# Catch you with cache: Out-of-VM introspection to trace malicious executions

Chao SU

Xuhua DING
*Singapore Management University*, xhding@smu.edu.sg

Qinghai ZENG

## Citation

SU, Chao; DING, Xuhua; and ZENG, Qinghai. Catch you with cache: Out-of-VM introspection to trace malicious executions. (2021). *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Taipei, Taiwan, 2021 June 21-24*. 326-337.

**Available at:** https://ink.library.smu.edu.sg/sis_research/6737

2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) | 978-1-6654-3572-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSN48987.2021.00045

# Catch You With Cache: Out-of-VM Introspection to Trace Malicious Executions

Chao Su
*State Key Lab for Novel Software Tech.*
*Nanjing University*
sc@smail.nju.edu.cn

Xuhua Ding
*School of Information Systems*
*Singapore Management University*
xhding@smu.edu.sg

Qingkai Zeng
*State Key Lab for Novel Software Tech.*
*Nanjing University*
zqk@nju.edu.cn

***Abstract*—Out-of-VM introspection is an imperative part of security analysis. The legacy methods either modify the system, introducing enormous overhead, or rely heavily on hardware features, which are neither available nor practical in most cloud environments. In this paper, we propose a novel analysis method, named as *Catcher*, that utilizes CPU cache to perform out-of-VM introspection. Catcher does not make any modifications to the target program and its running environment, nor demands special hardware support. Implemented upon Linux KVM, it natively introspects the target's virtual memory. More importantly, it uses the cache-based side channel to infer the target control flow. To deal with the inherent limitations of the side channel, we propose several heuristics to improve the accuracy and stability of Catcher. Our experiments against various malware armored with packing techniques show that Catcher can recover the control flow in real time with around 67% to 97% accuracy scores. Catcher incurs a negligible overhead to the system and can be launched at anytime to monitor an ongoing attack inside a virtual machine.**

***Index Terms*—Out-of-VM Introspection, Cache, Malware Analysis, Non-intrusiveness, Transparency**

## I. Introduction

Control flow tracing is one of the fundamental goals of dynamic malware analysis as it reveals malicious behaviors some of which are hardly available with static analysis. Typical ways to capture the control flow are either software instrumentation or the use of hardware facility. Code instrumentation places hooks to the target program and records control transfer destinations, especially system calls and function calls. Insertion of foreign instructions inevitably alters the target's virtual address space. Hardware facility for performance profiling and debugging purposes, e.g., Intel's Performance Monitor Unit (PMU), Processor Tracing, and ARM's TreadMill, has the capability of logging control transfers without modifying the target. Nonetheless, since this type of facility is often disabled in a production or end-user system, its activation strongly indicates the presence of malware analysis. To turn on the needed facility, the OS hosting the malware needs to explicitly set it up and handle events (if any) triggered by the facility in use. Moreover, certain facility also has noticeable performance slowdown. For instance, BTS is reported to introduce 6.1% overhead [34].

Malware nowadays infiltrates defense mechanisms by devising advanced evasion techniques. It might check debugger or other system artifacts to infer whether it is under analysis. Introspection for malware with anti-analysis abilities requires more transparency. Most analysis methods, working with a debugger or dynamic binary instrumentation, can be detected by the malware. We characterize existing methods as *environmental intrusive* approaches, because environmental changes (e.g., address space alternation, hardware setting changes, and performance slowdown) would not exist if the malware execution were not under analysis. Clearly, environmental intrusiveness is undesirable for malware analysis since it impairs transparency. Advanced malware may detect the presence of analysis, especially after it obtains the kernel privilege. Unlike x86 virtualization, those changes are strong evidence of analysis and malware will prohibit its malevolent behaviors.

In addition, the necessity of environmental variation hinders smooth and "silent" deployment of those analysis tools in scenarios wherein there is no prior setup or users cannot securely login to the concerned system. Although there are ways to impose the change on the system (e.g., by trapping it to the hypervisor [10]), they often contaminate the attack scene and may even cause malware execution to terminate.

Out-of-VM introspection [7], [9], [23], [31], [37], [39] examines a target virtual machine's virtual or physical memory and is especially known for its non-intrusiveness to the target as compared to in-VM introspection. However, the introspection scheme is limited to *read* the target memory and is *not* capable of capturing control flows in the VM. In this paper, we propose a novel out-of-VM introspection technique (termed as Catcher) to trace malicious execution without environmental variability.

## II. Synopsis

### A. System & Adversary Models

We consider a multicore platform with CPU and MMU virtualization extensions. It is managed by Linux KVM. To specify different systems in this paper, the virtual machine hosting the OS with target malware to analyze is defined as the target VM. We assume hardware security and cache latency cannot be manipulated by OS or malware. We also assume the host OS (i.e., the hypervisor) managing the target VM is trusted. The malware in study may contain anti-analysis techniques that check if it is under monitoring.

We focus on how to trace malware execution in a non-intrusive and transparent fashion. Applications of the tracing

978-1-6654-3572-7/21/$31.00 ©2021 IEEE
DOI 10.1109/DSN48987.2021.00045

data are not of our interest. Reverse engineering techniques are used as a building block in our scheme. Their accuracy is also out of our study scope.

### B. Idea Sketch

The design of Catcher is based on the fact that cache states are closely related to control flow. Different execution paths result in different cache states. Figure 1 depicts two possible execution traces of a process with dashed and solid arrows. Recently executed code blocks will show *"hit"* while non-executed code blocks will show *"miss"* when we test their cache states. The two cache states A and B are tightly bound with their execution paths. Hence, if the states of the cache can be obtained, the control flow of the program can probably be inferred.

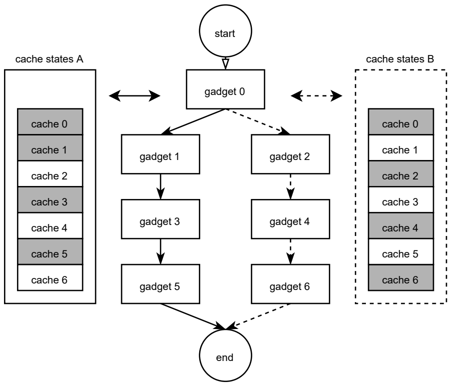

Fig. 1. The dashed arrows show one possible transfer while solid arrows show another. By probing all the cache state, two execution paths lead to two different cache states. Cache in gray means *"hit"* in cache probing while cache in white means *"miss"*

The basic idea of our method is to obtain cache states from time to time so that executions can be inferred. However, even if we keep observing cache states of all memory repeatedly, all we can get is a heat map of code execution. There is so much information to deal with that the analyzer could hardly stay synchronized with the target process.

In our core scheme, instead of scanning cache states of the whole memory, we only perform cache probing on branches of identified code blocks. For instance, in Figure 1, assuming that cache state of *gadget 0* is identified as *hit*, we need to identify *gadget 1* and *gadget 2*. If the cache probing shows a *hit* at *gadget 1* and *miss* at *gadget 2*, then *gadget 1* is recently executed while *gadget 2* is not. Algorithm 1 presents the pseudo code of the core scheme.

**Algorithm 1** Core scheme of Catcher

```
get a starting code block $g_0$
while $g_0 != end$ do
    $g_1, g_2 \leftarrow$ different branches from $g_0$
    $s_1 \leftarrow$ProbeCacheLineState($g_1$)
    $s_2 \leftarrow$ProbeCacheLineState($g_2$)
    if $s_1 == hit$ && $s_2 == miss$ then
        $g_0 \leftarrow g_1$
        add $g_0$ to execution path $P$
    else if $s_1 == miss$&& $s_2 == hit$ then
        $g_0 \leftarrow g_2$
        add $g_0$ to execution path
    else
        get another starting code block $g_0$
    end if
end while
return $P$
```

## III. Design of Catcher

Catcher is designed to analyze malware in a non-intrusive way. Other than pausing or relaunching the malware, Catcher dynamically fetches malware's codes, analyzes code distribution, and then introspects execution without disturbing the target process. By probing the cache status of basic blocks constantly, it detects if these basic blocks are executed. The states of cache (whether it hits or misses) are closely associated with control flow, and thus this provides a non-intrusive way to monitor target malware.

### A. Cache Access

Monitoring the target process from across VMs non-intrusively through CPU cache is hard. In most cases, the analyzer cannot obtain cache states directly across VMs. CPU cache-based side-channel attacks [2], [18], [32] utilize system features to access cache lines with an accordant or conflicting relationship. Various schemes in the literature [13]–[16] make use of page de-duplication and huge page to reveal the mapping of memory cache in victim processes. Other attacks [11], [12], [21], [38] require specific shared libraries. These leveraged features are not necessarily available in most environments, and thus they cannot be used to perform cross-VMs introspection. Besides, in CPU cache-based side-channel attacks, these methods tend to monitor cache line state of a small piece of memory, which is not practical in massive cache states management.

Nearly all processors nowadays use a "virtually indexed physically tagged" cache index policy in their last-level cache. The location of the cache line for the memory is decided by both its virtual address and physical address. Legacy out-of-VM introspection could hardly acquire the target process's cache state because of software-based MMU emulation. The isolation of memory space makes it difficult for analyzers to directly obtain the physical address, and thus cannot read its memory state. Methods utilized in side-channel attacks to leak memory mapping are limited by the scenarios.

If an analyzer needs to have the same virtual address and physical address as the target, it can probe a cache line by reading the virtual address directly. The time of the memory read indicates whether it's in cache or not. In other words, we expect the introspection analyzer to have the ability to read memory at native speed.

*Immersive Execution Environment* (ImEE) [39] proposes a prototype that introspects guest address space at native speed.

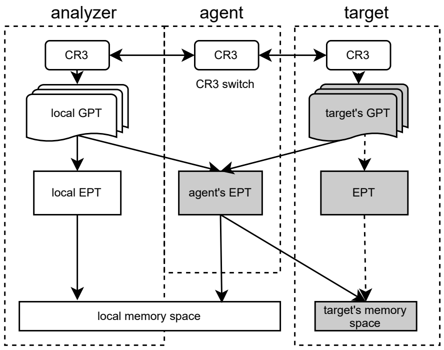

Fig. 2. By switching CR3, the agent can read from both the local and the target's memory space. It shares the same *VA* and *PA* with the target if the agent is loaded with its CR3

Typically an out-of-VM analysis program synchronizes the CR3 with the target VM so that its MMU can use target's VA-to-GPA mappings directly. Inspired by ImEE, Catcher provides a smart way for the analysis program to share the same virtual address and physical address. As is illustrated in Figure 2, Catcher launches an agent that switches its CR3 to the one from the target. Agent's mappings are ensured to be consistent with the target. It provides a method for the agent to share the same virtual address and physical address as the target, and thus it can read the target's cache state directly across VMs. In this period the analyzer do nothing but cache probing. After the agent switches back to its own CR3, the analyzer can still be executed in its memory address space under its local mapping.

### B. Heuristics

The fundamental base of Catcher is CPU cache, which is intensely limited by environmental factors. Although the hypothesis seems to be valid, the result of the analyzer is not practical enough. The false-positive rate is high due to the design of microarchitecture in CPUs. Much of the instability in the implementation stems from the situations listed below.

*1) **Synchronization Issue**:* Different from traditional dynamic analysis tools like binary instrumentation, Catcher does not let the target trigger anything to assist tracing. In the ideal case, the analyzer and target processes are executed at the same speed. The analyzer can probe on code blocks that have just been executed. However, the high speed of modern CPU clock speed makes it hard to keep both analyzer and target running at the same speed. The instruction pointer of the target might have been far ahead of the analyzer already. Long time intervals between cache line usage and probing might lead to an eviction. The time gap between the target and analyzer will be greater as time goes on if there is no synchronization.

To minimize the gap, Catcher sets a code interval in probing. The velocity contrast mostly results from switching memory space between the analyzer and the agents. The analyzer sends requests to inform its agents so that they probe specific memory regions. This takes extra time, making the execution lagging behind. So the first heuristics is to introduce a code gap into the analyzer. Instead of probing code blocks one by one, we leave out some of the blocks to reduce the impact of switching CR3. After the analyzer identifies one recently executed code block, it tries to probe on the cache lines that should be executed in a fixed time instead of the next code blocks. There might be a few code blocks between the one we choose and the identified block. By this means, we lower the rate of switching to reduce the impact of being out of synchronization.

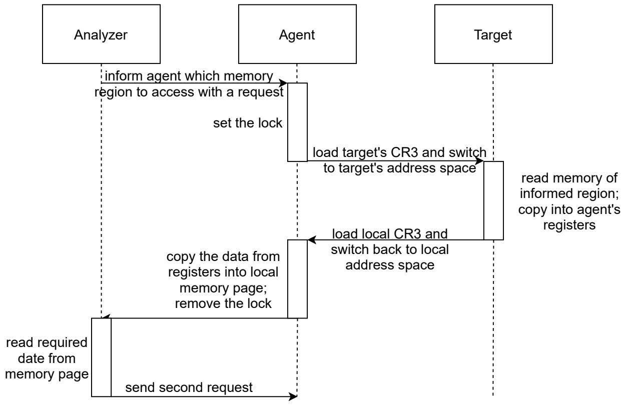
Analyzer
Agent
Target
inform agent which memory region to access with a request
set the lock
load target's CR3 and switch to target's address space
read memory of informed region; copy into agent's registers
load local CR3 and switch back to local address space
copy the data from registers into local memory page; remove the lock
read required date from memory page
send second request

Fig. 3. Communication between the analyzer and the agent over probes

Figure 3 illustrates the communication between analyzer, agent, and target. An agent will set the interface lock on after it receives a VMI request from the analyzer, then loads CR3 of target and access memory. The results are stored in registers to make sure they are available after CR3 is switched back. The analyzer needs to wait until the lock is off and then read the result from the agent. Agents are designed to access the target's memory at native speed. The communication includes two CR3 loads and memory access, so it might take some time, which is the key to define the execution time gap between two probes.

To evaluate the CPU cycles between two probes, we add rdtsc instructions in our analyzer and record CPU cycles every time the analyzer sends VMI request. It takes around 2500 CPU cycles between two probes so the interval is set as 2500 cycles. Intel 64 and IA-32 Architectures Optimization Reference Manual [4] documents the latency and throughput of different instructions. Under the guidance of [8], we estimate the execution time of every code block so that we pick a basic block around 2500 cycles away from the previous one. In other words, when the analyzer finishes its first probe, it prefers a basic block about 2500 CPU cycles away from the first probing. This heuristic makes the analyzer and target process run at approximately equal speed, mitigating the synchronization problem.

*2) **Branch Prediction and Speculative Execution Issue**:* A program may contain a branch that depends on a result from a prior slow instruction. Rather than wait for the result, CPU can predict the branch and speculatively execute instructions. To maintain the correctness of the process, CPU tracks the speculatively executed instruction and their results in a reorder buffer (ROB). If the prior result comes and the speculative execution is valid, CPUs can retire these speculatively executed instructions in ROB. However, if the guess is wrong, CPUs have to discard the incorrect instructions in ROB and run from the correct codes.

Although the instructions are discarded from ROB and the result of the code executed is roll-backed, the influence has already been made in CPU cache. Spectre and Meltdown vulnerabilities [19], [22] reveal the fact that even the correctness of the program is guaranteed, the cache state has already changed by speculative execution. Misprediction leads to a situation that both branches are loaded in cache. Catcher will be confused and cannot infer what the real execution trace looks like.

To mitigate this problem, we propose the heuristic that the analyzer prefers addresses near the end of basic blocks. Addresses far away from the branch instruction might be not polluted by speculative execution. Because of the limits on reorder buffer, speculative execution will not keep going perpetually. Once the reorder buffer is full of out-of-order instructions to be executed, CPUs stop speculative execution. If we choose an address far enough, the code at this address hasn't been speculatively executed, and thus it will no longer be affected by branch prediction and speculative execution.

Unfortunately, the size of ROB is not documented. To evaluate the capacity of ROB, we designed a method under the idea of [33]. We first set two memory reads to uncached memories. Between these two `mov` instructions, we add various filler instructions. If ROB is big enough for all these instructions, two memory reads can be executed in parallel. So if we measure the access time of these two memory reads, both latency is low. However, if we fill in so many filler instructions that the ROB is full, the second memory read needs to wait for the CPU to issue instructions in ROB. This will cause the latency of two memory reads to be different. So if we increase the number of filler instructions gradually until memory access latency at the second memory read changes, we can define the capacity of the ROB.

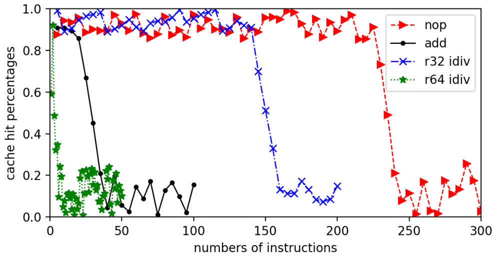

Fig. 4. Measure the size of ROB. The decline of cache hit percentage means number of filter instructions is over ROB's capacity.

We test an i5 3470 of Ivy bridge, and Figure 4 shows its ROB can hold approximately 225 `nop` instructions before the latency of the second memory read declines. We also find out the upper bound of instructions in ROB seems to vary among different instructions. More complicated instructions like `add` and `idiv` of `r32`, which take 10 and 88 $\mu$-ops, have lower upper bounds. In worst cases, ROB can only hold 3 instructions like `idiv` with operands `r64`.

The upper bound of the ROB suggests that before CPU realizes its misprediction and rolls back, speculative execution won't cover the code near the end of the basic blocks. Even if the size of the basic blocks is small enough to fit in ROB, it will not always have a bad influence on our analysis. As a matter of fact, branch predictors in modern CPUs are so accurate that in most circumstances they always choose the right branch. A 2-bit dynamic predictor was used in [24] where the misprediction rate is less than 10%. High accuracy of branch prediction mitigates the problem caused by speculative execution.

*3) **Shared Library Issue:*** Both Windows and Linux are introduced with shared library techniques to save disk and memory storage. *.dll* files in Windows and *.so* files in Linux are dynamically linked library files, which can be shared through several processes. Other than static linking, dynamic linking no longer copies codes of libraries into its executables. Instead, these libraries are linked at run time. They are loaded only once, and different programs are able to use the same copy in memory.

Cache states can be confusing because they can be affected by different processes through shared libraries. Both the target process and other irrelevant processes are executing the same library. In our heuristics, once the control flow turns into a shared library, we stop tracing. We are not interested in the functions of shared libraries. When the destination of a control flow transfer is a shared library, the analyzer will not trace it anymore. Instead, it handles it as if it loses its trace routine. Resetting strategies will be illustrated in the following sections.

To skip the shared libraries, the analyzer learns the mapping of libraries. By checking Linux's */proc* file system, we are able to know the addresses of different *.so* files. */proc/pid/maps* shows a detailed distribution of libraries used by the process. Tools in Windows, like *Process Explorer* and *ListDLLs*, are also available. *Process Explorer* shows information of DLL recently opened or loaded. *ListDLLs* reports the DLLs loaded into processes. Other tools like *Tasklist depends* and *dllshow* provide information on DLL usage as well.

*4) **Noise Issue:*** Noise is always an essential problem of works [30], [35] based on CPU cache. Cache is originally designed for CPU to access data from memory quickly, and it shouldn't be used as a side-channel. CPU cache states can be affected by the environment like CPU loads. To evaluate the impact of system noise caused by CPU loads, we run a test on cache. We start with a list of memory addresses mapping to different cache lines. By reading them one by one, we make sure each one of them should be in cache theoretically. Under different CPU loads with `stress -m`, we read them again and evaluate the access time with `rdtsc` instructions. If its access time is short, this memory should be in cache, meaning it has not been evicted yet. Otherwise, this cache line is already affected by system loads. Then we measure the influence on memories out of cache. Similarly, we use `CLFLUSH` instructions to make sure all these memory addresses are out of cache. Then we read them and evaluate the access time to find out how many addresses are now in CPU cache. We deploy our experiment on Ubuntu 12.04 with Linux kernel 3.2.79. We tested cache activities under quiet system, `stress -m 2` and `stress -m 4`. The percentages of addresses affected by system loads are presented in Table I. It draws a conclusion that system loads have a strong impact on cache states, introducing extra noise on cache probing. The system load has less effect on these memories out of cache.

TABLE I
PERCENTAGES OF MEMORY CACHE AFFECTED BY SYSTEM LOAD

| | silent | -m 2 | -m 4 |
|---|---|---|---|
| % of affected memory (cached): | 1.5 | 7.1 | 19.1 |
| % of affected memory (uncached): | 0.1 | 6.8 | 12.9 |

To minimize the adverse impacts of noise and make the analysis robust against high system load, we launch a vote in cache probing. When agents try to estimate if one basic block has been executed, they check it multiple times. Inside a basic block, we choose 3 different instructions whose address does not share the same cache line instead of one. if more than 2 probes out of 3 show a cache hit, agents are willing to take this basic block as executed. It is possible to run more cache probes inside a basic block. However, too frequent probes will slow down agents, deteriorating synchronization issue. Multiple probes can improve the accuracy of the analysis. Assuming the probability of one cache line influenced by noise is $p$, with a voting policy of best out of three probes, the probability can be reduced to $3p^2 - 2p^3$.

*5) **Lost Trace Handling:*** CPU cache is unstable so that it is common for the analyzers to lose trace. It may find *hits* in all branches, indicating all branches have been executed; or no *hit* at all, indicating no branch has been executed. In this case, we define a continuing strategy to follow when the analyzer has no idea about the target's execution. Choosing a reasonable continuing strategy contributes to high precision and code coverage.

The most basic yet most general strategy is picking an address to restart randomly. Once the analyzers are uncertain of the control flow, it scans all the addresses in the address pool. If a randomly-picked address is out of cache, the analyzer tries to re-pick another address after flushing the first address out of cache. Noticing probes can also include instructions into cache, it is necessary to always flush it after probing so that this won't affect the subsequent analysis. However, a random picking strategy is a game of chance. The more complicated the target is, the slimmer the chances of finding a valid address are. In extreme cases, the address pool can be so large that the analyzer can never find an address in cache since cache hit status is always changing. Spending too much time on choosing an address magnifies the time gap between the analyzer's and target's execution mentioned before.

Catcher implements a continuing strategy under the guidance of stack. When a function has been called, it pushes its parameters and return address into the stack. As stack is a last in first out data structure, there is a high likelihood that the function at the top of the stack will be executed in the following moment. The continuing strategy aims at return addresses recorded in stack, and keep probing them until one of the return addresses finally goes through a cache hit, indicating this function has returned. The analyzer can restart tracing at this recently returned address.

To trace the stack, Catcher maintains snapshots of the stack pages with a specific monitoring process. A process is designed to monitor the memory regions of the stack by its own agents. By comparing snapshots, it defines the top of the stack. If the memory area of the lowest address is modified, these changed parts are deemed to be function frames that were recently pushed into the stack.

This heuristic requires the analyzer to be aware of the address of the stack. However, Catcher agents are not supposed to access registers. Stack-related registers like `ESP` and `EBP` are not available to the agent, leaving the analyzers unknown of the address of the stack. We need to perform a stack revealing at the prologue phase. Stack is actually a memory area of different stack frames. Every frame is loaded with parameters and return addresses. By finding memory pages with several return addresses, we can find where the stack is. So when the analyzer dumps all the code and goes through reverse engineering, it marks all the addresses of `CALL` instructions. In this way, all the return addresses are known because they are the same as addresses of instructions next to the `CALL` instructions. Although modern compilers are likely to use `JMP` instructions to replace `CALL` instructions, the left `CALL`s are still enough to expose the stack. [6] shows in *coreutils* compiled by *gcc* at `-O2`, only about *10%* of the function calls are in the form of `JMP`s. Once the monitoring process knows where the stack is, it can keeps eye on it persistently. Whenever the analyzer lost trace of the target program, this monitoring process recommends an address that should be executed in no time.

Strategy based on the stack is not always efficacious. The heuristic described above is essentially a method that relies on an address of one instruction believed to be executed soon. This instruction will be executed as soon as the previous function returns. However if this function's life cycle is so long that it does not return in a short time, the analyzer then keeps probing cache states of the instructions and never advances. This is possible because in cases like *main* function or nested function calls, their stack frames stay in stack for a long time. If the monitored process delivers an address of this kind, the analyzer might stop tracing. To solve this problem, we take a hybrid strategy. When the analyzer is not able to trace the target with stack strategy for a long time (*100* probes by agents in our prototype), it will turn to a random strategy and pick an available address to continue tracing.

## IV. Implementation

### A. System Overview

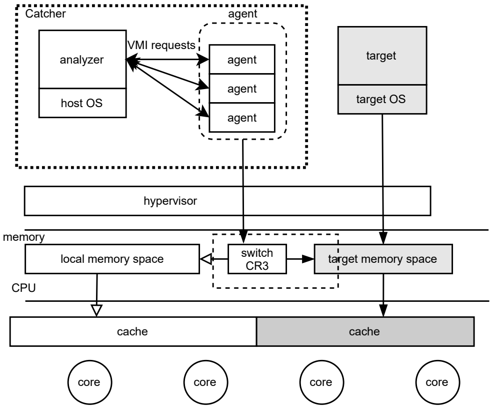

Fig. 5. High level architecture

Figure 5 shows the high-level architectural view of Catcher. The analyzer launches agents to send introspection requests. These agents will continuously give access to memory in the target process's memory space by switching CR3 to the one from the target. While the analyzer is working, the target malware is executed at the same time. The onsite tracing has almost no influence on the execution of the target process, and thus the target cannot sense the existence of the analyzer.

When agents switch their CR3 into the same one with the target process, they are capable of access the memory space of the target VM natively. The Last-level cache is shared across CPU packages, so agents can read memory and test CPU cache status by timing the access directly. With target and agents sharing the same *VA* to *PA*, if the accessing time of the memory is less than the threshold (in our case, 150 CPU cycles), it is more likely this memory has recently been read or executed. If it is out of the threshold, we assume this memory is not recently used. By probing different memory blocks' CPU cache states, we can reveal the target's execution dynamically.

We implemented our prototype on a desktop with an Intel Core i5-3470 3.2 GHz processor and 8GB DRAM. This CPU is equipped with 4 cores and 6MB shared level 3 cache. We run it on a Linux of kernel 3.2.79 with KVM. We created three different types of agents to access the target's memory space and support the analyzer. We can launch multiple analyzers. We designed two analyzers to cooperate with each other, which are consist of around 3100 SLOC.

### B. Agents

An agent is one small piece of code executed between the analyzer and the target. Like a trampoline, it attaches its window to either the analyzer or the target process by switching CR3. It contains a small memory frame to receive requests from an analyzer. When an agent is loaded with the CR3 from the analyzer, it shares the same memory address space with the analyzer, and thus the analyzer is capable of writing its requests to the memory frame. The agent then switches its CR3 to the one from the target. According to the requests agent receives from the analyzer, it will read memories of the specific virtual address and measure memory access time to define cache states. The result is also stored in the small memory frame so that after the agent switches back its CR3, the analyzers can get its responses to their requests.

We design three types of agents to perform memory revealing, cache probing, and data monitoring. Algorithm 2 illustrates part of the core logic in the code from a memory revealing agent. Firstly it keeps checking the lock. Once it is unlocked, the agent knows the analyzer has sent its requests. It resolves these requests and loads the address and the size of the memory region that the analyzer wants it to deal with next from the consensual frame. Then agent switches to the target's memory space by changing its CR3 and read the data. After copying it into the shared frame agent switches back CR3 and free the lock. Now our analyzer is able to read the result of their requests from the shared frame.

**Algorithm 2** The sketch of pseudo code of a memory revealing agent

```
while TRUE do
    repeat
        check the lock;
    until $lock == false$
    /* in local memory space*/
    read size $n$ and address $addr$ from the shared frame;
    load $n$ and $addr$ in agent's registers;

    /* switch to target's memory space*/
    load target's CR3;
    read requested data from target's memory space ;
    load it into registers;

    /* switch to local memory space*/
    load local CR3;
    write data from the register into local memory space;
    $lock \leftarrow true$;
end while
```

Cache probing agent is similar to the memory revealing cache except cache probing agent will utilize rdtsc instruction to measure the time of reading the target memory. The result it returns is whether the virtual address the analyzer requires is in cache or not. Data monitoring agent only returns when the data in the required memory area changes. The revealing agent is used to dump the executable memory of the target. It is vital in the prologue stage because all the analysis starts from here. The probing agent is the core part of the analysis. We use it to infer execution status and conclude the control flow of the target. Monitoring agent, however, is mostly used in stack-based heuristic. We suggest a monitoring agent keep an eye on the stack area of the target so that we can continue from a function return even when Catcher loses tracing.

### C. Analyzers

Analyzers are processes that schedule agents to detect execution states of the target process. In Catcher we launch two types of analyzers at once. They perform different functions and help researchers analyze targets' execution. Typically we have a main trace analyzer and stack support analyzer.

*1) **Main Tracing Analyzer**:* The main tracing analyzer is the principal part that takes most of the analysis work in Catcher. It is the only part that keeps working all the way through the whole analysis procedure. The main trace analyzer needs input from the analyzer, indicating the `CR3` of the target process. Then it will initiate a memory revealing agent to read the code distribution of the target process. After reverse engineer the memory area with the support of IDA Pro, the main tracing analyzer concludes a tree of virtual addresses, whose nodes refer to basic blocks of the target process and children refer to the next possible nodes to be executed. Details are introduced in the next section.

The main tracing analyzer also takes responsibility for dynamic tracing. That is, along with the execution of the target process, this analyzer keeps inspecting the running states of the target process. All this work is carried out silently. The analyzer monitors in onsite mode, which means it has almost no influence on the target, nor can it be detected by the target. The main tracing analyzer launches cache probing agents. It sends requests containing the addresses to probe on to the agent and waits for its response. The agent checks the cache states of these addresses. Usually the agent can find one of the cache states is shown as a *hit*, then it returns to the analyzer. The main tracing analyzer chooses addresses in the child nodes of the hit nodes and sends another request asking its cache probing agent to probe on them.

*2) **Stack Support Analyzer**:* Sometimes the cache probing agent may find none of the addresses is in cache, or more than one address is found in cache. In this case, the main tracing analyzer asks for stack support analyzer's help. Typically stack support analyzer synchronizes it with a recently called function so that the main tracing analyzer can go back to work again.

Stack support analyzer aims at monitoring the change in the area of the target's stack. Since Catcher is not able to check values in registers (`RSP` for example), we can not locate the malware's user stack directly. However, stack is basically a segment of data related to function calls. Every stack frame always contains a return address and we can therefore locate its user stack by finding memories containing a list of return addresses. When scanning the whole memory space, we find the address of `CALL` instructions, then the addresses of their next instruction are possible return addresses. Next we locate a memory space involving these return addresses, namely where the stack is.

Stack support analyzer has two parts. One is keeping an eye on stack changes. It launches an agent assigned to back up the stack. Once the stack changes, the analyzer realizes a new function has been called and a new stack frame has been pushed into the stack. As a matter of fact, this agent is insensitive to function returns because the popping act does not write anything in stack. The change in `RSP` is invisible to agent and the memory doesn't change. So when a function pops, agent can hardly notice it. However, we can still monitor the stack. The surveillance is based on the general knowledge that the function at the top of the stack will be returned in a short time. Eventually, the stack support analyzer picks these functions and recommends them to the main tracing analyzer when it loses trace of the target.

The stack support analyzer may fail if the process keeps calling functions and never returns (nested function calls). In this situation, the agent will keep backing up the stack. The cache state in this return address will never be *hit*. This problem can be settled because even though the stack keeps growing, the number of functions is limited. The code at the return address will eventually be executed, leaving the tracing work back to normal. Besides, the main tracing analyzer changes its continuing strategy when the stack support analyzer is not able to support tracing. It turns to a random continuing strategy.

The other part of the stack support analyzer is synchronizing with the main tracing analyzer. For most of the time, the stack support analyzer works individually. It focuses on the change of the user stack. When the main tracing analyzer asks for support, it recommends the most possible return address to it. The synchronization is by inter-process communication. The stack support analyzer shares a small chunk of memory with the main tracing analyzer. Through this shared memory, the tracing analyzer shows its request for suggestion on continuing address, and then the stack support analyzer responds with its answer.

### D. Analysis Workflow

To analyze a target process from scratch, we follow a workflow demonstrated in Figure 6. Catcher works in three phases: prologue phase, trace phase, and epilogue phase.

*1) **Prologue Phase**:* In the prologue phase, the first thing is to locate the target. `CR3` is the bridge between the virtual address and physical address. Catcher locates its target process through `CR3`. We can get target process's `CR3` by its *pid*. By searching `task_struck`, Catcher locates the target's process descriptor. From its `mm_struct`, we get `mm->pgd`. Then `virt_to_phys()` function uncovers the actual value of `CR3` of the target process.

Then the analyzer launches a memory revealing agent to read the code distribution. It fetches the code segment from the target's memory space. Agents are designed to be able to read memory blocks of different sizes. Although reading a small number of bytes at a time reduces the complexity of the agent code, it adds switching cost because the analyzer has to send requests more frequently. We choose 512 as the number of bytes for an agent to read at a time. It balances the performance and complexity of the agent.

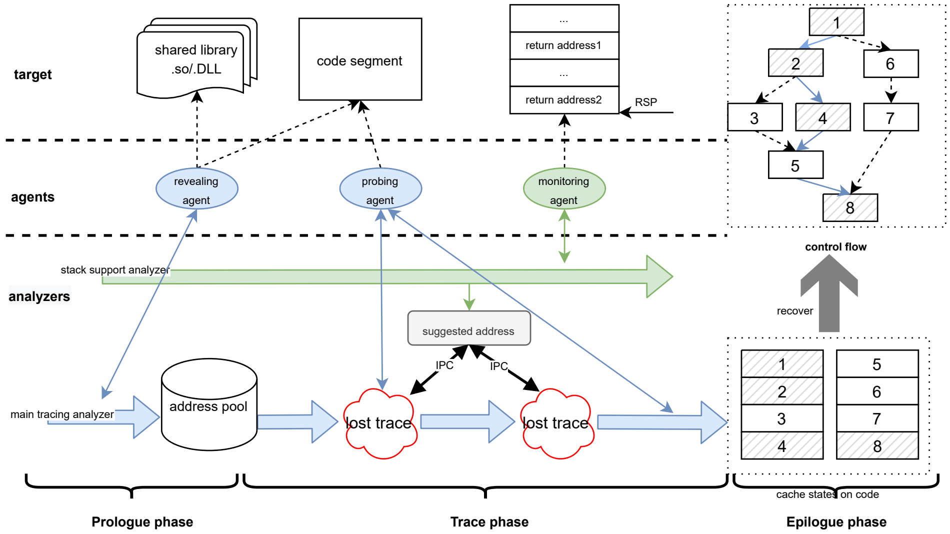

Fig. 6. Workflow of analysis in three phases. The main tracing analyzer asks for a suggested address when it gets lost.

After dumping out the code segment, the analyzer disassembles it preliminarily. Work about reverse engineering in Catcher is out of scope so it will not be discussed in detail. IDA Pro helps to reverse engineering and draw the connecting of different code blocks. This method dynamically gets the target's code from memory instead of ELF or EXE file.

In order to organize the probing addresses of all basic blocks, we evaluate the approximate execution time of every block. By looking up instruction latencies [8], the analyzer adds up all the instruction latencies in every basic block. It marks the next basic blocks to probe on by fulfilling the time gap. That is, when one basic block is designed to be probed, the analyzer tries to advance its possible control flow until the step is over the threshold (2500 cycles). Although the analyzer may miss some of the basic blocks, the lost code blocks can be inferred later in the trace phase manually if both their previous block and next block are identified in cache. We choose the virtual address near the end of the basic block as the probing address of the code block.

*2) **Trace phase**:* In this phase, one analyzer monitors cache states of the address pool marked in the prologue phase continuously while another one keeps eye on the stack. In no case will analyzers interrupt the execution of the target process. With the target process being executed, the analyzer starts probing cache states from an executed virtual address. This address can be manually selected, as long as the code is in cache. Then the analyzer will try to check which child virtual address of this block is also in cache, finding out how the actual execution trace is. By this means, the analyzer reveals the executed code blocks one by one.

It is common that sometimes analyzer may find a situation that all the child virtual address available is not in cache. The reason has been explained in the previous section. In this case, the stack monitoring analyzer, which is tracking function calls, suggests a possible virtual address to restart from. These two analyzers exchange information through inter-process communication.

*3) **Epilogue phase**:* The outputs from the previous phase are a list of virtual addresses that Cather captures. To turn these virtual addresses into a control flow transfer trace, we need to combine this address list with the basic block relationship from the prologue phase. We highlight the blocks whose address contains any of the virtual addresses from the list. Then we extend it by adding basic blocks that can be inferred. However, not all pruned basic blocks are able to be recovered. We will mark these blocks as uncertain blocks, meaning one of them should have been executed, but we are not sure which one it is. In CFG generation, code blocks are turned into nodes and all the transfers are turned into edges to connect nodes.

## V. Evaluation

In this section, we evaluate Catcher's usability, effectiveness, and performance overhead. We perform our experiments on a DELL OPTIPLEX 7010 desktop custom with an Intel Core i5-3470 3.2GHz processor and 8GB DRAM. This CPU is equipped with 6MB of L3 cache. ASLR is turned off for the sake of convenience. ASLR has no impact on the tracing since the code is dynamically fetched. Turning it off only simplifies the process of evaluation.

### A. Usability

Although Catcher is designed on Linux KVM, the target OS can be either Windows or Linux. To get `CR3` value in Windows, we check a kernel structure called the *EPROCESS*. Inside *EPROCESS* there is a member called *DirectoryTableBase*. We set the `DirectoryTableBase[0]` as the value of `CR3` of the target process.

Malware like keyloggers and rootkits tries to hide. Analyzers and AV scanners are not able to find the process because it hides its process directory in */proc* of the file system. It also manipulates kernel queue data structures by deleting itself from `task_struct`. Although pointers (`*prev_task` and `*next_task`) are removed, the process is still in memory. Catcher can catch it with a given `CR3`. We test Catcher on 4 keyloggers that can work secretively.

Some malicious programs detect whether they are under analysis based on the system artifacts [1]. Time-based detection utilizes API functions [26] (e.g., `QueryPerformanceCounter()` and `GetTickCount()`) or `rdtsc` instruction to get the time experienced by certain events, in order to infer whether there is a debugger. Some malware can even use external resources to assist timing (*NTP* for example). Once noticing the time of some specific events surpasses the threshold, they take a different path of execution and act benignly. Catcher is born with a great advantage in this situation because it does not intervene in the execution of the target process. Running in onsite mode, it has such an extremely subtle impact on the time of the events that malware cannot speculate whether there is a debugger. We test Catcher against three malware known to use timing-based detection to expose the presence of a debugger: HIV [17], Ratos [25] and Mydoom [20].

Malicious programs often use packers to deform their own code, thereby changing the shape and feature of their executable binaries. A different appearance helps to evade the detection of signature-based anti-virus scanners. Unpacking has no adverse impact on Catcher because the agent shares the same memory mapping with the target. Different from the legacy methods, EPT update in the target is also visible to the analyzer in Catcher, so the analysis is performed on the unpacked memory. We test four programs packed by packers: UPX, Upack, Armadillo, and Themida.

### *B. Effectiveness*

To demonstrate how effective this analysis method base on CPU caches, we measure its accuracy performance. The notations in use are listed in Table II below.

TABLE II
NOTATIONS AND DEFINITIONS

| | |
|---|---|
| $P$ | target process in memory to analyze |
| $e$ | execution of $P$ |
| $O(e)$ | the oracle we constructed |
| $C(e)$ | the execution trace caught by Catcher |
| $TP$ | true positive number, i.e., the number of code blocks $b \in O(e)$ such that $b \in C(e)$. |
| $FN$ | false negative number, i.e., the number of code blocks $b \in O(e)$ such that $b \notin C(e)$. This means it is actually executed while is not captured by Catcher. |
| $FP$ | false positive number, i.e., the number of code blocks $b \in C(e)$ such that $b \notin O(e)$. This means Catcher falsely captured a code block that has not been executed. |

We also define three notations for precision: $p$, recall $r$ and $f$-score as follows:

$$p = TP/(TP+FP) \quad (1)$$

$$r = TP/(TP+FN) \quad (2)$$

$$f\text{-score} = 2p \cdot r/(p+r) \quad (3)$$

*1) Ground Truth:* To evaluate the accuracy of Catcher, we need to construct an oracle of the actual execution path, that is, the ground truth. We need to record the addresses of all the code blocks executed. We use dynamic binary instrumentation to record the execution path. An Intel Pin tool is designed to monitor the execution at instruction level. It records the source address and destination address of all the control flow transfer instructions.

Using Intel Pin to instrument a program can be both intrusive and reboot-needed, which is against our assumption. The instruction level dynamic binary instrumentation also introduces massive overhead if it suspends the process for every instruction.

There are two hardware features widely available on commodity processors that can be used to help us obtain the ground truth. Last Branch Recording (LBR) is a mechanism that tracks branches. Previous work [3], [28], [29] shows that using LBR to record source and destination of filtered instructions incurs almost zero-overhead. LBR can record up to 16 from-to records in Model-Specific Registers (MSRs), according to our CPU model. That is, we can find 16 pairs of source and destination addresses of branch instructions at one time. Nevertheless, when it comes up with a 17th branch instruction, one of the 16 records will be wiped out. We log them before the 17th branch instruction is executed. [36] proposes an idea that utilizes Performance Monitoring Unit (PMU) to co-operate with LBR. PMU can be configured to set the events it needs to monitor. It contains a counter named Performance Monitoring Counters (PMC). Whenever the configured event occurs, the PMC in the PMU will increase automatically. PMU will trigger an interrupt when the PMC overflows.

TABLE III
BIT FIELD FOR FILTERING DIFFERENT TYPES OF BRANCH INSTRUCTIONS

| **Bit Field** | **Bit Offset** | **Description** |
|---|---|---|
| `CPL_EQ_0` | 0 | branches in ring 0 |
| `CPL_NEQ_0` | 1 | branches in ring 3 |
| `JCC` | 2 | conditional branches |
| `NEAR_REL_CALL` | 3 | near relative calls |
| `NEAR_IND_CALL` | 4 | near indirect calls |
| `NEAR_RET` | 5 | near returns |
| `NEAR_IND_JMP` | 6 | near indirect jumps except near indirect calls and near returns |
| `NEAR_REL_JMP` | 7 | relative jumps except near relative calls |
| `FAR_BRANCH` | 8 | far branches |

According to Intel developer's manual, we configure `IA32_DEBUG_MSR` and `IA32_PERF_GLOBAL_CTRL` to enable Last Branch Recording and Performance Monitoring

TABLE IV
PRECISIONS, RECALLS AND F-SCORES FOR TARGET PROGRAMS

| Type | Target | $p$ | $r$ | $f-score$ |
|---|---|---|---|---|
| Keylogger | *Spyrix* | 0.81 | 0.38 | 0.52 |
| | *Blackbox Express* | 0.99 | 0.03 | 0.05 |
| | *kidlogger* | 0.72 | 0.77 | 0.74 |
| | *Revealer keylogger* | 0.69 | 0.43 | 0.53 |
| Virus | *MyDoom* | 0.72 | 0.43 | 0.54 |
| | *Ratos* | 0.94 | 0.29 | 0.44 |
| | *HIV* | 0.75 | 0.27 | 0.40 |
| Packer | *Armadillo* | 0.84 | 0.62 | 0.71 |
| | *Themida* | 0.98 | 0.09 | 0.17 |
| | *WinUpack* | 0.68 | 0.53 | 0.59 |
| | *UPX* | 0.88 | 0.39 | 0.54 |

Unit. A ground truth needs to capture all types of control-flow transfers. We set up a sub-event mask for the types of instructions that we need. Table III lists the flags that we set in `MSR_LBR_SELECT` so that all these control-flow transfers can been captured. When PMU triggers an interrupt, a handler is designed in kernel that outputs the source and destination addresses into a log file. It shows the ground truth of the basic block sequence, that is, the execution trace.

Although LBR can decide to monitor user-mode program or kernel-mode program through two flags of `CPL_EQ_0` and `CPL_NEQ_0`, it cannot distinguish the branch instructions both generated by the processes in user-mode. Branch instructions in different user-mode processes will all be captured by LBR and then recorded. To solve this, we modify the Linux kernel so that it can mark the Process Control Block (PCB) of the target process. When the target is scheduled in, LBR is turned on and the PMC is reset. It is turned off when the target process is scheduled out. All the address pairs in the log file are generated by the target process.

*2) Coverage:* To measure the percentage of codes covered in the analysis, we calculate the precision ($p$) and recall ($r$). $r$ represents the completeness of Catcher's result. In other words, $r$ defines the ratio of how many executed blocks are captured. $p$ represents the correctness of the result. That is, how many blocks in Catcher's output are actually executed.

According to the exhaustive result in Table IV, Catcher provides an analysis result not as accurate as legacy methods, yet it is still usable. Compared with the methodology that needs to modify or reboot the system, Catcher is completely non-intrusive. This is important in face of malicious processes with anti-analysis capabilities. When the program or system cannot be restarted, our non-intrusive method can provide a practical result. The average value of $r$ is 0.385 (0.457 if remove the abnormally low malware) while $p$ is 0.817. The reason for a low $r$ is that Catcher misses parts of executed code blocks because of no pausing. Unlike intrusive analysis methods, e.g., dynamic binary instrumentation, the lack of event trigger of Catcher makes it impossible for Catcher to find all the executed blocks. In addition, some code blocks are sacrificed to reduce the impact of the synchronization problem, which will lead to a drop in recall $r$. The reduction of recall is actually a kind of compromise to perform analysis in harsh environments. The result also shows precision $p$ is satisfying. A practical precision indicates the feasibility of using cache to analyze program execution. By adopting the heuristics introduced in section III, Catcher mitigates the adverse effect of cache instability. Compared with the ground truth, the inferences on more than 80% of basic blocks is correct.

*3) Smilarity:* Merely using precision and recall to measure the accuracy of the generated control flow is not complete because it does not take execution order into consideration. For instance, a process is executed as *block a*, *block b* then *block c*. However the analyzer comes to a sequence as *block c*, *block b* and then *block a*. In this extreme case, we can present a perfect precision, recall, and $f$-score, while the control flow obtained is actually wrong.

To avoid this situation, we use similarity in NLP to define the difference between the obtained trace and ground truth. We treat the complete sequence as one sentence and every block in it as a word. We first calculate the Levenshtein distance (as known as edit distance) between $O(e)$ and $C(e)$, and then get the Levenshtein ratio to define the similarity.

For a random trace that picks a random branch in every fork, it will come to a similarity much lower than 0.1 if there are millions of blocks within the execution. In this case, if the tracer does not correctly guess the destination basic block of a control transfer, its subsequent guesses are wrong as well because the branching instructions under analysis are different from the actual execution, so the subsequent branches are wrong. It is not the same as guessing a sequence of binary bits at a fixed juncture of execution. The result in Table V shows Catcher's output is actually close to the real execution.

TABLE V
SIMILARITY MEASURES

| Target | Similarity |
|---|---|
| *Spyrix* | 0.51 |
| *Blackbox Express* | 0.25 |
| *kidlogger* | 0.74 |
| *Revealer keylogger* | 0.53 |
| *MyDoom* | 0.54 |
| *Ratos* | 0.44 |
| *HIV* | 0.49 |
| *Armadillo* | 0.70 |
| *Themida* | 0.16 |
| *WinUpack* | 0.61 |
| *UPX* | 0.53 |

## C. Evaluate Heuristics

To evaluate the contributions of each of the five heuristics described in Section III-B, we run experiments to measure the accuracy of Catcher without applying one of them at a time. According to the results in Figure 7, on average all the heuristics improve the precision, except Heuristic 5 (Lost Trace Handling). Heuristic 1 and 3 might cause a slight drop in recall. All of them effectively lead to a better $f$-score.

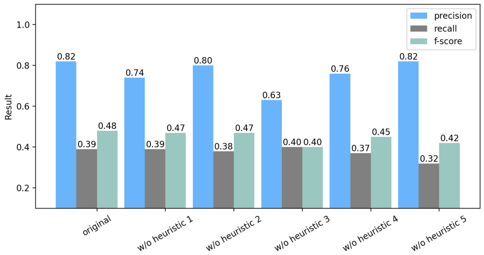

Fig. 7. Average precisions, recalls and $f$-scores w/o different heuristics

Heuristic 5 is meant to re-synchronize Catcher and the target execution by leveraging the stack. It is more beneficial to recall enhancement, meaning that more executed code blocks can be traced by Catcher. The average recall in Figure 7 shows that, although Heuristic 5 causes 0.4% drop in precision, the recall is increased nearly by 20.7%.

### D. Performance

TABLE VI
TIME COST OF DIFFERENT OPERATIONS

| Operation | Time ($\mu s$) |
|---|---|
| CR3 fetching | 3.9 |
| agent switching | 6.9/0.7 |
| memory reading | 1.7 |
| block probing | 0.8 |

*1) Overhead:* Table VI lists the tasks performed by the analyzers and the agents in three phrases of Catcher. In the prologue phrase, the analyzer fetches target's CR3 by *pid*. It takes around 3.9 $\mu s$ on average. Reading 512 bytes of memory costs around 5500 CPU cycles. The key to agent switching is modifying the CR3 of them. When launching the agent for the first time and switching it to the target's memory address space, we need to use the obtained CR3 to load the corresponding *EPT*. This takes more than 2,2000 CPU cycles (6.9 $\mu s$ in our experiment). We do not need to reload *EPT* every time in trace phrase. An agent switching without reloading *EPT* cost less than 0.7 $\mu s$. One probing on a basic block costs 0.8 $\mu s$.

*2) System:* We measure the slowdown caused by Catcher on SPEC2006 benchmarks [5]. We compare the performance overheads with the baseline. The baseline refers to SPEC CPU2006 benchmarks with no Catcher running. The normalized performance cost on average is 2.89% as shown in Figure 8.

We also measure its influence on memory and cache by Cachebench [27]. It incorporates benchmarks for different cache operations like *cache read*, *cache write*, and *cache read/write/modify*. It also tests on `memset()` and `memcpy()` from the C library. Figure 9 shows the performance of cache drops about 8.16% when Catcher is introspecting a process.

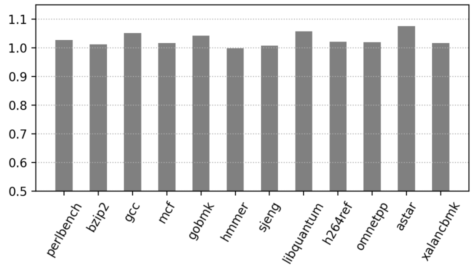

Fig. 8. Normalized benchmark run times when Catcher is running.

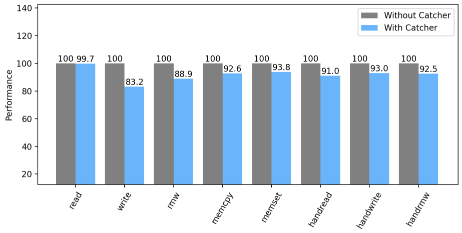

Fig. 9. Performance decline in *CacheBench*

## VI. Discussions

### A. Multithreading

Multithreading enables a process to create several threads that work in the same memory space. Shared memory address space and non-sequential execution cause inconsistencies in cache states. In other words, even with the same code and same input, the cache states obtained vary from time to time, so they cannot be utilized to infer the control flow.

Luckily, there is no real multithreading in Linux. Linux kernel creates Light Weight Process or LWPs to simulate threads. A process containing multiple LWPs is known as a multi-threaded process. Each thread is actually an independent LWP. It has its own process identifier, and can be scheduled by the kernel like a normal process. To locate a thread, we get its LWP through `PS -Lf pid`. Catcher can switch to the CR3 related to this LWP so that this thread can be analyzed.

## VII. Conclusion

To summarize, we propose a novel out-of-VM introspection technique called Catcher to trace a target process from scratch. It utilizes CPU cache to reveal execution situations of the target without interrupting its execution. Due to its non-intrusiveness and transparency, it introduces no side-effects into the system and thus can be applied to anti-analysis malware. According to our experiment, although it is not as accurate as other intrusive analysis tools, Catcher still provides a practical result. Its passive monitoring mode makes overhead negligible. Introducing no environmental variability, Catcher gains an advantage in analyzing malware with anti-debugging techniques.

## Acknowledgment

We thank all the reviewers for their valuable suggestions. This article is partially supported by the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001 and National NSF of China under Grant Nos. 61772266, 61431008.

## References

[1] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28, 2019.

[2] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *International Conference on Advances in Cryptology-asiacrypt*, 2009.

[3] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Deng. Robert. ropecker: A generic and practicalapproach for defending against rop attack.(2014). In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14), February 23-26, 2014, San Diago, CA*. Citeseer, 2014.

[4] Intel Coorporation. Intel 64 and ia-32 architectures optimization reference manual, september 2019. *URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf*.

[5] CORPORATION. SPEC CPU2006 benchmark. https://www.spec.org/cpu2006/.

[6] Franck De Goër, Sanjay Rawat, Dennis Andriesse, Herbert Bos, and Roland Groz. Now you see me: Real-time dynamic function call detection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 618–628, 2018.

[7] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE symposium on security and privacy*, pages 297–312. IEEE, 2011.

[8] Agner Fog. Instruction tables. *Lists of instruction latencies, throughputs and microoperation breakdowns for Intel, AMD and VIA CPUs. Last updated 2011-06-08*, 2011.

[9] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *2012 IEEE symposium on security and privacy*, pages 586–600. IEEE, 2012.

[10] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. Hypershell: A practical hypervisor layer guest os shell for automated in-vm management. In *Proceedings of the 2014 USENIX Annual Technical Conference*, Philadephia, PA, June 2014.

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[12] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[13] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[14] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[15] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[16] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 85–96, 2015.

[17] Kaspersky. Virus.win32.hiv. Retrieved June 2020 from https://threats.kaspersky.com/en/threat/Virus.Win32.HIV/, 2000.

[18] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Design Automation Conference*, pages 1–6, 2016.

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[20] FortiGuard Labs. W32/mydoom.mmm. Retrieved June 2020 from https://www.fortiguard.com/encyclopedia/virus/12110/w32-mydoom-m-mm, 2007.

[21] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 549–564, 2016.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[23] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 416–427. IEEE, 2014.

[24] Scott McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[25] Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE software*, 17(5):33–41, 2000.

[26] Microsoft. Acquiring high-resolution time stamps. https://docs.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps, 2018. Online; accessed 9 September 2020.

[27] Philip John Mucci. Cachebench. http://icl.cs.utk.edu/llcbench/cachebench.html.

[28] Vasilis Pappas. kbouncer: Efficient and transparent rop mitigation. *Apr*, 1:1–2, 2012. http://www.cs.columbia.edu/∼vpappas/papers/kbouncer.pdf.

[29] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent {ROP} exploit mitigation using indirect branch tracing. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 447–462, 2013.

[30] Arun Raj and Janakiram Dharanipragada. Keep the pokerface on! thwarting cache side channel attacks by memory bus monitoring and cache obfuscation. *Journal of Cloud Computing*, 6(1):28, 2017.

[31] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: an efficient" out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 363–374, 2011.

[32] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[33] H Wong. Measuring reorder buffer capacity. http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/, 2013. Online; accessed 9 September 2020.

[34] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[35] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[36] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Recent Advances in Intrusion Detection*, pages 66–85. Springer, 2015.

[37] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. In *Proceedings of the 11th Annual International Conference on Virtual Execution Environments*, Istanbul, Turkey, March 2015.

[38] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[39] Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. Seeing through the same lens: introspecting guest address space at native speed. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 799–813, 2017.