 32bit	stack	Historical	0x20 bytes disclosure (rep movsd)
58	win32k!TraceGreReleaseSemaphore	win7 64bit	stack	Historical	0x24 bytes disclosure (memove)
59	win32k!xxxWindowEvent	win7 32bit	stack	Historical	0x04 bytes disclosure (rep movsd)
60	win32k!NtUserGetScrollBarInfo	win7 32bit	stack	Historical	0x04 bytes disclosure (rep movsd)
61	win32k!NtUserGetMenuBarInfo	win7 32bit	stack	Historical	0x06 bytes disclosure (rep movsd)
62	win32k!ClientLoadLibrary	win7 32bit	stack	Historical	0x06 bytes disclosure (rep movsd)
63	win32k!NtUserBeginPaint#2	win7 32bit	stack	Historical	0x08 bytes disclosure (rep movsd)
64	nt!NtTraceEvent	win7 32bit	heap	Historical	0x04 bytes disclosure (rep movsd)
65	csc!CscDclpInitializeFsctlBufferContext	win7 32bit	heap	Historical	0x05 bytes disclosure (rep movsd)
66	nt!PfSnBuildDumpFromTrace	win7 32bit	heap	Historical	0x0260 bytes disclosure (rep movsd)
67	nt!PfSnGetCompletedTrace	win7 32bit	heap	Historical	0x44 bytes disclosure (rep movsd)

Table 4: The result to detect vulnerabilities in old versions of Windows kernels. These vulnerabilities (18 in total) can be detected by our system, but missed by Bochspwn.

	Component	System	stack/heap	Status	Brief description
1	win32k!NtQueryCompositionSurfaceBinding	win7/win10 32/64bit	stack	Historical	CVE-2017-8678, 0x04 bytes disclosure (xmm0 to memory)
2	nt!NtQueryVolumeInformationFile	win7/win10 32/64bit	stack	Historical	CVE-2018-0970, 0x10 bytes disclosure (xmm0 to memory)
3	nt!NtQueryVolumeInformationFile#2	win7/win10 32/64bit	heap	Historical	CVE-2017-8462, 0x01 byte disclosure (eax to memory)
4	nt!NtSetIoCompletion/NtRemoveIoCompletion	win7/win10 32/64bit	heap	Historical	CVE-2017-8708, 0x04 bytes disclosure (rax to memory)
5	nt!NtQueryInformationTransactionManager	win7/win10 32/64bit	heap	Historical	CVE-2018-0972, 0x08 bytes disclosure (ecx to memory)
6	msrpc!MesEncodeIncrementalHandleCreate	win7/win10 32/64bit	heap	Historical	CVE-2018-8407, 0x10 bytes disclosure (xmm0 to memory)
7	nt!PspWow64GetContextThreadOnAmd64	win7 64bit	stack	Historical	0x02 bytes disclosure (r10 to memory)
8	win32k!NtUserToUnicodeEx	win7 32bit	stack	Historical	0x04 bytes disclosure (eax to memory)
9	win32k!NtUserCallNoParam	win7 32bit	stack	Historical	0x04 bytes disclosure (eax to memory)
10	nt!CommonDispatchException	win7 32bit	stack	Historical	0x04 bytes disclosure (ecx to memory)
11	win32k!SfnDWORD	win7 64bit	stack	Historical	0x04 bytes disclosure (r10 to memory)
12	nt!ExpReleaseResourceForThreadLite	win7 64bit	stack	Historical	0x04 bytes disclosure (r10 to memory)
13	win32k!DEVLOCKBLTOBJ::bMapTrgSurfaceView	win7 64bit	stack	Historical	0x04 bytes disclosure (r10 to memory)
14	nt!NtWriteFile	win10 64bit	stack	Historical	0x04 bytes disclosure (xmm0 to memory)
15	win32k!PopThreadGuardedObject	win7 64bit	stack	Historical	0x08 bytes disclosure (rax to memory)
16	win32k!NtUserPeekMessage	win7 64bit	stack	Historical	0x08 bytes disclosure (rax to memory)
17	nt!NtLockFile	win10 64bit	stack	Historical	0x08 bytes disclosure (xmm0 to memory)
18	nt!NtUnlockFile	win10 64bit	stack	Historical	0x08 bytes disclosure (xmm0 to memory)

(`c0000061`), and four bytes in the kernel stack (`LocalResponse`) will be leaked to the user space memory `Response`.

Our system locates this vulnerability when replaying the instances with the poisoned kernel stack. After executing `5.3 billion` instructions, a differential point is encountered at PC `0x819e6147` (`nt!NtRaiseHardError+0x175`). The instruction in the differential point is `mov dword ptr [ebx], eax`, where four bytes of `0xaa` (i.e. the poison value) in `eax` have been moved into a user space memory region pointed by `ebx`. Then we go back 500 stack frames ahead of the differential point, and start the forward symbolic taint analysis. When reaching the differential point, the taint symbolic expression of `eax` is `TAINT_FunctionID_896[224:255]`, denoting that the uninitialized value comes from `NtRaiseHardError` at stack offset `0x1C-0x1F`. The expression clearly reflects the relationship between the uninitialized kernel stack and information leaked to user space memory. This shows the capability of our system to detect the use of uninitialized variables. Moreover, it demonstrates that the symbolic expression can help an analyst to understand which part of the memory has been leaked.

Comparison with Bochspwn: Since Bochspwn is the state-of-art tool to detect kernel memory leak, we would like to compare our tool with it. To this end, we used the proof-of-concept (PoC) exploits publicly released by Bochspwn and ran them in old versions of Windows systems (Windows 7 Service Pack 1 6.1.7601.17514 32/64 bits, Windows 10 pro 1703 10.0.15063.674 32 bits and Windows 10 pro 1607 10.0.14393.0 64 bits). We use these old versions because they contain vulnerabilities that have been fixed in the latest version. In total, we collected 52 public exploits from Bochspwn and used them in our evaluation.

The result is encouraging. Our system detected 85 vulnerabilities, while Bochspwn can only detect 67 of them. The result is shown in Figure 7 (Table 3 and 4 show the details.) After carefully inspecting all the findings and cross-checking with the CVE information of Microsoft from 2016 and 2018, we found that 55 of them have CVE numbers. For the other 30, they do not have CVE numbers,

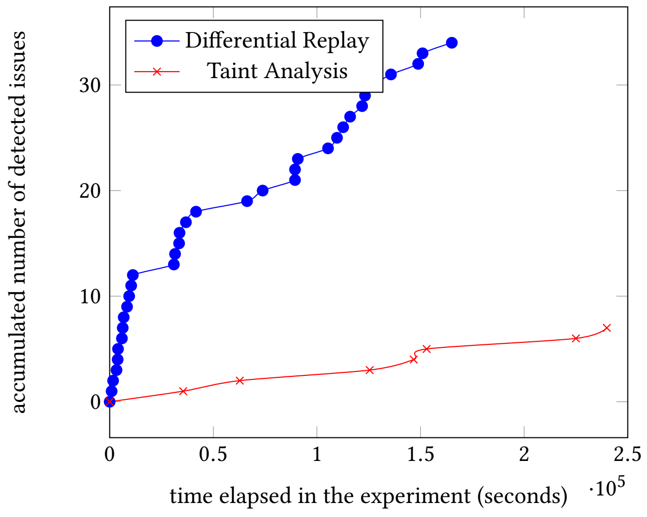


Figure 8: Time used to detect new vulnerabilities in Table 2 by differential replay and dynamic taint analysis. We find that differential replay can detect all of them in less than 170,000 second (47 hours), while dynamic taint analysis can only detect 7 of them in 240,000 seconds (66 hours).

and are not publicly known. However, our manual analysis confirmed they are indeed kernel information leaks. We believe they are vulnerabilities detected and patched internally by Microsoft. This demonstrated the effectiveness of our system.

Summary `TimePlayer` can detect new vulnerabilities that affected the latest versions of Windows systems. Using the same test cases, all the vulnerabilities reported by Bochspwn could also be detected by our system, and our system reported 18 more vulnerabilities.

Table 5: Overhead of differential replay. The second column shows the time consumed in vanilla replay, and the third column shows the time consumed in replay with memory poisoning and differences checking, the fourth and fifth columns is the number of issues found in corresponding test case.

Test Case	Vanilla Replay	Replay with Poisoning & Checking	# of Instructions	Issues: stack/heap
ReactOS_a	222.85s	5,063.25s	14,311,248,868	4/2
Chrome	837.32s	18,962.72s	61,142,430,613	5/3
cp	1507.04s	37,503.84s	113,650,380,756	3/2
Chrome64	1110.92s	24,885.625s	77,569,624,726	6/3

6.2 Efficiency

One advantage of our system is leveraging the differential replay to quickly detect the use of uninitialized variables, instead of using taint analysis to track the data flow. In the following, we will compare the efficiency of the differential replay with the taint analysis, from the perspective of tracking uninitialized variables.

To this end, we implement a system to track the data flow using the taint analysis (this one is called reference system). Specifically, we use PANDA to log the execution trace, and then perform the offline taint analysis on the trace to detect whether the variable in the kernel space has been leaked to user space. We feed same test cases (Table 1) to the reference system and `TimePlayer` one by one, and log the time when each vulnerability was detected.

The result is shown in Figure 8. The x axis means the time elapsed in the experiment and the y axis denotes the accumulated number of detected vulnerabilities. `TimePlayer` consumed $165,102$ seconds to detect all 34 issues, and analyzed $553,385,983,467$ instructions in total. The average speed is around $3,351,775$ instructions per second. However, for the system using taint analysis, it consumed more than $240,000$ seconds (66 hours) and only detected 7 of them within that time period. This is due to high performance overhead introduced by taint analysis on the whole system trace.

Note that, our system also leverages taint analysis to locate the sources of variables. However, we only need to apply taint analysis to a small part of the whole system trace, and this process usually finishes in less than one minute (Section 6.3).

Summary The key technique, i.e., differential replay, is more efficient to detect the use of uninitialized variables than the taint analysis.

6.3 Performance

In the following, we will show the evaluation result of the performance of two key techniques, i.e., differential replay and symbolic taint analysis, respectively.

Differential replay Table 5 shows the overhead of differential replay. `ReactOS_a` is one test case inside the ReactOS test suits used in our evaluation. We also used `Chrome` to visit multiple web pages, and `cp` operation to copy and paste $1,226$ files (total size is around 40M bytes) inside the guest Windows operating system. The result shows that, differential replay is around 22-24x slower, compared with the vanilla replay. Note that, though the absolute speed is slow, it's still more efficient than dynamic taint analysis to detect vulnerabilities (Section 6.2).

Table 6: Overhead of differential replay in parallel (four instances).

Test Case	Replay	Replay with Poisoning & Checking	# of Instructions
ReactOS_a	222.85s	5,063.25s	14,311,248,868
ReactOS_a_1	59.66s	1,312.15s	3,577,812,223
ReactOS_a_2	62.38s	1,318.15s	3,577,812,214
ReactOS_a_3	61.81s	1,354.33s	3,577,812,215
ReactOS_a_4	64.24s	1,313.71s	3,577,812,189
Chrome64	1,110.92s	24,885.63s	77,569,624,726
Chrome64_1	292.12s	7,542.66s	19,392,406,183
Chrome64_2	214.56s	5,824.78s	19,392,406,181
Chrome64_3	246.94s	6,608.77s	19,392,406,179
Chrome64_4	221.52s	6,131.71s	19,392,406,157

Table 7: Overhead of symbolic taint analysis. The second and third columns show the time used to generate the trace, and the trace size, respectively. The last column shows the time used to perform the taint analysis.

Test Case	Trace Time	Trace Size	# of Instructions	Taint Analysis Time
CVE-2018-8408	30s	7.2GB	91.22M	5.842s
CVE-2018-8477	16s	3.2GB	40.15M	1.145s
win32k!xxxInitTerminal	15s	3.4GB	42.87M	4.871s
CVE-2019-0569	16s	3.2GB	40.12M	1.925s
win32k!xxxInterSendMsgEx	10s	2.6GB	32.24M	37.645s

Our system leverages an optimization strategy to replay multiple instances in parallel (Section 4.3). In the following, we evaluate the effectiveness of this optimization. We use the same test case `ReactOS_a` and `Chrome` as in the previous experiment. Since the number of CPU cores on the machine is four, we split recorded traces into four instances, which are poisoned and replayed in parallel. The result is shown in Table 6. We can see that, the time consumed depends on the slowest instance, i.e., the instance that runs $1,354.33$ seconds. Compared with the reported $5,063.25$ seconds without parallel replay, this optimization speeds up nearly 4x.

Symbolic taint analysis As shown in Table 7, the symbolic taint analysis is very efficient. When a `differential point` is detected, our system looks back several stack frames and replay from there to generate detailed system trace for further taint analysis. The second, third and fourth columns show the time used to generate the trace, the trace size and the number of instructions inside the trace. After that, we apply the (offline) symbolic taint analysis on the trace. In particular, `TimePlayer` propagates the taint tags and generates the symbolic expression along the trace, until it reaches the `differential point`. The time is reported in the last column.

Note that, during evaluation, we set the initial value of the window size as 500, and the threshold to $5,000$ (Section 5.1). In most cases, our system can successfully locate the source with a window size less than $1,500$. Since we only need to consider instructions inside the window, the taint analysis typically will end in one minute.

Summary The differential replay is around 22-24x slower than the vanilla replay of PANDA. However, this process can be optimized using parallel replaying. The symbolic taint analysis typically finishes in one minute for real test cases.

7 DISCUSSION AND LIMITATION

False positives Our system compares differences of program states to determine the use of uninitialized variables. This operation is performed at the `checking point` (Section 4.3). In our prototype and evaluation, our system is leveraged to detect the kernel information leak. It only considers a kernel information leak when *a memory write instruction is executed with the kernel privilege and the destination address is in user-space area, while at the same time there is a difference between two replay instances.* This is a very conservative strategy, and will not introduce false positives. This has been confirmed by our findings that all the reported leakage can be manually confirmed (Table 2, Table 3 and Table 4).

False negatives Our system may have false negatives. That means some vulnerabilities could be missed. This is due to the nature of th dynamic system whose effectiveness depends on the code coverage during this process. In our evaluation, we leverage ReactOS test suits and popular programs (Table 1) to drive and record the system's execution. This leads to the discovery of 34 issues and vulnerabilities. However, we did not leverage any path exploration technique to actively trigger new paths. We believe the orthogonal efforts on the code coverage improvement, e.g., the fuzzing testing tools AFL [69] or KAFL [56] could be borrowed. Moreover, our system may miss the kernel memory leak due to DMA requests. That's because our system does not check memory operations issued from the DMA controller.

The value to poison memory needs to be selected carefully. That is because in some cases, the differences caused by the poisoned memory could be lost. For instance, if the uninitialized data (its value is 0x0) is used to perform the bitwise-AND operation with a constant value (0x1 for instance) and the poisoned word is 0xaa, it will not cause any difference between the vanilla replay instance and the poisoned replay instance, and the uninitialized variable will not be detected. To solve this problem, we can run the program multiple times with different data for poisoning to reduce the possibilities of false negatives.

Performance overhead and optimizations As shown in the evaluation, the differential replay introduces 22x-24x slowdown compared with the vanilla replay of PANDA. Note that this slowdown does not affect the effectiveness of our system to detect vulnerabilities, as demonstrated by the new vulnerabilities reported by our system. Also this overhead could be optimized by the parallel replay. Specifically, we split traces into multiple ones, and replay them in parallel according to the number of available CPU cores. Our experiment showed an obvious speedup (Table 6). However, the potential issue of the optimization is that it may cause false negatives. If the use of uninitialized variables is *across* parallel replay instances, then it will be missed by our system after applying this optimization.

8 RELATED WORK

Uninitialized variables detection Modern compilers such as GCC [37], Clang [32] and Visual Studio [39] usually provide the feature to detect uninitialized variables. However, most of them are limited to a single function and fail to deal with arrays, pointers and loops [53]. Some commercial products, such as CoBOT [2], Coverity [4] and Code Sonar [3], also show their abilities to detect the use of uninitialized variables. But they suffer from high false positives, especially when analyzing arrays. R. Jiresal et al. [27] try to reduce false positives by leveraging a summary based function analysis and control flow analysis on COBOL. However, whether its method is suitable for other languages like C or C++ is unknown. These tools are mainly for analyzing source code, while `TimePlayer` can analyze binaries without source code.

Some other tools [11, 58] employ dynamic analysis or hybrid analysis [24, 25, 62, 66] to detect such vulnerabilities. For instance, Memcheck [58] uses the Valgrind [45] binary translation system to detect the use of uninitialized memory based on the shadow memory. MemorySanitizer [62] relies on compile time instrumentation and bit-precise shadow memory. However, these systems are either for user-level applications, or require the compiler-aided instrumentation, which cannot be applied to privileged and closed-sourced Windows kernels.

Digtool[48] uses a specific byte pattern to fill memory regions during stack and heap/pool allocations, and searches for the pattern in the transferred data from kernel to user space. It can detect kernel information leaks, however, it cannot detect the case that the leaked data has been modified during the transfer. Compared to simple pattern matching used in Digtool, differential replay is immune to data change during the transfer (since it could still cause program state differences.) DieHard [9] performs differential syscall fuzzing to discover the use of uninitialized variables in system calls. The idea is close to our system. However, it only focuses on system calls, while `TimePlayer` aims to detect uninitialized variable vulnerabilities in the entire system.

Both UniSan [34] and SafeInit [40] intend to detect and fix uninitialized data leaks, using a compiler-based solution. Specifically, UniSan uses static data-flow analysis to check whether the uninitialized data can reach some predefined sinks, e.g., copy_to_usr and sock_sendmsg. If so, it fixes the vulnerable code with the help of the LLVM compiler. SafeInit adds an initialization pass to the LLVM compiler to initialize variables if they are not properly initialized. The main difference between our system and these two is that they require the source and leverage a compiler to perform the analysis, while our system works towards the binary code of Windows kernels directly. Due to this difference, our system faces different challenges, e.g., how to leverage differential replay to find uninitialized variables (without the availability of the source code).

A recent system kMVX [70] uses the concept of multi-variant execution (MVE) to detect the kernel information leaks in Linux. However, kMVX needs to extensively change the source code of target systems, thus it cannot be applied to Windows kernels. On the contrast, `TimePlayer` is a non-intrusive system that can work towards closed source systems, and its effectiveness has been demonstrated by detecting zero-day vulnerabilities in both Windows 7 and Windows 10 kernels.

Differential testing Our system leverages the differential replay to detect kernel information leaks. The idea of observing differences in program states is also used in the area of differential testing. For instance, differential testing was used to test the compiler of C language [38, 52, 65], SSL/TLS implementations [10, 13, 49, 60] and complex software systems [5, 12, 26, 29, 61]. These systems usually leverage the source code to do the test. On the contrast, our system does not rely on the source code to be effective.

Record and replay Record-replay technique aims at providing deterministic replay of programs in the presence of non-deterministic events, which can be applied to fields like debugging and security [19, 54, 55]. Usually non-deterministic events are the major challenges in record and replay systems. Some of them rely on customized hardware to handle non-deterministic events [22, 43, 44, 50, 64], while some others require a modified OS kernel [6, 8, 31]. SMP-ReVirt [20] is the first system that records and replays execution of the entire unmodified system within commodity multiprocessor hardware. It uses hardware page protection to detect the interactions between different CPU cores. RR [47] is a lightweight, practical user-space tool for record-replay. It runs only one thread at a time to avoid non-deterministic events caused by interaction between different cores. Our system uses PANDA [18] for a whole-system deterministic record and replay. By doing so, we are able to analyze Windows kernels with easy-to-use APIs to extend the functionalities of PANDA.

Dynamic taint analysis In the past decade, taint analysis has been extensively used in the field of computer security, such as data leakage tracking, vulnerability discovery, and etc. Some systems, e.g., TaintCheck [46], Taintgrind [30], TaintPipe[42], TaintTrace[14], are based on binary instrumentation to perform taint propagation. These tools are usually designed to track data flow in a single binary, not for the whole system data flow analysis.

Some other tools, such as TEMU [67], Panorama [68], TaintDroid [21], and OFFDTAN [63], use virtual machines to perform the whole system taint tracking. Since taint tagging and tracking consume huge amounts of resources, the efficiency of these tools become their major weakness.

FlowWalker[15] is an offline dynamic taint analysis tool, separating recording with analysis procedure. Such an architecture improves the efficiency of taint analysis. Based on FlowWalker, other techniques like in-memory fuzzing[16] and gray-box file formats analysis[17], also achieved good experiment results. FlowWalker builds taint analysis logic directly on the x86 assembly language, makes it hard to extend to other architectures (like x86-64). In addition, it is difficult for FlowWalker to handle taint elimination caused by bit shifts, logic operations, and arithmetic operations, while symbolic taint analysis in TimePlayer can handle these issues easily. StraightTaint[41] employs symbolic taint tagging and offline analysis, which is similar to the symbolic taint analysis of `TimePlayer`. However, StraightTaint leverages the user-level instrumentation tool PIN, which makes it unable to analyze operating system kernels.

9 CONCLUSION

In this paper, we aim to detect kernel information leaks due to the use of uninitialized variables. To this end, we propose two key techniques, i.e., differential replay and symbolic taint analysis to quickly find the use of uninitialized variables and locations where variables were allocated. We developed a prototype system called `TimePlayer`. The evaluation of applying our system on both Windows 7 and Windows 10 kernels demonstrated its effectiveness, with the discovery of 34 new issues (17 of which have been confirmed as vulnerabilities.)

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was partially supported by Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies, the National Natural Science Foundation of China under Grant 61872438 and the Fundamental Research Funds for the Central Universities. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

REFERENCES

[1] 2015. Revisiting an Info Leak. https://blog.rapid7.com/2015/08/14/revisiting-an-info-leak/.
[2] 2018. CoBOT Homepage. http://www.cobot.net.cn/
[3] 2018. Code Sonar Static Analysis Tool. http://www.grammatech.com/products/codesonar/overview.html
[4] 2018. Coverity Static Analysis Data Sheet. http://www.coverity.com/library/pdf/CoverityStaticAnalysis.pdf
[5] George Argyros, Ioannis Stais, Suman Jana, Angelos D Keromytis, and Aggelos Kiayias. 2016. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.*
[6] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2012. Efficient System-enforced Deterministic Parallelism. *Commun. ACM* (2012).
[7] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Conference on Usenix Annual Technical Conference.*
[8] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. 2010. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation.*
[9] Emery D Berger and Benjamin G Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation.*
[10] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy.*
[11] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization.*
[12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.*
[13] Yuting Chen and Zhendong Su. 2015. Guided Differential Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.*
[14] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. Tainttrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications.*
[15] Baojiang Cui, Fuwei Wang, Tao Guo, Guowei Dong, and Bing Zhao. 2013. FlowWalker: A Fast and Precise Off-Line Taint Analysis Framework. In *Proceedings of the 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies.*
[16] Baojiang Cui, Fuwei Wang, Yongle Hao, and Xiaofeng Chen. 2017. WhirlingFuzzwork: a Taint-analysis-based API in-memory Fuzzing Framework. In *Joural of Soft Computing.*
[17] Baojiang Cui, Fuwei Wang, Yongle Hao, and Lingyu Wang. 2016. A Taint Based Approach for Automatic Reverse Engineering of Gray-box File Formats. In *Joural of Soft Computing.*
[18] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop.*
[19] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay. *ACM SIGOPS Operating Systems Review* (2002).
[20] George W Dunlap, Dominic G Lucchetti, Michael A Fetterman, and Peter M Chen. 2008. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments.*
[21] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth.

2014. TaintDroid: an Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *ACM Transactions on Computer Systems (TOCS)*.
[22] Derek R Hower and Mark D Hill. 2008. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ACM SIGARCH computer architecture news*.
[23] Intel. 2018. Intel XED. https://intelxed.github.io
[24] François Irigoin, Pierre Jouvelot, and Rémi Triolet. 2014. Semantical Interprocedural Parallelization: An overview of the PIPS project. In *ACM International Conference on Supercomputing 25th Anniversary Volume*.
[25] Anushri Jana and Ravindra Naik. 2012. Precise Detection of Uninitialized Variables Using Dynamic Analysis-Extending to Aggregate and Vector Types. In *Proceedings of the 19th Working Conference on Reverse Engineering*.
[26] Suman Jana and Vitaly Shmatikov. 2012. Abusing File Processing in Malware Detectors for Fun and Profit. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*.
[27] Rahul Jiresal, Adnan Contractor, and Ravindra Naik. 2011. Precise Detection of Un-initialized Variables in Large, Real-life COBOL Programs in Presence of Unrealizable Paths. (2011).
[28] Mateusz Jurczyk. 2017. Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking. (2017).
[29] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*.
[30] Wei Ming Khoo. 2018. Taintgrind: a Valgrind Taint Analysis Tool.
[31] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *ACM SIGMETRICS performance evaluation review*.
[32] Chris Lattner. 2018. Clang: a C language Family Frontend for LLVM. http://clang.llvm.org/index.html
[33] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.
[34] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.
[35] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-before-initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*.
[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*.
[37] Manuel López-Ibáñez. 2007. Better Uninitialized Warnings. http://gcc.gnu.org/wiki/BetterUninitializedWarnings
[38] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* (1998).
[39] Microsoft. 2018. Visual Studio.
[40] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. Safeinit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium*.
[41] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. StraightTaint: Decoupled Offline Symbolic Taint Analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*.
[42] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proceedings of the 24th USENIX Security Symposium*.
[43] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. Delorean: Recording and Deterministically Replaying Shared-memory Multiprocessor Execution Efficiently. In *ACM SIGARCH Computer Architecture News*.
[44] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. 2006. Recording Shared Memory Dependencies using Strata. *ACM SIGARCH Computer Architecture News* (2006).
[45] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[46] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*.
[47] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*.
[48] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A virtualization-based Framework for Detecting Kernel Vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*.
[49] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient Domain-independent Differential Testing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*.
[50] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T King, et al. 2013. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. *ACM SIGARCH Computer Architecture News* (2013).
[51] Nguyen Anh Quynh. 2014. Capstone: The Ultimate Disassembler.
[52] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[53] Prof. John Regehr. 2011. Uninitialized Variables. http://blog.regehr.org/archives/519
[54] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: a Fully Integrated Practical Record/replay System. *ACM Transactions on Computer Systems (TOCS)* (1999).
[55] Yasushi Saito. 2005. Jockey: a User-space Library for Record-replay Debugging. In *Proceedings of the 6th international symposium on Automated analysis-driven debugging*.
[56] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*.
[57] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.
[58] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision.. In *Proceedings of the annual conference on USENIX Annual Technical Conference*.
[59] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*.
[60] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. 2017. HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*.
[61] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. 2011. A Security Policy Oracle: Detecting Security Holes Using Multiple API Implementations. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[62] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*.
[63] Xiajing Wang, Rui Ma, Bowen Dou, Zefeng Jian, and Hongzhou Chen. 2018. OFFDTAN: A New Approach of Offline Dynamic Taint Analysis for Binaries. In *Joural of Security and Communication Networks*.
[64] Min Xu, Rastislav Bodik, and Mark D Hill. 2003. A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay. In *ACM SIGARCH Computer Architecture News*.
[65] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[66] Ding Ye, Yulei Sui, and Jingling Xue. 2014. Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-flow Analysis. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*.
[67] Heng Yin and Dawn Song. 2010. Temu: Binary Code Analysis via Whole-system Layered Annotative Execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3* (2010).
[68] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*.
[69] Michal Zalewski. 2018. American Fuzzy Lop: a Security-oriented Fuzzer. http://lcamtuf.coredump.cx/afl/
[70] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.