Closing the Execution Gap: Hardware-Backed Telemetry for Detecting Out-Of-Context Execution

Matt Hand, Michael Barclay, Connor McGarr
Mitchell Turner, Tyler Holmwood, John Uhlmann
Robin Williams, Keith Robertson, Isaac Chen
Lewis Zimmerman

Prelude Research, Inc.
research@preludesecurity.com

Abstract

Modern offensive tradecraft has largely abandoned the use of identifiable on-disk artifacts in favor of executing malware directly in memory. While Endpoint Detection and Response (EDR) systems were developed to counter this, they have been drawn into an endless cat-and-mouse game, relying on fixed observation points that can be approached with a view to deceive, or even simply avoided, forcing a reactive and unsustainable cycle of rule-based detection. This paper introduces a novel paradigm for endpoint security by establishing an unavoidable chokepoint at the fundamental point of execution itself. Our system leverages low-level, hardware-assisted telemetry, including Last Branch Record (LBR) and Intel Processor Trace (IPT), correlated with OS context switch and Performance Monitor Interrupt (PMI) events. By observing the CPU directly, we can reliably observe the execution of code residing in private memory, a core attribute of contemporary threats. In response to that observation, the system generates a rich, graph-based representation of the event's full provenance, mapping the inter-relationships between processes, threads, memory regions, and other artifacts to provide deep, actionable insights for security analysts. This approach moves beyond chasing the ever-changing symptoms of malware and instead focuses on the immutable essence of its execution, creating a more durable, proactive, and effective defense against the next generation of advanced threats.

0. Introduction

Technological innovation is often a direct response to a founding problem. This is especially true for endpoint security, where a perpetual cat-and-mouse game unfolds between offensive tradecraft and the defensive products designed to counter it.

The founding problem for the first generation of endpoint security was straightforward: How can we detect malicious files written to disk? Antivirus (AV) products answered this question, establishing a powerful chokepoint at the filesystem. By creating massive signature databases augmented with trained models of known malware and deploying them at scale, AV provided a form of herd immunity against widely circulating threats. While not insurmountable, the deterrent effect of knowing that artifacts written to the file system will be subjected to the scrutiny of AV forced an evolution in attacker methodology, creating a new founding problem for offense: How can we execute malware on a target without writing an easily identifiable PE to disk?

The answer was not merely better payload obfuscation, but the circumvention of the chokepoint itself. The era of "fileless" malware gave rise to a new defensive founding problem: If we can't rely on files, how can we identify malware based on its actions after it is detonated? This question led to

the creation of Endpoint Detection and Response (EDR). EDR's goal was to collect vast amounts of system telemetry, such as process interactions, network connections, and system calls, and write behavioral rules to detect the post-exploitation artifacts generated by malware.

In many ways, EDR succeeded in its initial mission. Heightened security around techniques such as registry run key persistence and LSASS credential theft had a deterrent effect, causing attackers to view these techniques as "off limits" due to perceived operational security risk and leading to their abandonment as viable tradecraft. Unlike AV, however, EDR never established a true, unconditional chokepoint. Instead, it triggered an infinitely regressive coevolution that modern security organizations engage with every day.

Attackers quickly learned to exploit the operating system-imposed limitations of EDR's telemetry. The widespread use of technologies like function hooking by EDR was countered by attackers moving down the stack to make direct system calls (e.g., SysWhispers [1], Hell's Gate [2]), rendering the hooks blind. The sheer volume of telemetry from sources like the Windows registry forced vendors to make tradeoffs about how much telemetry could be collected without impacting the system. With most EDRs operating predominantly in the kernel via the use of a driver, the performance constraints required to sit in line of critical system operations further exacerbated this, requiring the creation of complex software architectures to support the collection of as much telemetry as possible while balancing system performance. Eventually, more advanced evasion strategies emerged to exploit the gaps created in the process of making these tradeoffs, such as insufficient, missing, or unreliable telemetry and flaws in detection logic resulting from inherently coarse-grained false positive reduction strategies.

As new tradecraft and evasions emerged, EDR vendors responded by adding additional telemetry sources and creating more detection rules. This cycle of subtle tradecraft evolution following defensive updates is a classic example of the Red Queen effect. They are over-encumbered by the need to detect both irrelevant legacy tradecraft and a flood of new adversary adaptations, which can be developed with far greater agility and lower opportunity cost. While defenders have to build new detection rules and collection technologies to keep pace, attackers simply need to make better tradecraft decisions. The advent of technologies such as generative AI exacerbates this issue by lowering the skill floor for creating new offensive capabilities and increasing the velocity at which they can be employed during an attack [3].

The prevailing industry narrative holds that malware is becoming increasingly sophisticated, thus justifying ongoing high investment in the development and maintenance of process behavior-based detection rules. However, in practice, small red teams can consistently achieve targeted EDR bypasses using only minor variations on well-known tradecraft. The operational cost of this stalemate is staggering. The industry has responded by hiring more analysts and writing more rules. Mature security organizations now spend monumental resources on tuning alerts and trying to predict which of the thousands of possible techniques or procedures will be used against them, effectively playing a high-stakes guessing game. Commercially available threat intelligence platforms emerged from this uncertainty to offer a way to ease the cognitive burden by providing lookbacks into previous breaches, helping to anchor our guesses in historical fact. However, this approach also suffers from the fact that modern adversaries are adaptive by default.

At the heart of this enduring conflict is a single, fundamental technique that has persisted for decades: the execution of dynamic code. This technique, which was brought to the mainstream by Metasploit's Meterpreter as early as 2004 [4] but popularized by Stephan Fewer's 2008 research into reflective DLL injection [5], bypasses both the file-based chokepoint of AV and the process creation heuristics of early behavior blockers like EDR. Simply put, dynamic code execution refers to the process of running code that was not included in the original image, encompassing techniques such as injection (both local and remote) and exploitation. The nature of the dynamic code is entirely in the adversary's control. It can be something as complex as a fully-staged command and control (C2) agent or a tiny piece of position-independent code (PIC) that performs a very specific action on the system.

For over 20 years, vendors have chased the accidental properties of malware - the specific artifacts created from dynamic code and its resultant post-detonation behaviors - rather than the essential act of executing code from private memory. As soon as one mechanism is put under scrutiny, attackers find a slight variation to evade detection, but the universal truth that an adversary must run their code on your computer is unavoidable.

This brings us to today's founding problem: How can we reliably detect the execution of dynamic code, regardless of the mechanisms used to allocate, write, or trigger it? The answer lies in exploiting an unavoidable chokepoint around the act of execution itself. This paper outlines a system capable of observing any attempt to execute code residing in memory that is not backed by an image file on disk by taking advantage of recent advancements in hardware features and telemetry. We assert that by narrowing our focus through this fundamental chokepoint - one that all modern, sophisticated malware must pass through - we can escape the reactive, rule-based arms race plaguing security operations today.

Our system achieves this by:

- Observing and caching the allocation of private memory in all processes on the system,
- Observing and caching interactions with that memory over its lifetime,
- Observing the execution of instructions in those allocations, regardless of the specific execution mechanism used,
- Correlating execution attempts with the cache of private memory,
- Collecting additional information necessary to describe the provenance of the process, threads, and memory regions associated with a confirmed attempt to execute private memory, and
- Constructing a graph-based representation of these system objects and their inter-relationships

By focusing on the essence of modern tradecraft rather than its ever-changing symptoms, we can move beyond the diminishing returns of the current EDR paradigm and build a more durable and effective defense against modern adversaries.

1. Tracking Execution

There are many native and third-party supplied ways to execute code on an operating system: scripting interpreters like PowerShell and Bash, interpreters like Python, system utilities (i.e., LOLBINs) such as rundll32.exe, and frameworks that support the execution of dynamic code, including .NET and Java. These have primarily been the targets of precise, categorical detection based on process image names and command line arguments, but these require that the adversary predictably invokes them (i.e., with specific arguments) or that they are instrumented to emit specific telemetry (e.g., via the anti-malware scan interface (AMSI)). Because adversaries have so much control over the artifact at this level and can change attributes and remove or mute instrumentation, a problem of infinite permutations exists here that makes cataloging all sources of execution untenable.

At a deeper level, code execution is possible by using smaller measurements of execution, such as the creation of a thread or the queuing of an asynchronous procedure call (APC). These units of execution still provide adversaries total control over the code to be executed - this includes specifying the start address of a target thread (including the process the thread should be executed in) or the target routine the APC should execute in the context of a particular thread.

Some of these units of execution result in the emission of OS-provided telemetry, most often upstream in the kernel. For instance, when a process is created, kernel notification routines registered on the system will be invoked. This allows kernel mode software to undertake point-in-time inspection of the presented characteristics and enforce various security policies on the target process in line of its creation. Process creation events are also emitted by the system over Event Tracing for Windows (ETW), a native system logging facility that can be consumed in user mode.

In some instances where associated telemetry is completely uninstrumented, EDR may inject their own DLLs (e.g., CrowdStrike's "Additional User Mode Data" feature) to intercept calls to these "security-relevant" functions (largely those exported by ntdll.dll) and inspect these calls to

gain insight into their behaviors. Unfortunately, due to the hostile nature of injecting a DLL into a completely untrusted process, these hooks can be trivially evaded since an attacker has full control over the process's address space. To provide both a kernel-protected mechanism and additional telemetry to cover security-relevant functionality, which previously lacked crucial insight (such as manually manipulating and altering the context of a thread), Microsoft instruments this (and other) telemetry through the Microsoft-Windows-Threat-Intelligence ETW provider, a special provider which can be consumed through a Secure ETW channel that is available to certain antimalware vendors.

Detections targeting these smaller base units of execution at a higher degree of resolution are generally more robust because they cover the code execution facilities more broadly without the need to necessarily catalog each tool individually. Unfortunately, these detection strategies are largely beholden to the operating system to provide the quality of data that they require and can choose only from the set of fields given to them, with limited ability to enrich the data. Much like with the first detection strategy involving the cataloging execution tools, vendors must catalog each execution-related operation and rely on the data quality of the telemetry that the OS emits.

In building a system that tracks the execution (a term we'll use to generalize the concept of control flow of all code on the system), the first challenge was to identify the sources of telemetry that provide the most robust account of execution on the endpoint. The challenges with legacy telemetry sources, such as thread creation, are that many valuable event fields, like the thread start address, are under direct attacker control. This allows adversaries to set the terms of if and how they'll be detected. To build the most robust system possible, we must focus our efforts on artifacts that are not in the direct control of an adversary. In our research, we found that both the operating system and specific hardware features offer a prime opportunity to build on this execution chokepoint. Figure 1 shows the inverse relation between the level of adversary control and proximity to the CPU.

Regardless of how an attacker initiates thread execution, the operating system's thread lifetime restrictions will always be enforced. To observe

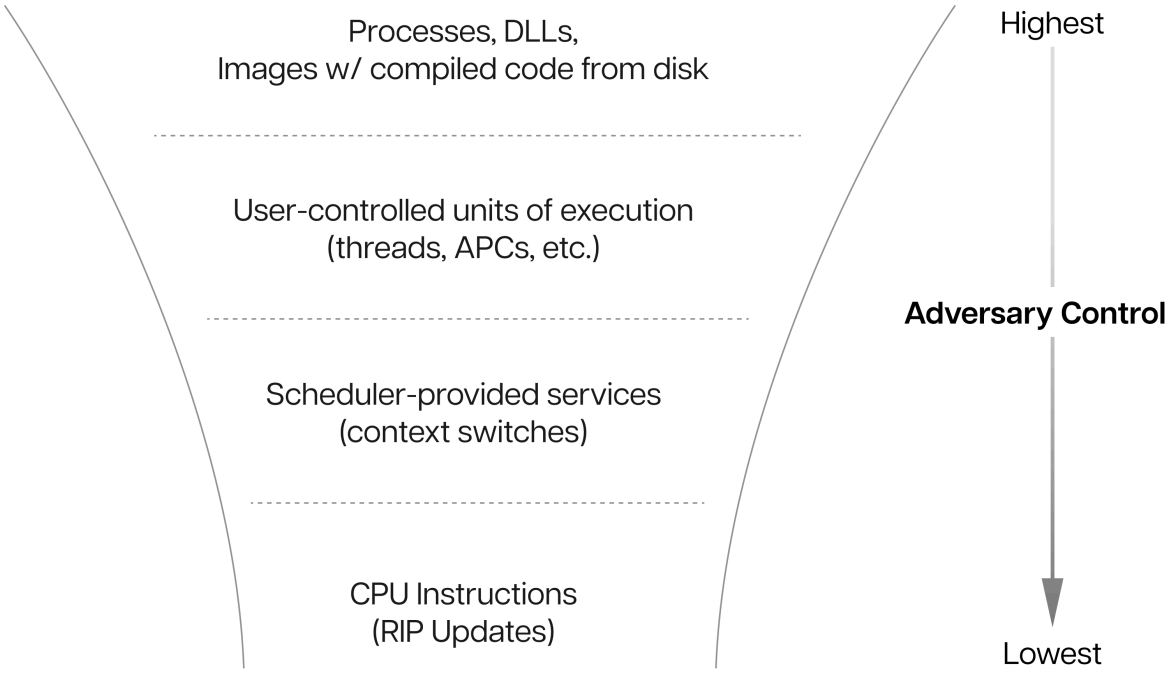

Figure 1: The generation of telemetry focused on the intrinsic properties and defined behavior of operating system components, like the System Scheduler or the nature of specific CPU instructions, is not subject to specific implementation decisions made by attackers.

execution reliably, the strategy involves identifying “inflection point events” and gathering “enrichment events” for detailed information. Two promising, adversary-proof telemetry sources are the thread scheduler and CPU-based sampling.

The research identified two CPU-dependent inspection mechanisms: context switch events and Performance Monitor Interrupt (PMI) events. Both act as “inflection points” to examine thread behavior at specific times, linked to OS-mediated thread lifecycle events. “Enrichment events” supplement these inflection points with crucial context for understanding execution. This section will explain how to establish inflection points using hardware-centric telemetry via Event Tracing for Windows (ETW), enrich these events, and combine them for a strong execution tracking strategy.

1.1 Inflection via Context Switch Events

A context switch is the process of saving the state associated with a currently running thread, retrieving the previously saved state associated with a new thread that needs to run, and finally scheduling/starting the new thread.

Context switches occur when a processor needs to stop executing one thread and begin executing another thread. This process is governed by “the scheduler”, which is not a singular entity on Windows but rather a set of logic distributed throughout the kernel, which is responsible for determining which thread will be given the next opportunity to execute on a processor [6, Chapter 4, p. 214]. There are a few common reasons why a context switch may occur, such as:

- a thread’s quantum - the pre-determined amount of time that the scheduler has allotted to the thread to work before another thread is allowed to execute - has expired (though there is no guarantee that the thread will use its entire quantum),
- a thread voluntarily yields its execution time, such as entering a wait state (e.g., waiting on an event), or terminates, or
- a thread is preempted by a thread of a higher priority.

To describe a situation where we might see context switches in the course of a standard system compromise scenario, consider a long-running Command and Control (C2) agent continuously looping, awaiting tasks from the control server. This open-ended task allows the C2 thread to run as long as the OS permits. However, the OS won’t indefinitely dedicate an entire processor core to a single user mode, low-priority thread. When its quantum expires, a context switch occurs. Since the C2 agent’s work loop isn’t finished, a `CONTEXT` record - a structured representation of the thread’s state at the time of a context switch - is created. On x64 systems, its RIP member will contain the address of the code that was executing at the time of the context switch, which in this case would be in a region of memory associated with the C2 tasking thread.

When the scheduler allows the C2 thread to run again, perhaps due to its higher priority preempting the current thread, another context switch occurs. This will result in switching back into the C2 agent thread, leveraging the previously captured `CONTEXT` record, which provides the information about how to restore this thread’s execution. Specifically, this sets the CPU’s RIP register to now point back to the agent’s memory region, allowing the C2 agent’s tasking thread to resume its work loop. This processor context switching is transparent to the involved threads, giving the impression of uninterrupted execution to the user. Figure 2 illustrates the context switching process.

When a thread’s quantum expires, the scheduler typically manages this through a Deferred Procedure Call (DPC), a software interrupt mechanism. Each CPU maintains a queue of DPCs. In Windows, DPCs are primarily used to enable time-sensitive or Interrupt Request Level (IRQL) specific tasks, such as processing critical interrupts, to defer non-critical tasks, such as thread scheduling, to a later time.

The processor’s clock plays a crucial role in task scheduling. The Interrupt Service Routine (ISR) responsible for handling clock tick updates examines the current thread’s remaining quantum, because clock interrupts occur in the context of whichever thread happens to be running. If the quantum has elapsed, the scheduler must select the next thread for execution on the processor that the interrupt targeted. However, since the clock ISR operates at the high-priority `CLOCK_LEVEL` IRQL, performing non-critical operations, including thread scheduling, would be destabilizing.

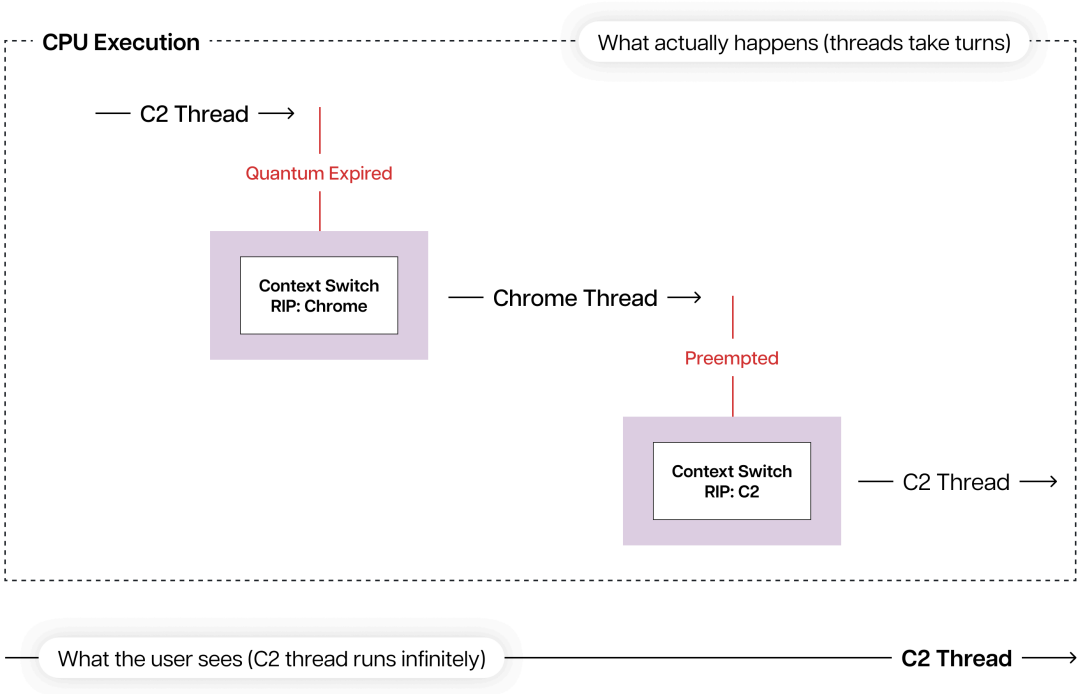

Figure 2: The C2 agent's quantum expires, initiating a context switch that transitions execution to a Chrome thread. This thread is later preempted and execution is resumed in the C2 thread based on its `CONTEXT` structure.

Instead, thread scheduling is deferred by queuing a DPC, which runs at `DISPATCH_LEVEL`. Because interrupts are prioritized, DPCs are only processed once the IRQL drops to `DISPATCH_LEVEL`, at which point the per-processor DPC queue is drained. The scheduling DPC then executes on its designated processor, causing a context switch to the newly chosen thread.

Moreover, each processor maintains an idle thread that runs when no other threads are ready. Before performing its idle duties, this thread also drains any pending DPCs, as other DPCs may have been queued to schedule a new thread. If, during this process, it discovers a runnable thread in the processor control block, it immediately context-switches to that thread. Consequently, many context switches in Windows occur within or adjacent to DPC code paths.

A thread may also voluntarily give up execution, causing a context switch to occur. The simplest example of this would be a thread that calls `kernel32!Sleep`. A more realistic example of this could be when a particular thread is sending and or receiving an Advanced Local Procedure Call (ALPC) message. ALPC is the underlying mechanism that provides much of the transportation layer and foundation for the Remote Procedure Call (RPC) interface. When ALPC needs to send a message, it is capable of letting the requesting thread wait for a particular dispatcher object, like an event object. Since this thread will now be in a waiting state, it will no longer receive CPU time, and a context switch out of the thread will occur, resulting in the thread relinquishing execution to a different thread via a call to `nt!SwapContext`. Listing 1 shows a context switch that was triggered by an RPC client call (`rpcrt4!NdrClientCall2`), which caused ALPC to force the thread to enter a waiting state, which was transparent to the caller of the RPC procedure. When this wait event is signaled, the ALPC thread will be resumed when the scheduler deems it appropriate.

```
Breakpoint 0 hit
nt!SwapContext:
fffff807`b3b39300 4855            push
rbp
2: kd> k
 # Child-SP          RetAddr
Call Site
00 fffffb84`cbac6c18 fffff807`b3b39296
nt!SwapContext
01 fffffb84`cbac6c20 fffff807`b3696e9a
nt!KiSwapContext+0x76
02 fffffb84`cbac6d60 fffff807`b36d9eab
nt!KiSwapThread+0x71a
03 fffffb84`cbac6e70 fffff807`b36f2d29
nt!KiCommitThreadWait+0x1eb
04 fffffb84`cbac6f10 fffff807`b36f5fbc
nt!KeWaitForSingleObject+0x4c9
05 fffffb84`cbac6ff0 fffff807`b3d0d619
nt!AlpcpSignalAndWait+0x4cc

<snip>

10 00000080`00ffdae0 00007ff8`54c4f1a3
RPCRT4!NdrClientCall2+0x1f
```

Listing 1: A remote procedure call (`rpcrt4!NdrClientCall2`) causes ALPC to force the thread to wait, initiating a context switch.

Attackers do not necessarily have control over these various scheduler scenarios from user mode. Since adversary code must run on a CPU for an attacker to achieve their objectives, context switch information about how a particular thread arrived or departed from the processor can be valuable for detection, as it covers the entire time execution on a processor was dedicated to a particular thread. This would guarantee that every thread that receives CPU time can be analyzed, allowing us to inspect what the thread did during its execution and why it left the processor, and providing us with the total coverage of execution tracking we desire. Importantly, context switch telemetry is not available to kernel mode software (i.e., drivers), although some kernel mode software may patch code in the kernel to hook various scheduling routines. On the other hand, user mode software does have the ability to inspect all context switch information legitimately via ETW.

1.2 Observing and Enriching Context Switch Events

Available since Windows XP, context switch telemetry was officially documented with Windows Vista and has traditionally been used for performance profiling and diagnostics. Windows includes additional telemetry to enrich context switch events, such as the associated call stack and information about the old and new thread states, as well as the reason for the context switch. The 24H2 build of Windows, released on October 1, 2024, introduced a new data point: the complete thread context information (the `CONTEXT` structure), which contains the new instruction pointer for the CPU.

ETW is the primary method for gathering this information due to, for the most part, performance and stability considerations. Traditional kernel mode notifications, such as those for threads and processes, are dispatched by iterating through a list of function pointers registered by kernel mode software. Microsoft documents that these specialized functions should adhere to safe call best practices, meaning they should avoid actions like blocking calls [7]. This is crucial because each callback is invoked in line with critical OS operations (e.g., process creation). With potentially 64 functions registered for a single kernel-mode notification, allowing significant processing during a context switch would severely impact system performance, potentially rendering the computer unusable. Since ETW operates asynchronously, it is the only supported mechanism capable of logging context switches.

While context switch logging is possible, the raw metadata, including the old thread and new thread IDs, isn't enough for effective detection. Three vital pieces of information are still absent:

1. The call stack associated with the context switch
2. The process to which the thread being switched into belongs, needed for attribution
3. The complete state of the thread being switched into (i.e., the `CONTEXT` structure)

ETW system loggers can request call stack information for context switch ETW events. This call stack event provides both the call stack of the new thread being switched into (showing where the thread was when it previously yielded the CPU, usually via `nt!SwapContext`) and the process ID of the process housing the new thread. This information is valuable for detection purposes as it offers insight into why the new thread previously relinquished the CPU, which can illuminate the

current context switch. It also solves the process attribution problem.

A third crucial piece of information, available since Windows build 24H2, is the "context registers" ETW event. This event provides the actual state of the thread being switched into, most importantly its instruction pointer, effectively containing the thread's `CONTEXT` structure.

These three events are emitted with the same CPU number and timestamp. This means that by using the CPU number where the context switch occurred, plus the timestamp associated with each of these three ETW events, it is possible to correlate all of this context switch information into a comprehensive event containing the new thread's process ID, call stack associated with the context switch, metadata for the switch, and the actual `CONTEXT` structure for the thread being switched into. Figure 3 shows this multi-event correlation logic.

This correlated context switch can be used to detect the presence of a C2 agent executing in memory. Consider the situation where the C2 agent is performing its infinite tasking loop, awaiting new work to perform on behalf of the adversary. Because the operating system does not want to allow this low-priority thread to monopolize the time of a processor core, it decides that, for one reason or another, it is another thread's turn to execute. At this point, the entire context of the thread, including its call stack and its register state information, which includes the memory address of the C2 agent code currently being executed in the RIP register, is preserved in a `CONTEXT` structure.

When the scheduler determines that it is again the C2 agent thread's turn to execute, the thread's state is restored from the `CONTEXT` structure. This results in the instruction pointer for the processor returning to the point where it was previously executing: the private, executable, and unencrypted region of memory holding the C2 agent code itself. All of the additional information related to the context switch, including the call stack and register state, is emitted by ETW, allowing us to detect the execution of private memory and attribute it to the C2 agent process. In the WinDbg output in Listing 2, we can see that a thread in `StartMenuExperienceHost.exe` was executed on CPU number 4 and transitioned out due to a quantum end. The scheduler then selected the C2 task-polling thread of `malware.exe`, a commercial off-the-shelf C2 agent, for execution. This thread had previously been preempted and is now being rescheduled for execution. We observed the processor loading the malware thread back onto the CPU and resuming its execution.

This provides a strong case for detection. By processing every context switch and examining what the thread to be executed is going to do upon its next allotted runtime on the processor, it is possible to examine every thread's reentry point on a particular processor, for which it is scheduled for execution. Through our research, however, we discovered that there are currently limitations that make relying solely on context switch telemetry for detecting in-memory tradecraft suboptimal to cover all execution across a processor. The two most limiting are:

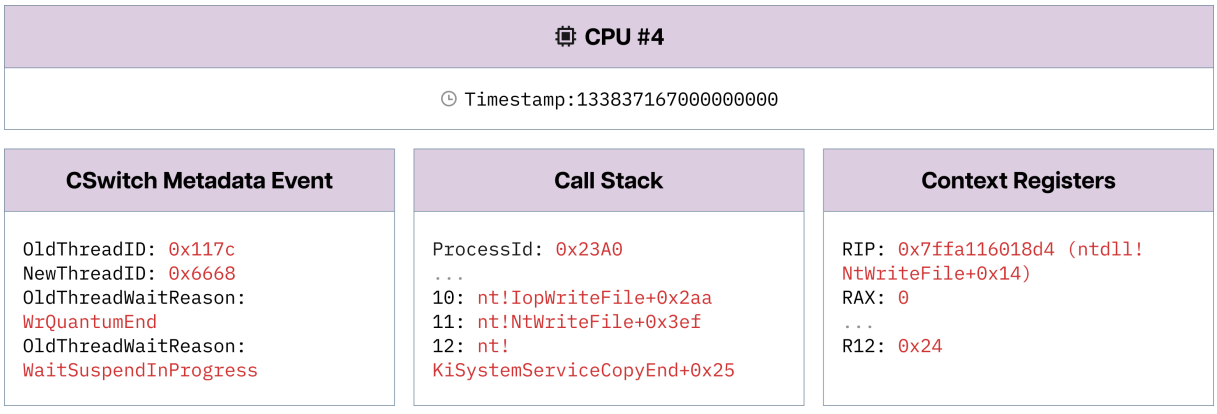

Figure 3: Correlation of ETW Context Switch events (Metadata, Call Stack, and Context Registers) based on CPU number and timestamp.

```
0:003> dx -r2 correlatedContextSwitch
correlatedContextSwitch                 [Type: _CORRELATED_CONTEXT_SWITCH]
    [+0x000] TimeStamp        : 884381691531 [Type: __int64]
    [+0x008] NewInstructionPointer  : 0xa6415f [Type: unsigned __int64]
    [+0x020] TargetProcessName : 0x80021cf7a0 :
"\Device\HarddiskVolume3\ProgramData\malware.exe" [Type: wchar_t *]
    [+0x028] OldProcessName   : 0x80004a66d0 : "\<snip>\StartMenuExperienceHost.exe" [Type: wchar_t
*]
    [+0x030] CpuIndex         : 0x4 [Type: unsigned short]
    [+0x032] OldThreadWaitReason : 30 [Type: char]
    [+0x034] OldThreadState   : 1 [Type: char]

lkd> dx nt!_KWAIT_REASON::WrQuantumEnd
nt!_KWAIT_REASON::WrQuantumEnd : WrQuantumEnd (30) [Type: _KWAIT_REASON]

lkd> dx nt!_KWAIT_STATE::WaitSuspendInProgress
nt!_KWAIT_STATE::WaitSuspendInProgress : WaitSuspendInProgress | WaitFirstSuspendState (3) [Type:
_KWAIT_STATE]

lkd> !vad 0xa6415f
VAD             Level         Start             End              Commit
ffffe78ee1cdb100  4             a40             aa1              98 Private      EXECUTE_READWRITE
```

Listing 2: WinDbg output showing that a thread in `StartMenuExperienceHost.exe` was executed on CPU number 4 and transitioned out due to a quantum end. The `TargetProcessName` value was truncated in this listing.

1. Only user mode threads can be associated with the context registers event. If the thread being resumed has the `KTHREAD.SystemThread` flag set, no context register event will arrive.
2. Depending on the IRQL at which the context switch operation occurs, the context registers event may be emitted through an Asynchronous Procedure Call (APC). ETW manages ETW-adjacent APCs through an undocumented feature called ETW APC pools. Due to the implementation of the ETW APC pool component, periods of high system load can lead to diminished data quality through missing or dropped events. This only affects the context registers event, not the primary context switch event or the call stack event associated with a context switch.

We determined, therefore, that an additional or alternative inflection point must be used to address tracking all execution across the CPU.

1.3 Hardware-Assisted Execution Tracking

While using context switches and the metadata associated with the thread's lifecycle on a processor is effective for detecting adversaries residing in memory in certain circumstances, CPU tracing technologies, specifically Last Branch Record (LBR) and Intel Processor Trace (IPT), offer a lower-level facility to observe execution at an even higher fidelity. Using a combination of context switch and other mechanisms described in this section, it is possible to observe virtually all of a thread's activity on a particular processor during its scheduler-allotted execution window, establishing a thread's execution history. This approach, like context switch events, is available entirely from user mode.

In the context of this paper, we will be referring to LBR and IPT in a 64-bit, Intel, Windows-specific manner, but mainstream processors, such as those from AMD, have an LBR equivalent. ARM64 also has a feature known as CoreSight that offers similar functionality to that of IPT. AMD, at the time of writing, does not have a direct equivalent to IPT. LBR and IPT both require specific versions of Intel CPUs, as these features rely on Model-Specific Registers (MSRs) and architectural changes that are pinned to a CPU version. Earlier CPUs may not have some of these features. For example, CPUs older than Intel 5th generation ("Broadwell") do not have IPT support. LBR has had support for longer, but the specific features leveraged in this paper rely on Intel Pentium 4 and newer.

LBR and IPT have been used in the past for other security-related purposes. Most notably, they were used to provide forms of exploit detection and mitigation that, at the time, were not available in the operating system [8], [9]. An example of this was the use of tracing technologies to perform a custom version of control-flow integrity (CFI) [10]. Given that Windows operating systems, since Windows 8.1, have native exploit mitigations like Control Flow Guard (CFG) and the more recently added shadow stack mitigations (Intel and AMD), such use cases for these tracing technologies are no longer needed.

We instead take a different approach focused on out-of-context execution. Using LBR and IPT, we catalog all of the execution across every thread and ensure that the CPU's instruction pointer never accesses a region of memory that it shouldn't have accessed under normal execution flows. Specifically, this is accomplished in user mode using LBR, IPT, and using the existing ETW infrastructure provided by Windows.

Last Branch Record Tracking

Beginning with the older of the two tracing technologies, LBR is optimized for high-performance applications, making it a valuable tool for efficient execution tracing. It is primarily configured through the `IA32_DEBUGCTL` and `IA32_LBR_CTL` Model Specific Registers (MSR). These MSRs allow for the definition and configuration of various branch tracing properties. These properties include, but are not limited to:

1. Current privilege level (CPL) filtering – enables consumers to choose to trace code in user mode, kernel mode, or both.
2. Branch instruction filtering – allows the narrowing of types of branching operations that are collected to a specific subset, such as disabling tracing of near return branching instructions.

LBR captures branches taken on a per-thread basis in an n-sized set of MSRs, each forming a pair of "from" and "to" addresses contained in `MSR_LASTBRANCH_(N-1)_FROM_IP` and `MSR_LASTBRANCH_(N-1)_TO_IP`, respectively. On Intel's 64-bit Raptor Lake CPUs, there are 32 of these pairs, allowing up to the last 32 branching instruction sources and destinations to be captured. We refer to this collection of MSR pairs as the LBR stack.

Traditionally, a kernel mode driver is needed to configure the MSRs used by LBR. This is done through a "write MSR" assembly instruction (`wrmsr`), which is only available to kernel mode code (i.e., CPL 0) and results in a protection fault when called from user mode. To enable LBR collection from user mode, ETW exposes an interface that will configure the appropriate MSRs on behalf of the caller through an abstracted Windows LBR "options" system call. After calling the configuration function, `sechost!TraceSetInformation`, from user mode, a system call eventually transitions control to the kernel mode function `nt!EtwpUpdateLastBranchTracingConfiguration`. For our use case, we are only interested in user mode indirect branches (i.e., calling into targets which are computed at runtime, such as a function pointer) and near returns Listing 3 demonstrates the process of configuring an existing trace session to receive LBR data.

```
lbrConfig = (FilterKernel |
FilterJcc |
FilterNearRelCall |
FilterNearRelJmp);

error = TraceSetInformation(
          traceHandle,
          TraceLbrConfigurationInfo,
          &lbrConfig,
          sizeof(lbrConfig));
```

Listing 3: Configuring an ETW trace session to collect LBR events via `sechost!TraceSetInformation`

The underlying kernel code then interfaces with optional functionality that resides in the Windows Hardware Abstraction Layer's (HAL) private dispatch table (`HAL_PRIVATE_DISPATCH`). However, LBR recording, initiated by `nt!HalpLbrStartRecording`, will only occur once the trace session is configured with a list of inflection points, referred to as hooks. This raises the question of why this design choice was made.

To retrieve the branches from the LBR stack, a software component - an ETW trace session in this context - must access the data within these MSRs. ETW permits a trace session to specify a list of up to four hooks, at the time of this writing. These hooks are used to signal to the

ETW subsystem to invoke the HAL functionality responsible for retrieving all LBR stack data, via `nt!EtwpTraceLastBranchRecord` function and its subsequent call to `nt!HalpLbrCaptureStack`.

The hook configuration is once again supplied by user mode via `sechost!TraceSetInformation`. These hooks are provided as hook IDs. The hook ID associated with an ETW event is, to simplify, the kernel's abstract representation of a particular ETW event. Listing 4 demonstrates how LBR can be configured to capture LBR stack data upon a context switch (hook ID 1316).

```
hookIds[0] = CSWITCH_HOOK_ID; // 0x0524
(1316)

error = TraceSetInformation(
  traceHandle,
  TraceLbrEventListInfo
  &hookIds,
  sizeof(hookIds));
```

Listing 4: The supplied hook ID value `0x0524`, or `1316` in decimal, configures an ETW trace session to generate LBR stack data when a context switch occurs.

This enables us to retrieve and store the execution history for a thread on every context switch, thereby dramatically increasing our understanding of what the thread was doing during its execution window on a particular processor.

This approach is not without its limitations. The first issue is related to process attribution. LBR, along with other tracing technologies like IPT, are hardware-level features inherently lacking any awareness of operating system abstractions such as processes. Without knowing the process from which the branches originate, LBR stack data, for example, merely represents the most recent 32 call targets across the entirety of the operating system.

One option for process attribution in the context of execution tracing technologies is the ability to include tracing data in the XSAVE area, a region of memory used to store the state of extended processor features, enabled through the XSAVE state-component bitmap (`IA32_XSS[8]` for IPT, `IA32_XSS[15]` for LBR). On context switch, special XSAVE CPU instructions (`xsave[s]`/`xrstor[s]`) are used to store a number of items related to the current thread's extended processor data, most notably, for our use case, the previous thread's trace information.

Although LBR could be a prime candidate to store in the XSAVE area, this would mean that, on context switch, the LBR data associated with the thread leaving the processor would be stored on the stack of this thread. Upon context switching back into this thread, the MSRs would be restored with the thread's LBR data from the XSAVE area. Given that there are 64 MSRs, the LBR stacks alone would take up `320 bytes (64 * sizeof(ULONG_PTR))` on the thread's stack that is being traced, along with the contents of other LBR-related MSRs documented as being saved, for a total of 101 MSRs.

Although Intel does support saving the LBR data to the XSAVE area, the current implementation of LBR on Windows takes a more performant, memory-conservative approach. On context switch at the software level, the context switch code in the kernel (i.e., `nt!SwapContext`) will, if LBR ETW tracing is enabled, clear the LBR stack on context switch, shown in Figure 4. This guarantees that the LBR stack data is only attributable to the thread currently executing on the processor.

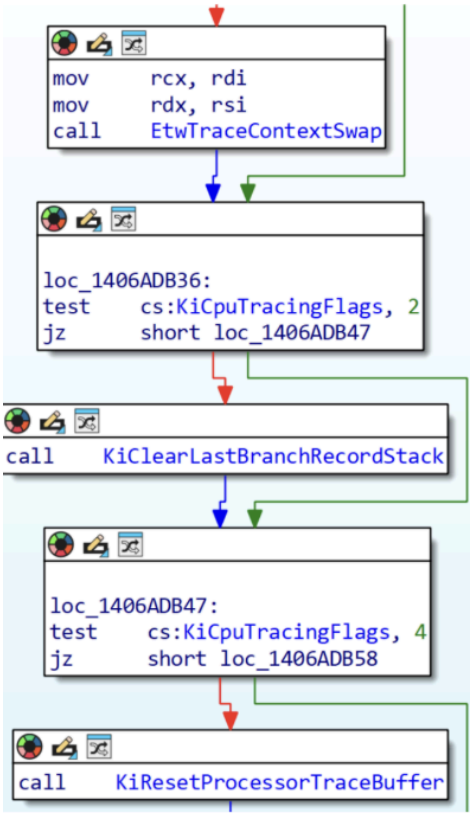

Figure 4: Disassembly output showing that context switches (EtwTraceContextSwap) result in a cleared LBR stack via a call to KiClearLastBranchRecordStack

This means process attribution, while using the Windows-provided tracing features, is simple: the currently executing thread belongs to the process to which the execution trace data can be attributed. Even though execution tracing data, like LBR and IPT, can be stored in the XSAVE area, Windows can simply clear the IPT and LBR data in their respective MSRs, specifically in the case of the Windows-provided tracing features being used, which maintains several benefits. One of the primary benefits is that less data needs to be read from and written to each thread's stack on a context switch. This allows for both the XSAVE area to potentially shrink in size in the case that certain tracing data is either not included or not supported in the XSAVE area, and for other scenarios where Windows needs to manually capture and restore all of the MSRs associated with IPT, all without sacrificing the ability to perform process attribution. This is the answer to a long-standing question which was posed during previous attempts to use execution tracing telemetry for detection: how can we, on context switch, include the IPT and LBR data without access to the context switch functionality in Windows [11], or the XSAVE area? ETW provides functionality to attribute IPT and LBR data to a particular thread even when it is not included in the XSAVE area (either because it was not configured or because it is not supported). Other solutions running in kernel mode likely need to rely on IPT/LBR being configured (and supported) to leverage the XSAVE area to ensure persistence across context switches.

All of this guarantees that the LBR stack data is only attributable to the currently running thread and that attributing to a process is as simple as referencing the current thread's process ID. This ensures that LBR stack data is linked to the operation's target process, and we capture branches for each thread from the time it started running on the CPU to the time it yielded execution, resulting in a context switch. Therefore, enriching context switch events with LBR telemetry immediately becomes an ideal candidate for our detection strategy.

A shortcut in context switching logic in the kernel (`nt!EtwTraceContextSwap`), shown in Figure 5, allows for the opportunistic emission of LBR events in the context switch code path, if the hook is configured.

On Windows, LBR tracing logic attributes process information associated with LBR data to the new thread, specifically during context switches. This necessitates taking additional measures to ensure proper process attribution.

When a context switch occurs, the LBR ETW event is emitted inside `nt!EtwTraceContextSwap`, while the LBR stack data is cleared after it. Consequently, in context-switch scenarios, the LBR data in the MSRs remains associated with the previous thread (as it hasn't been cleared yet), but the attributable process in ETW is that of the current thread's context. This process attribution problem is exclusive to context-switch scenarios; configuring LBR ETW with other event inflection points does not exhibit the same issue.

Our solution to this problem involves maintaining a thread-to-process cache. By correlating an LBR event generated on a context switch, along with its supporting data, based on the shared CPU number and timestamp, we can identify the process to which the previous thread and LBR

```
//
// Flags.LastBranchTracing
//
if ( (loggerContext->Flags & 0x8000) != 0 )
{
  for ( z = 0; z < loggerContext->LbrData->HookIdCount; z = (z + 1) )
  {
    //
    // 1316 == Context Switch "hook ID"
    //
    if ( loggerContext->LbrData->HookId[z] == 1316 )
    {
      EtwpTraceLastBranchRecord(loggerContext, &capturedTimeStamp, newThread, 0x505A05);
      break;
    }
  }
}
```

Figure 5: Decompiler output showing that LBR events are generated (via `EtwpTraceLastBranchRecord`) when the context switch hook value (1316) is present in the `HookId` array

stack belong. This also allows us to retrieve indirect calls and jumps for user-mode code associated with a thread's entire execution time on a CPU. Specifically, this enables us to observe up to the maximum number of supported branches (currently 32) taken by a thread from the moment it last ran on the CPU until it left, encompassing its complete execution duration on that processor. Figure 6 shows the final correlated event, combining the previously described context switch event and the newly attributed LBR stack.

In this case, since the number of LBR stack "frames" was fewer than the maximum amount, this ensures that all of the call targets were captured from the time this thread started until it left the CPU due to preemption. However, there is a slight coverage gap for long-running threads. Threads that perform large amounts of work, along with certain types of specially crafted threads, may perform more than 32 indirect calls during their quantum. In these cases, there is a chance for data loss by "overflowing" the maximum LBR stack size, limiting our visibility to the last 32 branches

CPU #4

Timestamp:133837167000000000

CSwitch Metadata Event	Call Stack	Context Registers
OldThreadID: 0x117c NewThreadID: 0x6668 OldThreadWaitReason: WrQuantumEnd OldThreadWaitReason: WaitSuspendInProgress	ProcessId: 0x23A0 ... 10: nt!IopWriteFile+0x2aa 11: nt!NtWriteFile+0x3ef 12: nt!KiSystemServiceCopyEnd+0x25	RIP: 0x7ffa116018d4 (ntdll!NtWriteFile+0x14) RAX: 0 ... R12: 0x24

LBR Event

```
0: conhost!ApiRoutines::WriteConsoleAImpl+0x2fd
1: conhost!ApiRoutines::WriteConsoleAImpl+0x363
2: conhost!ApiRoutines::WriteConsoleAImpl+0x376
...
11: conhost!ApiRoutines::WriteConsoleAImpl+0x376
```

Figure 6: The three ETW Context Switch events (Metadata, Call Stack, and Context Registers) correlated with an LBR event

taken rather than all branches on a processor. Due to these architectural challenges, we found that we needed a way to query the LBR stack data more than just once for a long-running thread, while it is still running on a processor.

CPU Interval-Based Sampling

Modern CPUs feature a performance monitor unit (PMU) designed to analyze CPU performance and workloads. The PMU can be configured to trigger a Performance Monitor Interrupt, or PMI, under specific conditions, allowing software to investigate particular events. Figure 7 shows a high-level architecture of this component.

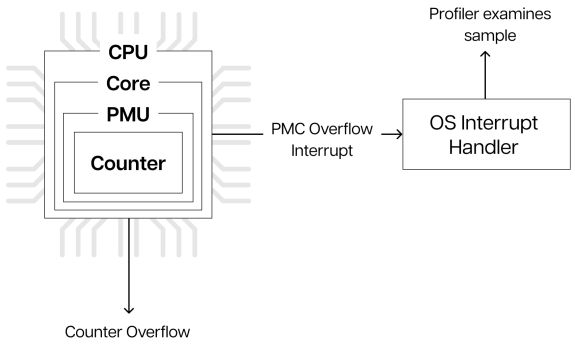

Figure 7: High-level architecture of the Performance Monitor Unit (PMU)

The PMU, while capable of monitoring events such as branch mispredictions or cache misses, is configured in this instance through sampling. This involves setting the PMU to generate a performance monitoring interrupt (PMI) at regular intervals. These PMI events can be gathered using ETW and serve as a crucial inflection point for collecting branch tracing data.

The Hardware Abstraction Layer (HAL) is responsible for maintaining the configured sampling interval. As was the case when enabling LBR events in a trace session, ETW offers an interface, `sechost!TraceSetInformation`, which can be used to globally configure the CPU's interval sampling profile via the `TraceSetProfileSourceConfigInfo` and `TraceSampledProfileIntervalInfo` information classes. Through our testing, we found that the minimum interval is 122.1 microseconds and the maximum interval is 1 second. We also found that it is possible to process, as an example, LBR events at the lowest performance interval supported without saturated ETW event loss. This also provides us with a consistent stream of events that does not require our solution to process every single context switch. Additionally, we still maintain the ability to switch between inflecting on a context switch event and a PMI event based on system context.

Rather than relying on execution tracing data to be emitted on a context switch, we can leverage a special PMC interrupt (PMI) event. This event is received whenever a PMI occurs, which in our case is when the configured interval expires, similar to a clock tick. This will emit an event containing the execution trace data for the currently executing thread at the time the PMC interrupt is received. Since this interrupt occurs up to 8 times per millisecond, it grants us a significant increase in visibility. If a thread runs on a processor for a long time (e.g., until its quantum fully expires), the thread may make more than 32 calls, which in turn causes data loss due to the MSR overflow issue previously mentioned. Given that the PMC interrupt will be issued at a known, fast interval that remains outside of adversary control, this may allow us to "poll" a thread multiple times before it leaves the processor, in turn allowing us to see more than just the 32 branch targets that we get with context switch correlation alone.

```
2: kd> dx -r0 @$curprocess.Name
@$curprocess.Name : svchost.exe

2: kd> k
 # Child-SP          RetAddr
Call Site
; Capturing LBR based on the sampling
interval
00 ffffb381`2a7cdbe8 fffff801`a54e57e2
nt!HalpLbrCaptureStack
01 ffffb381`2a7cdbf0 fffff801`a5407b37
nt!EtwpLogKernelEvent+0xdd2
02 ffffb381`2a7cdd40 fffff801`a5706208
nt!EtwTraceKernelEvent+0x97
03 ffffb381`2a7cdda0 fffff801`a56c6084
nt!EtwpPmcInterrupt+0x78
<snip>
; Interval expiry PMI, interrupting
execution of svchost.exe
0b ffffec89`b41e73b0 00007ffe`857b7bb0
nt!KiInterruptDispatchNoLockNoEtw+0x3c
; COM functioning as normal
0c 00000083`9d27eb08 00007ffe`857bed1b
combase!CComClassInfo::Release
[onecore\com\combase\catalog\class.cxx @
1791]
1b 00000083`9d27f870 00007ffe`862f8860
KERNEL32!BaseThreadInitThunk+0x17
1c 00000083`9d27f8a0 00000000`00000000
ntdll!RtlUserThreadStart+0x20
```

Listing 5: Debugger output showing the generation of LBR Call Stack telemetry when a PMC interrupt occurs

Unlike context switch-based inflection, the PMC-based approach does not require the use of a correlation algorithm for process attribution, such as the one using CPU number and timestamp described previously. When using the PMC interrupt event, execution profiling events contain the correct process ID.

However, for shorter-running threads (i.e., those that leave the processor very quickly), the PMI approach may also result in data loss, as the PMC interval may not expire by the time a thread starts and subsequently leaves the processor. Due to the architectural limitation of LBR's dependence on a fixed number of MSRs to hold execution trace data, a PMI-based approach focused on detecting the activities of long-running threads is more tenable. To account for the case of short-lived threads, we need a technology that can support a larger number of execution tracing events. Through our research, we found Intel Processor Trace (IPT) to be an ideal candidate for such cases.

1.3 Intel Processor Trace

IPT, introduced in 5th-generation Broadwell processors, is a tracing technology that can provide enough data to reconstruct the execution flow of a particular program completely. IPT contains a highly compressed and highly optimized trace buffer format to encode and provide as much data as possible. The trade-off with having such a highly optimized trace buffer is the fact that it then needs to be decoded to be meaningful.

Intel Processor Trace utilizes a few primary MSRs:

- `IA32_RTIT_CTL` - configured IPT's behavior
- `IA32_RTIT_OUTPUT_BASE` - the physical memory address of the IPT data buffer
- `IA32_RTIT_OUTPUT_MASK_PTRS` - offset of the next byte to write in the IPT data buffer

IPT provides the ability to, among other things, capture indirect call targets. What makes IPT more attractive for our purposes is not only that it offers more information surrounding various branching scenarios, but that the amount of data that IPT will report is generally limited only to a configurable buffer size. The IPT buffer is the memory that contains the IPT tracing data. The size of this buffer provides more opportunity to comprehensively consume call targets of branching instructions, as opposed to the architectural limitations of LBR.

Similarly to LBR, IPT telemetry can be consumed entirely from user mode using ETW, with special help from Windows. However, unlike LBR, which is partially documented, the internals of IPT and its use of ETW, including the event data, are largely undocumented. Our research involved reverse engineering the IPT ETW functionality to integrate it into our detection process.

On Windows, IPT requires a minimum buffer size of 4 kilobytes (`PAGE_SIZE`) and supports up to 128 megabytes. Each IPT ETW event carries a buffer of this specified size. While a 128 MB buffer is impractical for real-time event volume, it was initially designed for offline data capture, file writing, and decoding. However, our methodology necessitates runtime inspection of IPT data. We found that, for this purpose, a relatively small buffer, close to the minimum, was necessary for our needs.

IPT tracing can be configured in ETW through the undocumented `EventTraceProcessorTrace-`

```
iptConfig.EventTraceInformationClass = EventTraceProcessorTraceConfigurationInformation;
iptConfig.TraceHandle = traceHandle;
iptConfig.IptOptions.TraceMode = 1;
iptConfig.IptOptions.TraceSessionMode = IptMatchByAnyApp;
iptConfig.IptOptions.TimeMode = IptNoTimingPackets;
iptConfig.IptOptions.MTCFreq = 0;

// For ETW, this has to be set as Reg (Options), not via IOCTL.
iptConfig.IptOptions.TraceCodeMode = IptRegUserModeOnly;

// Set the IPT buffer size using a private function
// 0x1000 = 4KB (4KB minimum, 128 MB maximum)
iptConfig.IptOptions.BufferSize = ConfigureIptBufferSize(0x1000);

status = NtSetSystemInformation(SystemPerformanceTraceInformation,
                                &iptConfig,
                                sizeof(iptConfig));
```

Listing 6: IPT tracing can be configured in ETW via calls to `ntdll!NtSetSystemInformation`

`ConfigurationInformation` value from the `EVENT_TRACE_INFORMATION_CLASS` enumeration, utilizing the `ntdll!NtSetSystemInformation` system call. Listing 6 illustrates this configuration process.

IPT offers flexible configuration options, allowing users to tailor its behavior. For example, users can choose to disregard IPT timing packets, which reduces the amount of data in a trace buffer that is not directly related to call target data. This allows our detection agent to make the most out of the trace buffer. Similar to LBR, CPL filtering is available to limit the trace session to user mode branching instructions, thereby eliminating the overhead of processing irrelevant kernel mode code. The most critical configuration is arguably the IPT buffer size, which stores raw, compressed trace data from the CPU, including encoded call targets and metadata, within the target MSR(s).

Reconstructing program execution flow requires access to the binary being traced. IPT minimizes trace buffer size by utilizing taken-not-taken (TNT) packets, which are bits indicating exercised code branches, leveraging information from the analyzed binary. TNT packets are highly optimized, as no information is produced in them that cannot already be gleaned from the binary itself. Although some Intel CPUs allow disabling TNT packets for consumers not focused on control-flow reconstruction by setting `IA32_RTIT_CTL[55]` (DisTNT), Microsoft's ipt.sys driver, responsible for native IPT event collection and ETW emission, does not currently expose functionality that is capable of configuring IPT in such a manner. Intel states that the capability of disabling TNT packets can result in trace size reductions of 40-75% [12]. `ipt.sys` does, however, provide the ability to disable timing packets to reduce the overall packet volume.

Existing solutions for consuming IPT data at runtime often favor infrequent retrieval and processing of trace data. This preference aims to minimize the overhead associated with retrieving and processing data from MSRs. Instead of continuous operation, these solutions typically employ a larger trace buffer and process data less frequently. Given that decoding these trace buffers can be CPU-intensive, this task is commonly offloaded to the integrated graphics processing unit (iGPU). This strategy generally involves retrieving IPT data only under specific, limited conditions, accepting a larger volume of IPT data during decoding to maximize contextual information. Because this operation is not continuous, the increased buffer size allows for the capture of more data, which can then be efficiently offloaded to a GPU for decoding.

Our research led us to a different approach, finding the inverse to be effective for detection, particularly within a user mode detection framework. We frequently query and decode IPT data, but we do so using the smallest configurable buffer size. This makes our CPU decoding significantly less intensive than decoding a larger buffer while still providing sufficient insight into execution. Given our previous demonstration of configuring ETW to emit LBR events on a context switch, it is no surprise that the same can be achieved with

IPT. We can therefore decode IPT events received alongside context switches, using the smallest possible buffer size, with the understanding that the frequent arrival of these IPT events makes a small buffer feasible. Listing 7 demonstrates this process.

```
iptEvents.EventTraceInformationClass =
EventTraceProcessorTraceEventListInformation;
iptEvents.TraceHandle = traceHandle;
iptEvents.HookIds[0] = CSWITCH_HOOK_ID; //
1316

status = NtSetSystemInformation(
  SystemPerformanceTraceInformation,
  &iptEvents,
  sizeof(ipEvents));
```

Listing 7: Configuring IPT events to generate when a context switch occurs

For IPT configuration, the kernel uses NT's extension host functionality to both load and communicate with the ipt.sys driver, specifically via `nt!EtwpConstructIptData`. Within the Windows kernel, extension hosts offer a shared interface between the kernel itself and other kernel mode images. When ETW determines that an IPT ETW event is required, the kernel will invoke functionality provided by ipt.sys through this interface. Subsequently, when ipt.sys constructs the ETW event, the event write call is also invoked through this interface.

Just like LBR, IPT also has a special condition in the Windows context switch logic that, on logging a context switch event, will opportunistically emit the IPT ETW event if it is configured to do so, shown in Figure 8. This event can, like LBR, be correlated to a context switch metadata event, context registers event, stack walk, and even LBR event, using the CPU number + event timestamp correlation logic mentioned in the prior section. The same process attribution problem also applies to IPT ETW, as there is special logic to clear the IPT data on context switch (`nt!KiResetProcessorTraceBuffer`).

Our detection strategy prioritizes identifying all invoked call targets within a process and examining their memory locations, rather than reconstructing the entire program's flow. By employing a custom decoder for real-time processing of IPT events from user mode, we can optimize it to search for Target Instruction Pointer (TIP) packets exclusively. These packets indicate indirect call targets, including exceptions and interrupts, within an IPT trace buffer. This allows us to consult the memory tracker (detailed in a later section of this paper) to determine if an indirect call inadvertently resulted in out-of-context code execution, such as running code from a data section that has somehow become executable.

In our testing, we captured a single IPT ETW event that retrieved all call targets made by a thread during its entire execution on the processor. Analysis of this event revealed that our IPT decoder successfully decoded 825 indirect branch targets. Consequently, this means that a single IPT ETW event, triggered by a context switch, can

```
//
// Flags.ProcessorTrace
//
if ( (loggerContext->Flags & 0x4000000) != 0 )
{
  for ( j = 0; j < loggerContext->IptData->HookIdCount; j = (j + 1) )
  {
    //
    // Context switch hook ID = 1316
    //
    if ( loggerContext->IptData->HookId[j] == 1316 )
    {
      EtwpTraceProcessorTrace(loggerContext, &capturedTimeStamp, newThread, 0x505A05);
      goto OtherProcessing;
    }
  }
}
```

Figure 8: "Decompiler output showing that, just as with LBR, IPT events are generated when the context switch hook value is present in the `HookId` array"

Telemetry Source	Burstiness	Limitations	Visibility	Efficacy
Context Switch + Context Registers Only	High	ETW APC pool, context switch frequency	High	Lowest
Context Switch + Last Branch Record	High	Context switch frequency, architectural MSR capacity	High	Medium-Low
Context Switch + Intel Processor Trace	High	Context switch frequency	High	High
PMI + Last Branch Record	Low	PMI frequency, architectural MSR capacity	Low (opportunistic)	High
PMI + Intel Processor Trace	Low	PMI frequency	Low (opportunistic)	Medium

Table 1: Comparative analysis of execution telemetry

provide approximately 1000 call targets for examination and scrutiny during each context switch. This quantity is an order of magnitude greater than the architectural limit of 32 branch targets offered by LBR, thus providing an answer to the question we have already proposed: how do we capture all execution across a thread's execution time on a CPU?

We have shown in this section that it is possible to track the execution of code in a highly granular, robust manner that is not under direct adversary control. Table 1 provides a summarization of the telemetry sources we have mentioned, along with their associated efficacy:

It is possible to gain crucial insight into the execution occurring on a processor at any given time using a combination of these events. Specifically, both LBR and PMI can be used to observe direct call targets. On the other hand, IPT and context switches can be used for indirect call targets, exceptions, and other primitives.

Insight into the fact that code is executing on the system is only part of the problem. We need to know what to make of this execution, such as where these call targets live. For that, we need to examine the second component of our system: the memory tracker.

2. Correlating Memory

At rest, code resides on the file system, but at runtime, it is read into memory. This is required because the CPU can not execute code directly from disk due to constraints such as read/write latency and addressability. On Windows, this process is handled by the loader. The loader is relatively complex, but broadly, its job is to parse the executable image and map it, along with any dependencies, into memory before transitioning control over to the application's entry point. Importantly, the regions of memory reserved by the loader, known as image-backed memory, are typically assigned a `PAGE_EXECUTE_READ` (RX) protection mask. This prevents code modification in these memory regions by separating code and data segments. Consequently, data cannot be executed as code, nor can code become writable. This principle remains intact unless the kernel explicitly permits a change (e.g., via a call to `kernel32!VirtualProtect`). The primary exceptions to this are sections explicitly mapped as `PAGE_EXECUTE_READWRITE` (RWX). `W^X` mitigations, such as Arbitrary Code Guard (ACG), provide a mechanism to enforce these memory permission boundaries.

During the application's execution, it may need to work with data that lives longer than the lifetime of the current stack frame, which is unwound during the function epilogue before it returns to its caller. One way the application can do this is to request some amount of private memory - memory not backed by a file on disk. This memory typically holds things like heap allocations to house containers (e.g., vectors, queues), objects, and instances of structs whose size or lifetime aren't known at compile time. By default, these regions of private memory are marked `PAGE_READWRITE`

(RW) as they are not explicitly intended to be executed. However, these pages are often marked executable to facilitate the execution of things like just-in-time (JIT) compiled code, which is used by runtime environments like .NET's Common Language Runtime (CLR) and the Java Runtime Environment (JRE), as well as application frameworks like Chromium which forms the basis of Electron and most modern web browsers. In these cases, the regions tend to be marked with either `PAGE_EXECUTE_READ` (RX) if they are considered immutable, or RWX if they are intended to be modified during their lifetime.

The allocation of private memory results in event generation by the memory manager, a kernel component responsible for, among other things, allocating, protecting, and freeing memory. These events typically come from one of two sources: the kernel provider via the `PageFault_VirtualAlloc` class or the Microsoft-Windows-Threat Intelligence (EtwTI) provider via the `KERNEL_THREATINT_TASK_ALLOCVM_REMOTE` or `KERNEL_THREATINT_TASK_ALLOCVM_LOCAL` tasks. Each event emitted by these providers includes, at a minimum, the identity of the process to which the memory belongs, which is essential for virtual memory addressing, the base address of the allocation, and its size. The benefit to using the EtwTI provider is that it provides much richer context about the allocator and the protection mask at allocation. Allocation events come at a significantly lower volume than hardware-based execution events, well below a million per day, but include far more context, which makes them significantly larger.

The function of the memory tracker is to group observed allocations by process. When an execution event comes into the execution tracker and we unpack the branches, we associate each with the process from which it originated and then reference the memory tracker to see if we have any information about the region of memory in which the branch occurred. One of the most complex problems to solve is that events may come in out of order and with a delay, creating an event ordering race condition. For example, we may receive an event from LBR that includes a branch targeting a memory region for which we have not yet received the allocation event. To solve this problem, we provide many chances to correlate execution events to memory allocations using a sliding generational garbage collection approach, similar to what is shown in Figure 9.

When we do get a match between memory and execution, our focus becomes downselection of events

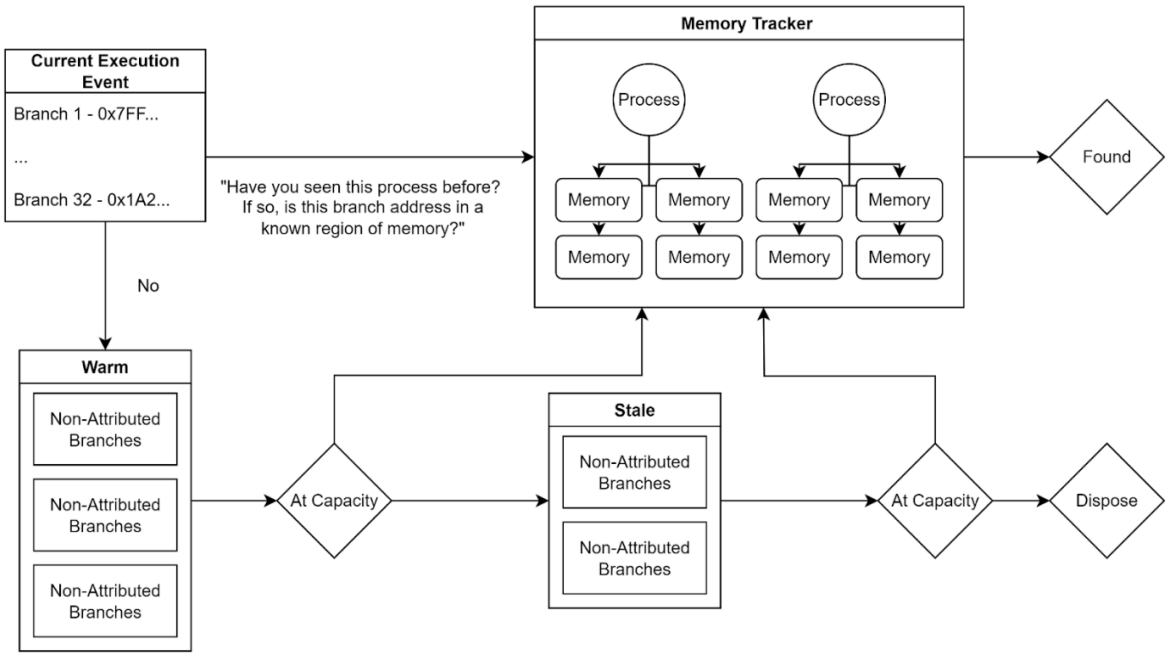

Figure 9: A sliding generational garbage collection approach to correlating branch instructions with known memory allocations

to reduce our total working set. To accomplish this, we make claims about conditions that we believe are untenable states for execution and memory, what we refer to as assertions. One such assertion is that private memory should only be executed in predictable ways. To evaluate this assertion, we evaluate whether the memory address currently being executed is backed by an image. The logic of this process must be incredibly simple due to the performance requirements of the system, so the basic logic is similar to the following code snippet:

```
if memory.is_private &&
(memory.start..=memory.end).contains(&address)
{ /* found */ }
```

Listing 8: Logic to identify the execution of private memory based on virtual memory addresses

When this simple assertion is violated, the agent is responsible for collecting additional telemetry necessary to further describe the execution that just occurred. At this point, we know that private memory is being executed, but we do not yet know if that execution should be considered out-of-context and thus worthy of investigation. To make this determination, we designed a means not just to store additional data that can be queried later, but to understand the provenance of the threads, processes, and memories related to the observed execution attempt.

For execution, changes to the instruction pointer are organized by a thread, which is the basic unit of execution on Windows. Every thread is owned by a process, which in turn belongs to a session backed by an identity. This forms a linear chain of connected objects that effectively assigns an instruction pointer to a user. Memory, on the other hand, has a far richer provenance. Virtual memory regions are owned by a process that belongs to a session, just like with execution. Memory also has actors, processes, and threads that are responsible for allocating, writing to, reading from, changing the protection of, and freeing. Memory can undergo state transitions, such as when its protection mask is changed from RX to RW and back to RX. It also has attributes like size and a lifespan (the amount of time between allocation and freeing). Because each of these pieces of information comes from a system telemetry source, we also get timestamps for each interaction. Together, we can represent a violation event as a graph between the provenance of execution and memory, as shown in Figure 10.

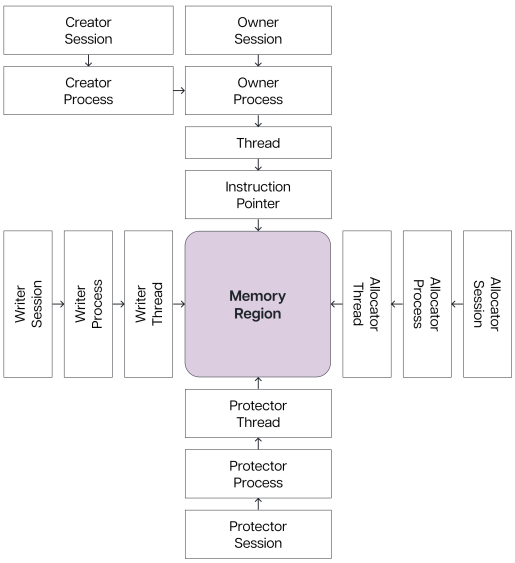

Figure 10: Memory region provenance

In the real world, many of these nodes overlap. For example, the writer and protector may be the same thread, or the session for all branches in the graph may be the same identifier. There are also many instances of different actors, such as when there are multiple writes or protection changes. The complex interconnectedness of these graphs gives us a significant amount of contextual awareness about how execution got to this point; we know that execution was occurring in private memory, the entities involved in preparation and execution, and the different state transitions that occurred. To get a better handle on what might have occurred, we can combine the provenance that we've established with context about the state of the system at the time of execution.

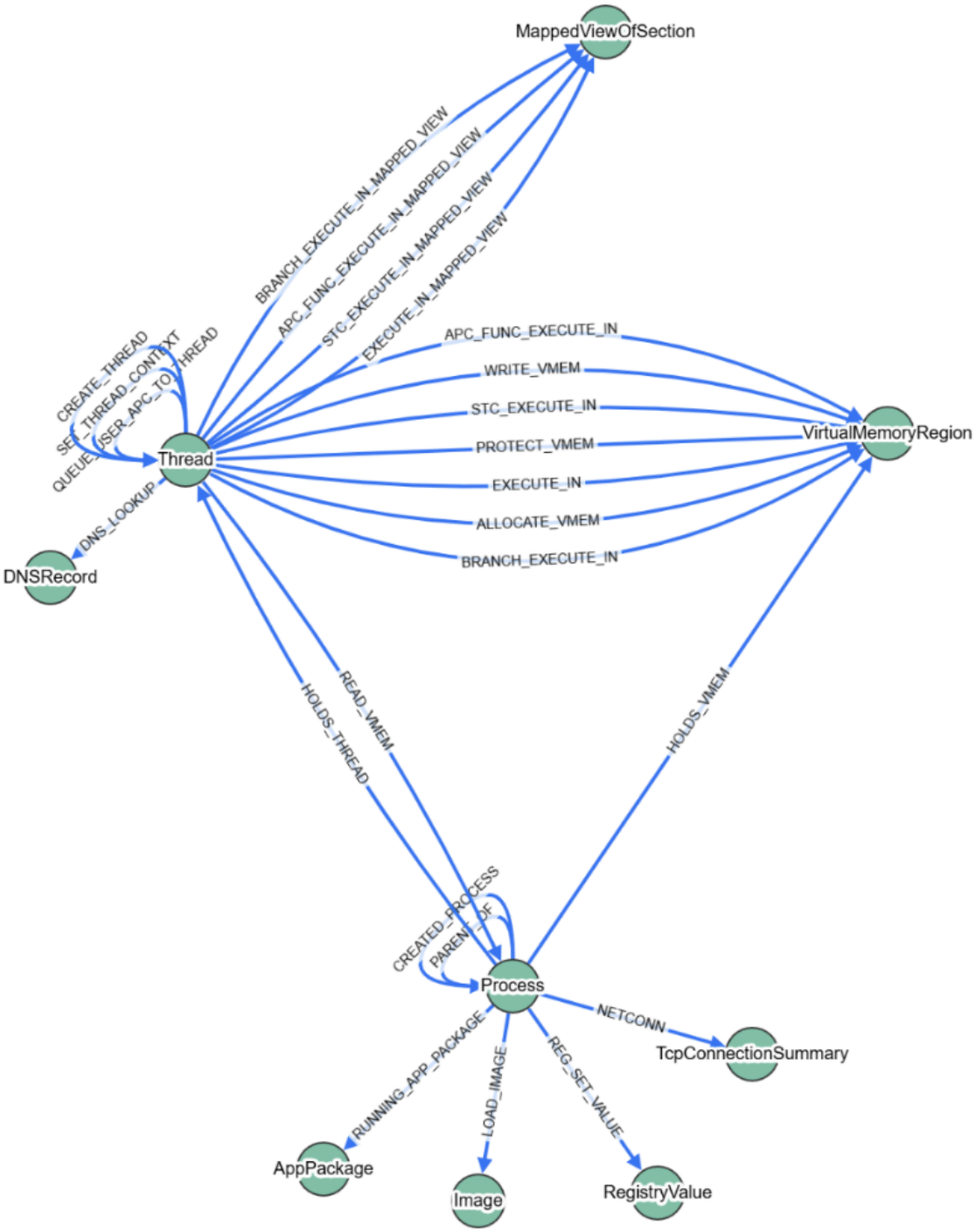

Figure 11: A diagram showing the possible types of edges for a small subset of the system objects covered by the graph schema

3. Local Context Database

This introduces the third component of our system - the local context database. Within the Windows kernel lives a component called the Object Manager. The Object Manager's job is to oversee the creation, security, interactions, and destruction of securable objects. At the time of this writing, there are 74 different object types, including process, files, and tokens, and more obscure objects like asynchronous local procedure call (ALPC) ports used for inter-process communication (IPC) and CpuPartitions used by Hyper-V. Each of these ob-

jects might interact with the others. As described previously, threads can be created and owned by processes, and processes themselves belong to sessions. Threads, being the base unit of work, can also act on objects like files and registry keys. Threads can also act on things not tracked by the Object Manager, such as by establishing network connections or loading an image. This model of objects interacting with one another forms the basis of a graph schema where objects are represented as nodes, and their actions create edges between them. Figure 11 shows a subset of the system objects covered by the graph schema, along with their possible edges.

The operating system furnishes telemetry regarding the creation, destruction, and numerous actions undertaken by various types of objects. The local context database enables the modeling of each object instance's lifecycle and interactions, offering a continuous perspective on the system's evolving state.

Rather than streaming these contextual events to an upstream service for later correlation, we store them in a local embedded database. When a violation of an assertion occurs, we query this database for the relevant nodes based on the provenance of execution and memory. For instance, we will collect information about the process that owns the memory allocation to include any network connections that any of its threads made, or any registry values that it created or modified. The trouble here is that provenance chains can be incredibly long, often spanning five or more generations of processes alone, which can result in massive amounts of arguably irrelevant contextual information being returned from the queries. Managing this requires careful downselection. We don't want to exclude important information that may, at face value, seem irrelevant, such as what network connections a grandparent process made, but we also don't want to process vast amounts of data that doesn't necessarily tell us anything useful, like the images loaded by another child of the grandparent process. Selecting the right amount of information to collect involves using a relevance score that is derived from the nodes' proximity to the triggering execution event and the significance of the object type or interaction that is represented - a sort of blast radius - shown in Figure 12.

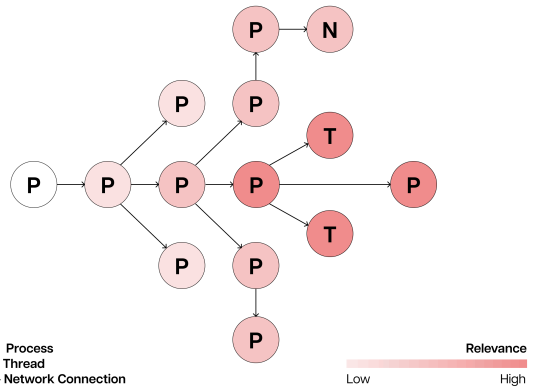

Figure 12: Collecting additional event data, reducing context based on proximity to primary objects

We present these rendered graphs to end users when they evaluate a notification for triage and investigation. The graph shows the system objects involved in the in-memory execution attempt and their inter-relationships, as expressed in the collection of events queried from the local context database through the downselection process discussed above. The specific conditions defined in the graph schema determine which events create new nodes in the graph, update existing nodes in the graph, or form edges between nodes. Because all of the information collected with the notification is presented visually, users do not interact with this data through queries by default. Instead, triage and investigation are primarily a matter of expanding, collapsing, and viewing the details of the presented nodes and edges, as shown in Figure 13. Each event is individually timestamped, enabling us to observe the graph as it evolves over time and easily identify patterns of behavior based on the sequence of specific operations.

4. Testing Results & Case Studies

On a modern Windows operating system, code injection follows four basic steps:

1. *Allocate* an area of memory to contain the instructions
2. *Write* the instructions into the allocated memory
3. *Protect* the area of memory to become "executable"
4. *Execute* the newly inserted instructions on the CPU

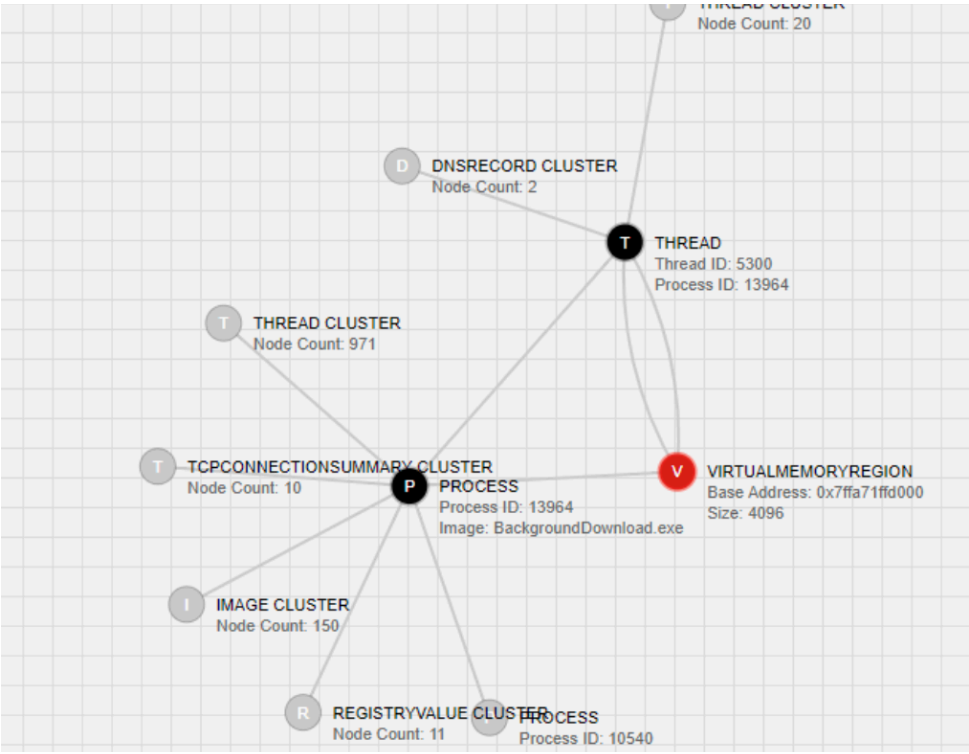

Figure 13: An example rendered graph informed by the graph schema

These steps can be achieved by a nearly infinite combination of operations and API calls on the operating system. Additionally, sometimes these steps are intrinsic: the code may be written to an area of memory already allocated by the operating system, such as a buffer on the heap or the empty space at the end of a mapped image section. Sometimes these steps are combined: an allocation can be given an initial `PAGE_EXECUTE_READWRITE` protection constant, combining the allocation and protection steps. Regardless of the steps used, it is inevitable that the instructions resident in memory will be executed on the CPU.

4.1 Methods

To validate the feasibility of a detection strategy focused on in-memory execution, we devised a series of tests that simulate a variety of common approaches to the execution of payload data written to private memory. Malware authors often use these techniques in an attempt to circumvent traditional behavior-based detections. These tests are formed by dividing the four basic code injection steps into two classes of abstracted primitives: allocation and execution.

Allocation primitives, listed below, represent modularized sets of operations that create memory regions in a process and then write or copy payload data to that region of memory in a way that is compatible with the type of allocation. This comprises the Allocate, Write, and Protect steps of a typical injection attempt. These allocation primitives can be performed both locally, within the calling process, or remotely, into a target process.

Exercised Allocation Primitives

RWX: Memory is allocated with a protection constant of `PAGE_EXECUTE_READWRITE` before writing a shellcode payload using `WriteProcessMemory`.

RW-RX: Memory is allocated with a protection constant of `PAGE_READWRITE` before writing a shellcode payload using `WriteProcessMemory`. The protection constant of the newly allocated and populated page is then changed to `PAGE_EXECUTE_READ` before execution occurs.

RW-RO-RX: Memory is allocated with a protection constant of `PAGE_READWRITE` before writing a shellcode payload using `WriteProcessMemory`. The protection constant of the newly allocated and populated page is then changed to

PAGE_READONLY before an additional change to PAGE_EXECUTE_READ occurs.

Shared Section Mapping: A page file-backed section is created via NtCreateSection. The section is mapped in the calling process with PAGE_READWRITE permission, and the shellcode is written to the mapped section. Finally, the section is mapped in the target process with PAGE_EXECUTE_READ.

Execution primitives, listed below, are also modularized sets of operations. They refer to ways to cause a thread to execute at an address contained in that newly allocated and populated memory region, comprising the Execute step of injection. Execution primitives have a set locality that is used to inform whether the execution can occur in a local or remote process using a given primitive.

Exercised Execution Primitives

Create Local Thread (Local): Creates a new thread within the current process with that thread's start address pointing to the newly written shellcode's entry point.

Indirect Function Call (Local): Casts a pointer to the newly written shellcode's entry point as a function pointer, then calls the function to cause the current thread to branch to that address.

Queue User APC (Remote): Queues an Asynchronous Procedure Call to a thread in the target process with the newly written shellcode's entry point supplied as the callback function associated with the APC. Execution will occur when the target thread enters an alertable state.

Create Remote Thread (Remote): Creates a new thread within the remote target process with that thread's start address pointing to the newly written shellcode's entry point.

Set Thread Context (Remote): Suspends a thread within the target process, overwrites the instruction pointer in the suspended thread to point to the newly written shellcode's entry point, and then resumes the thread

Callback Insertion (Remote): Inserts an object into the remote process that has a callback function pointer directed to the memory region, then triggers the callback.

To verify our detection strategy, we generated a collection of test cases containing all possible combinations of the allocation and execution primitives. Each combination was given a unique Test Case ID (CRUCIBLE-XXX), and the entire suite of combined primitives was executed in batches from a test harness that creates unique child processes for both the source and target of an execution attempt, where applicable. Each of these test cases executes shellcode that loops over branching calls to simulate an in-memory implant calling payloads, then creates a text file on disk to demonstrate successful execution.

The complete list of test cases is contained in the Appendix. Each of these tests were detected by the agent across repeated tests, indicated by the agent generating a notification associated with each test. To illustrate the approach that our agent uses to detect and record these types of in-memory execution attempts, we will discuss two contrasting examples in depth.

4.2 Case Studies

These two test cases were chosen for additional discussion because they represent diametric opposites in terms of observability. The first test case, what we call canonical process injection due to its ubiquity as the simplest example of remote code injection, is both widely understood and readily detected by most modern EDR products in some form. On the other hand, the second test uses an execution primitive that is not well-instrumented by ETW providers. Without the inclusion of hardware-based telemetry like LBR, the indirect function call execution primitive would not be observable using common telemetry collection strategies.

Canonical Injection (CRUCIBLE-013)

The canonical method of injecting code from one process to another is through the combination of the Win32 API functions VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread. This primitive pair is detected by most of the top EDRs today, but we use it as a generally-understood example to demonstrate not only parity but

how asynchronous processing prevents us from falling victim to the sliding context window problem, and that it is possible to detect this injection primitive entirely in user mode. Listing 9 shows how this injection is performed in `CRUCIBLE-013`.

```
HANDLE hProcess =
OpenProcess(PROCESS_ALL_ACCESS, FALSE,
targetPid);

LPVOID remoteMem = VirtualAllocEx(hProcess,
NULL, shellcodeLength, MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);

WriteProcessMemory(hProcess, remoteMem,
shellcode, shellcodeLength, NULL);

CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)remoteMem, NULL, 0,
NULL);
```

Listing 9: C++ implementation of `CRUCIBLE-013`

First, the injecting process obtains a handle to the target process. The injecting process then calls `VirtualAllocEx` using this handle to allocate a region of private memory in the target process; here we are using `PAGE_EXECUTE_READWRITE`, but the results are consistent regardless of the protections. The injecting process writes the payload into the allocated region, and finally, execution is achieved by calling `CreateRemoteThread` with the entry point of the copied code as the `LPTHREAD_START_ROUTINE` pointer.

The agent observes the allocation of a private memory region marked as RWX through a `KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE` EtwTI event and adds a record of this region to the memory tracker cache. Then, the agent receives a `ThreadStart` event through the Microsoft-Windows-Kernel-Process ETW provider. Going through the memory cache, the agent determines that the start address of the newly created thread resides within a known executable private memory region. This assertion causes the agent to generate an "in-memory execution" notification. In a world where the sensor did not collect a `ThreadStart` event for some reason, we would also see evidence of this execution (as a branching instruction) through the hardware-centric telemetry sources presented in this paper.

In response to this detection, the agent queries the local context database and collects any information associated with the event to determine the provenance of the relevant threads, processes, and memory regions before sending it all to the service to generate a graph.

The graph generated by `CRUCIBLE-013` in Figure 14 provides a representation of the in-memory execution attempt responsible for the notification. The `crucibles.exe` process (`PID 24504`) holds a thread (`TID 26444`) that 1 allocated virtual memory at `0x1f6d6df0000` with RWX protection and 2 performed a write action into that region. This region is held by charmap.exe (`PID 15256`), indicating that this notification identified cross-process injection. The same thread (`TID 26444`) in the `crucibles.exe` process that allocated and wrote shellcode to that virtual memory region also 3 created a new thread (`TID 31172`) in the `charmap.exe` process that began execution there.

Since the detection strategy used by the system introduced in this paper is agnostic to the specific method used to cause a thread to begin executing private memory, the graph and contextual events collection are needed to characterize the nature of that execution attempt. Edge labels provide end users with this characterization and allow for a correct recounting of the provenance of both private memory allocations and the threads that executed them.

In Figure 14, the `EXECUTE_IN` edge represents a thread (`TID 31172`) whose `Win32StartAddress` field holds a virtual memory address that falls within the start and end addresses of a tracked private memory region (represented by the `VirtualMemoryRegion` node in the graph). The `Win32StartAddress` value is taken from the `ThreadStart` event that created the Thread node and tells us that the executing thread (`TID 31172`) was tasked by its creator (`TID 26444`) to begin executing private memory immediately upon starting. Since this Thread node has both `CREATE_THREAD` and `HOLDS_THREAD` edges connecting it to two different processes, an analyst can quickly determine that the creator of this thread, rather than the process that holds the thread, is the caller responsible for this notification and needs additional investigation.

In other words, the directional, labeled edges of the graph provide an efficient means to determine the thread(s) and process(es) that actually initiated

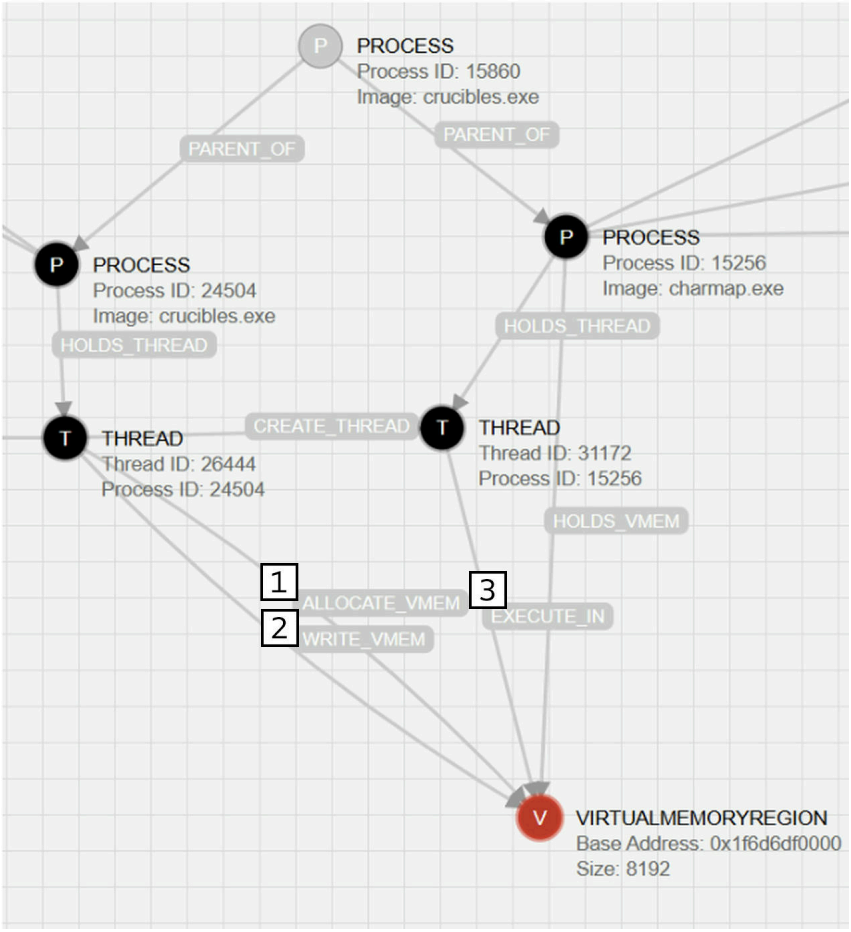

Figure 14: The primary alerting condition for the notification associated with the `CRUCIBLE-013`, rendered as a graph using the graph schema to translate event data to labeled nodes and edges

some noteworthy activity rather than the target of the injection attempt. If the charmap.exe process begins generating additional notifications, it is now clear that the crucibles.exe process is ultimately responsible for that activity and is likely the key to discovering a more meaningful root cause of the compromise.

Additionally, the graph shows non-execution-related nodes and edges that provide additional context about the nature of the threads and processes pictured. This example only shows image loads (represented by the `LOAD_IMAGE` edge) for the two processes represented, but the graph schema will render edges for registry interaction and network connections when that event data is included in the collection.

Indirect Function Call (`CRUCIBLE-005`)

Another common method of executing injected code is via indirect function calls. This approach has the advantage of not requiring the use of any Win32 API calls to perform the execution step, which means that it can never be instrumented from within the operating system. The telemetry for indirect function calls is sourced from the kinds of hardware-centric events discussed in depth in the Tracking Execution section. Specifically, the notification exhibited below was sourced from a Last Branch Record (LBR) event, which contained an entry that showed a thread making an indirect call to begin executing code resident in private memory. We are unaware of any other feasibly-collected telemetry source that can be used to identify this particular execution primitive.

Listing 10 shows the implementation of `CRUCIBLE-005`.

Event: 2a560d77-16ba-41c9-b57b-a14c4d3dbe19	
event_timestamp	0x1dbfb237eee2810
process_id	4052
thread_id	4264
lbr_options	0x8d
branches	0x15464910290 0x15464910290 0x15464910290 0x15464910290

Figure 15: A truncated subset of the last 32 destination addresses of branching instructions taken by the executing thread (`TID 4264`), as reported by Last Branch Record telemetry

```
LPVOID execMem = VirtualAlloc(NULL,
shellcodeLength, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);

memcpy(execMem, shellcode, shellcodeLength);

void (*entryPoint)() = execMem;

entryPoint();
```

Listing 10: C++ implementation of `CRUCIBLE-005`

First, a region of private memory is created as before, although this time in the injector's own process, with a call to `VirtualAlloc`. The shellcode (defined outside of this snippet) is copied into the private region with a simple `memcpy` call. Finally, the pointer to the memory region is cast as a void-parameter function pointer, which has the effect of creating an indirect `CALL` instruction, allowing for control flow of the processor to branch directly to the region of private memory.

The agent once again observes the allocation of a private memory region marked as RWX through EtwTI and adds a record of this region to the memory tracker cache. In this case, there is no ETW event associated with an indirect function call. However, it is a branching instruction and thus is recorded via LBR telemetry. The agent receives the LBR event, unpacks the last 32 branch records included in the event, pictured in Figure 15, and determines that one of the target addresses of a branching instruction is held within a known private memory region. This causes the agent to generate an "In-Memory Execution" notification and gather an associated collection of events from the local context database for graph generation.

Looking at the graph generated by the agent for `CRUCIBLE-005` in Figure 16, we can see that it is seemingly simpler than the graph from the previous example. The process `crucibles.exe` (`PID 4052`) holds a thread (`TID 4264`) that carries out the full chain of common code injection steps independently. This thread 1 allocated memory at `0x15464910000` with an RWX protection constant. It then 2 wrote 8192 bytes at that same virtual memory address. This region is held by crucibles.exe, the same process that created and owns `thread 4264`, indicating that it is local code injection. Finally, the same 3 thread executed instructions within that newly populated memory region based on an execution primitive that was only observed by an LBR event (based on the `BRANCH_EXECUTE_IN` edge label). From these relationships, it is clear that `crucibles.exe` has allocated, written to, and executed private memory within its own process.

Once again, the edge labels and the nodes they connect provide an efficient means to determine that the crucibles.exe process is injecting into itself and should be the primary focus of any additional investigation. This behavior is often associated with loading and executing post-exploitation tooling like Beacon Object Files (BOF) or similar COFF-based resources. In a real-world scenario, the behavior of other threads in this process and the existence of additional notifications for each time one of these COFF files was loaded and executed locally would provide additional circumstantial evidence to determine further if it is a long-running agent-hosting process. In such a scenario, additional contextual nodes that track network connections from the process would also appear in

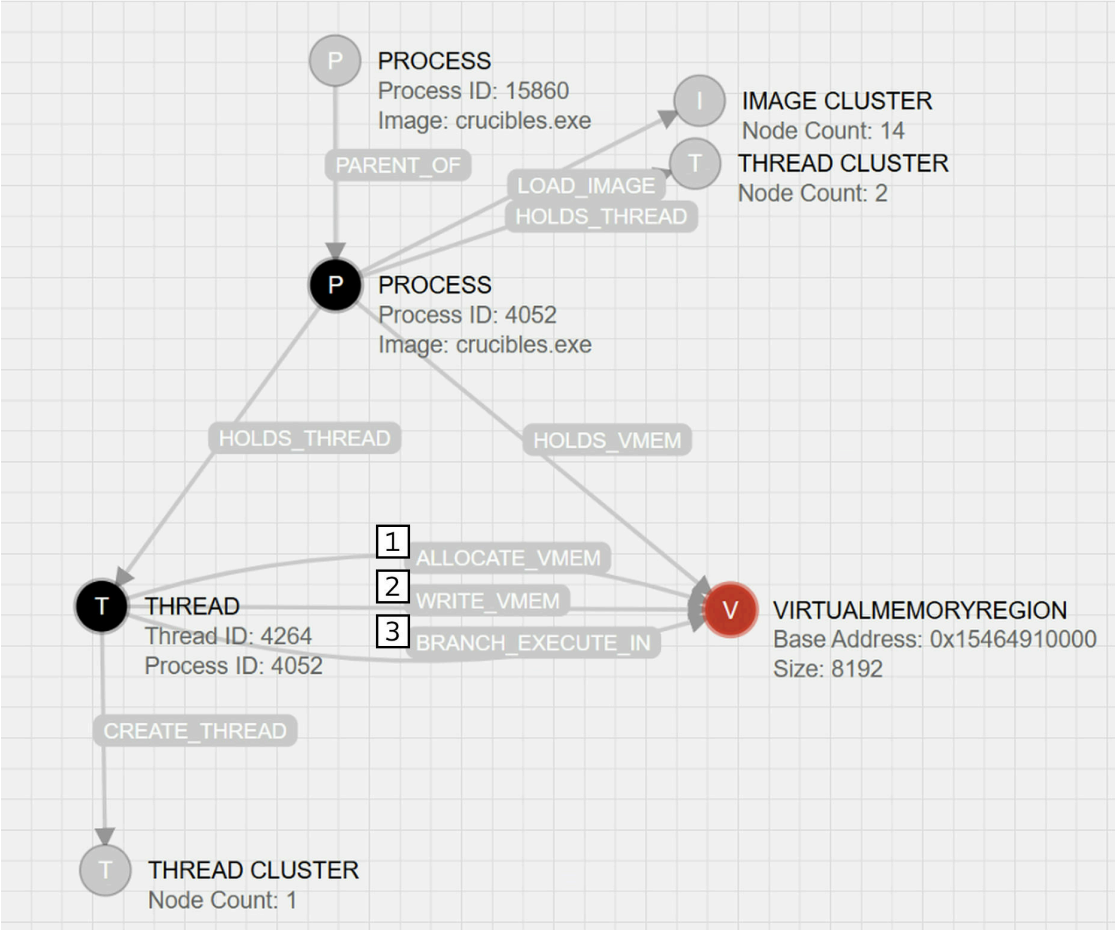

Figure 16: A graph showing the system objects, and their inter-relationships, involved in the notification for `CRUCIBLE-005`

the graph and may reveal additional evidence of a long-running agent.

The `BRANCH_EXECUTE_IN` edge is specific to execution attempts observed via an LBR event. This label is useful not because it directly characterizes the mechanism used to cause the thread (`TID 4264`) to execute this private memory, but because it is an execution primitive that Microsoft cannot instrument via EtwTI. This lack of instrumentation and subsequent loss of visibility may represent an execution primitive that was previously unaccounted for.

LBR and other hardware-oriented telemetry sources discussed in this paper are extremely valuable for observing individual execution attempts that are not otherwise visible to security software, but are agnostic of the execution mechanism used to cause a specific branch to be taken. This is a huge boon to detection because it provides a built-in defense against the uncertainty of offensive tradecraft. We do not have to predict every possible execution mechanism to trace execution in granular detail.

While not pictured here, additional notifications generated for this process will be merged into the graph in Figure 16 as they are shipped to the cloud service. If this notification did actually represent a long-running command and control agent loading and executing something like a BOF, we would expect to see repetitive execution attempts following the same pattern over time. The newly expanded graph would show multiple threads with the same provenance executing private memory with the same sets of edges. In a normal operational context, the in-memory execution chokepoint offers the ability to identify not just initial execution, but also the execution of post-exploitation tooling loaded into memory. This model of a limited agent armed with a capable COFF loader to carry out the majority of post-exploitation tasks has become the norm for many red teams and attackers in the wild.

5. Future Work

While our approach to detection shows significant promise, there remain significant hurdles to overcome, particularly in the areas of false positive management and expansions into execution primitives not based on the execution of private memory.

5.1 False Positives

It would be logical to traverse these graphs to identify patterns of malicious behavior and trigger an alert, but this puts us back in the space of cataloging and predicting adversary tradecraft, this time at the edge rather than upstream in a dashboard. There are, as mentioned in previous sections, a significant number of cases that will generate a false positive - JIT and copy-on-write image modifications being the largest sources. Because each instance of execution that violates our assertion results in a collection being generated, we can easily become inundated with unqualified false positives that must be manually sifted through. This historically would be considered a killshot for any detection tool, and rightfully so. Under the legacy model of detection, where everything is treated as malicious until proven otherwise, this would exhaust analysts as they work their way through a never-ending fire hose of alerts, eventually becoming fatigued and less rigorous in their review, potentially leading to missed true positives.

Instead, we believe that these graphs can do something far more interesting: highlight emergent patterns of behavior. The result of the combination of provenance and contextual information forms what we call collections and is essentially a complete snapshot of the state of the system at the time when the code was executed in a way that violated our assertion. These collections can be used to describe the general behavior of an application and its lineage in a way that allows us to say "this application is associated with these behaviors that we observed by the time of violation," importantly, at each instance of a violation.

Software is, for the most part, just repeated executions of the same finite number of code paths. For instance, video conferencing software spends most of its cycles encoding, transmitting, receiving, and decoding audio and video streams. Code paths that result in a violation of one of our assertions will likely do so in the same way each time - `functionA()` calls `functionB()`, which calls into private memory for one reason or another. This happens continuously, creating collections that look very similar across multiple instances of the application.

Our core hypothesis around the mitigation of false positives is that they are not a nuisance to be eliminated, but rather critical data points that help us gain a deeper understanding of what "normal" behavior on the system looks like. This is grounded in a principle of benign prevalence: the overwhelming majority of system activity is non-malicious. Malware, on the other hand, constitutes a statistical outlier that will stand out when evaluated against the benign behavior observed by it. Our continued work on signal extraction is built around two basic pillars: clustering and counting.

5.2 Clusters

Our ongoing research focuses on self-supervised techniques to characterize normal system behavior without relying on labeled data or predefined threat signatures. Rather than attempting to detect specific forms of malicious activity, we are exploring the use of embedding-based representations to capture patterns of legitimate activity directly from noisy, real-world telemetry. This approach involves constructing a graph-aware abstraction of system events, designed to retain both the local semantic meaning of individual events and the broader contextual relationships among them.

The goal is to learn a compact and generalizable representation of benign behavior, enabling the identification of outliers based on their divergence from learned norms. To this end, we are experimenting with techniques that enhance the fidelity of the learned embeddings, including multi-head attention mechanisms and the incorporation of broader collection-level context during training.

Initial findings are promising, indicating that this direction can effectively highlight subtle, low-signal anomalies—such as stealthy in-memory attacks—while significantly reducing false positives. This suggests the potential for a more robust

and adaptable anomaly detection framework that can better differentiate truly suspicious behaviors from the benign-but-unusual patterns that frequently trigger traditional alerts.

5.3 Counters

In our ongoing effort to create more resilient behavioral models, we are exploring techniques adapted from the field of performance profiling to fingerprint dynamic code. Traditional profiling methods like calling context trees are a powerful starting point, but they are insufficient for security contexts where the focus is on isolated, potentially malicious code rather than whole-application performance. We are investigating a novel approach that pivots the analysis around the dynamic code itself. Instead of a single-rooted call graph, we propose partitioning execution traces at each dynamic memory region, allowing us to grow a "trifocal" graph that captures the code's memory provenance, its execution primitives, and its subsequent interaction with the operating system.

This method aims to overcome the brittleness of full execution traces by collapsing complex call stacks into more stable, high-level fingerprints based on public API transitions. The central hypothesis is that legitimate dynamic code, such as that from JIT compilers, exhibits predictable and limited patterns across these three perspectives, while malicious code will create detectable anomalies. By modeling these distinct incoming and outgoing execution paths, we believe we can establish a stable baseline of normal behavior, providing a more nuanced foundation for distinguishing legitimate JIT activity from advanced in-memory threats and thereby reducing false positives.

5.4 Challenges

There are two primary problems with these approaches: masquerading and baseline stagnation.

Adversaries have a long track record of attempting to "blend into the noise" as a means of evading more precise detections. For instance, if a detection rule for extracting credentials from memory has an exclusion for processes called `GoogleUpdate.exe` to mitigate false positives, an adversary would reasonably rename their executable with that same name or inject into the legitimate process. Adversaries can exploit logical gaps in detection rules because the attributes are in their control - process names, command line arguments, network addresses.

As mentioned previously, the goal of our system is to detect adversaries at the lowest possible level in a manner that is, ideally entirely but often practically, outside of adversary control. Hardware provides us with many such opportunities. For example, a collection of branches that describe the flow of execution through a program is practically outside of adversarial control because their ultimate aim is to trigger the execution of their code, not to exercise the application's normal logic. The value in the RCX register is practically within the adversaries' control, but modifying its value could corrupt the first parameter of a function call under the `__fastcall` convention used by Windows, causing an error. The signer of an application is, barring vendor certificate theft, outside of an adversary's control, but they could inject into a legitimately signed application, which could result in them being detected.

Rather than using one of these attributes or data sources as the sole source of fidelity in a stream of behaviors, we can leverage multiple attributes in the collection to increase the complexity of our dataset dramatically. This has many parallels to how depth maps are formed in facial recognition software like FaceID [13], where two humans may share the same depth of features at the same place, but it would be nearly impossible to scale that out to all sample points. By combining attributes, we can make statements like "The RDX register was this value, the last four branches came from this module and were at these offsets, and the grandparent process was signed by this company," which drives the complexity of masquerading up to a point where adversaries can't control for all variables. Extracting and assessing normalcy against a set of features based on attributes outside adversary control will increase the robustness of our detection approach.

Maintaining a baseline also presents a significant challenge. Organizations adapt over time; software is onboarded and offboarded, workflows change, and the operating system updates. Many applications that rely on established organizational base-

lines establish a view of normalcy once, often at the start of deployment, or on some regular time interval (e.g., quarterly). The difficulty in these approaches is that organizational drift is a constant process, and in waiting for some set interval, it becomes increasingly irrelevant as time goes on until the baseline is refreshed. To avoid stagnation in our data, we are using a rolling baseline. At its most foundational level, this process involves the use of a sliding context window, similar to the functionality of a ring buffer. For example, we can use a time-bound window of 30 days, which would allow us to collect 30 days of contextual data to formulate a baseline for the month. Then on the 31st day, the 1st day in the window - the most outdated data - will be evicted. In situations where event volume is significant enough that a time-based approach is untenable, we can employ a capacity-based approach that functions identically. Both approaches ensure that only the latest, most relevant data is included in the baseline, reducing the likelihood of organizational drift.

5.5 False Negatives

The approach outlined in this paper covers only one assertion: private memory should only be executed in predictable ways. Our objective is to build a solution that is intolerant to false negatives for memory-based threats. While this does provide coverage for a majority of modern adversary tradecraft, ranging from the execution of C2 agents and post-exploitation tooling to exploitation primitives based on code execution, there are apparent gaps that require the employment of other assertions.

One example of this is interpreter-based malware, such as that built using .NET or Python. These dynamic runtime environments interpret bytecode, which permits dynamic control flows while all execution appears image-backed. There are many potential avenues to detect adversaries' use of interpreters and JIT engines during their operations, but this does pose a challenge requiring the inclusion of additional assertions beyond those covered in this paper.

Another example of this is return-oriented programming (ROP), call-oriented programming (COP), and other types of "weird machine" techniques [14]. These techniques operate by leveraging existing, legitimately executable code fragments (gadgets) within image-backed memory rather than relying on private, non-image-backed regions. As a result, our primary assertion about the execution of private memory is inherently ineffective against such methods. ROP, COP, and related primitives reuse legitimate code sequences already loaded into memory by the operating system or applications, rearranging and chaining these sequences to perform arbitrary malicious operations without introducing explicitly executable private memory. However, the execution flow of chained gadgets is in itself an anomalous pattern of behavior. The false positive mitigation approach built around the prevalence of code paths described previously offers one potential solution to this, as the return instructions used by ROP and the call instructions used by COP are both branching instructions and therefore subject to collection by technologies like LBR.

There is also the possibility that a regression in tradecraft occurs in response to increased pressure on in-memory techniques. In this case, we advocate for a shared responsibility model. Antivirus has a strong chokepoint built around the filesystem and benefits from global scale. This makes it ideal for intercepting, scanning, and establishing the global prevalence of file-based threats that would not trigger a detection based on executing private memory. EDR provides a second, multifaceted layer of defense. Its stream of behavioral events can be used when scoping and investigating an incident, and its real-time response capabilities can be used for forensic analysis and live response. This would allow security operations centers to leverage multiple approaches when detecting and investigating endpoint compromise, while allowing complementary solutions to focus their efforts on building more robust coverage within their areas of expertise.

In summary, there exist in the modern adversary's toolkit opportunities to execute code in memory that would evade the private memory-based assertion outlined in this paper. A majority of our current and future research is focused solely on the identification and elimination of these bypasses in the most robust way possible. Initial approaches to mitigation show promise, but there is still more to do.

6. Conclusion

In this paper, we have demonstrated the implementation of a novel endpoint detection system capable of observing attempts to execute private memory using artifacts of execution that are outside of an adversary's control, providing an unprecedented opportunity to detect the presence of in-memory threats entirely from user mode. The example scenarios and graph-based output presented in the Use Cases section validate the efficacy of this system's ability to go beyond observing execution, correlating the resulting data with tracked regions of private memory, and generating a notification. It also provides end users with labeled graphs that summarize the processes, threads, and virtual memory regions whose mutual interactions caused the notification to be generated, along with critical context used to gain a deep understanding of the system when the execution of private memory was detected. These graphs allow for a straightforward triage process where the provenance of those system objects is clearly defined through a series of directional, labeled edges between nodes. The validity of this graph is mediated by a graph schema that describes which events, collected by the agent, should create new nodes in the graph, update existing nodes, and draw edges between specific nodes.

In describing this system, this paper also outlines multiple approaches to instrumenting hardware-assisted execution telemetry via ETW. We assert that this kind of telemetry requires the establishment of an inflection point, which defines when the system's sensor has the opportunity to inspect the execution path (whether current or historical) of a given thread. This paper suggests two candidate events outside of the adversary's control for inflection as a starting point: (1) context switch events and (2) PMC interrupt events. While these events allow us to inspect execution at a specific point in time, they both require additional enrichment from other CPU hardware-mediated event sources, namely (1) Last Branch Record events and (2) Intel Processor Trace events. The Approach section details strategies for combining these various events to trace every execution attempt across all threads on a Windows system.

This source of telemetry is not subject to tradecraft choices made by defenders because it relies on facts of a thread's lifetime governed by the operating system. A user mode application has no means to cause a thread to execute at an arbitrary address in memory and govern the way that thread is scheduled (and generates context switches) or the way that the system processes Performance Monitor interrupts.

Even an entirely unknown execution primitive will still cause a thread to be scheduled, switched, and interrupted in this manner when it goes to execute. This lack of reliance on predictability in offensive tradecraft selection confers huge benefits for robust, execution-centric detection strategies and establishes a chokepoint through which attackers must pass.

More work is necessary to extract true positive signals while also reducing the total number of false positives. Additionally, more work is needed to extend this detection strategy to other out-of-context primitives beyond this first pass at observing data being treated as code, or code becoming writable.

7. References

[1] J. Thuraisamy, “SysWhispers: AV/EDR evasion via direct system calls.” [Online]. Available: https://github.com/jthuraisamy/SysWhispers

[2] am0nsec and smelly__vx, “Hell's Gate.” [Online]. Available: https://github.com/vxunderground/VXUG-Papers/blob/main/Hells%20Gate/HellsGate.pdf

[3] “Hardware-Enhanced Security: Leveraging Intel CET and Intel PT for Advanced Exploit Detection.” [Online]. Available: https://arxiv.org/html/2506.07586v1

[4] M. Miller, “The Metasploit Meterpreter.” [Online]. Available: https://hick.org/code/skape/papers/meterpreter.pdf

[5] S. Fewer, “Reflective DLL Injection.” [Online]. Available: https://www.exploit-db.com/docs/english/13007-reflective-dll-injection.pdf

[6] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals, Part 1*, 7th ed. Microsoft Press, 2017.

[7] Microsoft, “Windows kernel-mode process and thread manager.” [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-process-and-thread-manager#best-practices-for-implementing-process-and-thread-related-callback-functions

[8] C. Pierce, M. Spisak, and K. Fitch, “Capturing 0days with PERFectly Placed Hardware Traps.” [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Pierce-Capturing-0days-With-PERFectly-Placed-Hardware-Traps-wp.pdf

[9] T. Kreuzer, Y. Shafir, S. Tanda, and B. Foster, “Introducing Falcon Hardware-Enhanced Exploit Detection.” [Online]. Available: https://www.crowdstrike.com/en-us/blog/introducing-falcon-hardware-enhanced-exploit-detection/

[10] “Processor tracing,” in *2012 IEEE Hot Chips 24 Symposium (HCS)*, 2012, pp. 1–31. doi: 10.1109/HOTCHIPS.2012.7476487.

[11] R. Johnson and A. Allievi, “Harnessing Intel Processor Trace on Windows for Vulnerability Discovery.” [Online]. Available: https://www.fuzzing.io/Presentations/Harnessing%20Intel%20Processor%20Trace%20on%20Windows%20for%20Vulnerability%20Discovery%20-%20rjohnson.pdf

[12] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developer's Manual.”

[13] Apple Inc., “Face ID Security Guide.” [Online]. Available: https://www.apple.com/business-docs/FaceID_Security_Guide.pdf

[14] T. Dullien, “Weird Machines, Exploitability, and Provable Unexploitability,” 2017. doi: 10.1109/TETC.2017.2785299.

Appendix

Full List of In-Memory Execution Tests

Test Case ID	Execution Primitive	Allocation Primitive
CRUCIBLE-001	Create Local Thread	RWX
CRUCIBLE-002	Create Local Thread	RW-RX
CRUCIBLE-003	Create Local Thread	RW-RO-RX
CRUCIBLE-004	Create Local Thread	Shared Section Mapping
CRUCIBLE-005	Indirect Function Call	RWX
CRUCIBLE-006	Indirect Function Call	RW-RX
CRUCIBLE-007	Indirect Function Call	RW-RO-RX
CRUCIBLE-008	Indirect Function Call	Shared Section Mapping
CRUCIBLE-009	Queue User APC	RWX
CRUCIBLE-010	Queue User APC	RW-RX
CRUCIBLE-011	Queue User APC	RW-RO-RX
CRUCIBLE-012	Queue User APC	Shared Section Mapping
CRUCIBLE-013	Create Remote Thread	RWX
CRUCIBLE-014	Create Remote Thread	RW-RX
CRUCIBLE-015	Create Remote Thread	RW-RO-RX
CRUCIBLE-016	Create Remote Thread	Shared Section Mapping
CRUCIBLE-017	Set Thread Context	RWX
CRUCIBLE-018	Set Thread Context	RW-RX
CRUCIBLE-019	Set Thread Context	RW-RO-RX
CRUCIBLE-020	Set Thread Context	Shared Section Mapping
CRUCIBLE-021	Remote Callback Insertion	RWX
CRUCIBLE-022	Remote Callback Insertion	RW-RX
CRUCIBLE-023	Remote Callback Insertion	RW-RO-RX
CRUCIBLE-024	Remote Callback Insertion	Shared Section Mapping

Table 2: All combinations of listed allocation and execution primitives were tested, where compatible. All test cases were consistently detected by the system.