Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA

Enhancing DFIR in orchestration Environments: Real-time forensic framework with eBPF for windows

Philgeun Jin [a], Namjun Kim [b], Doowon Jeong [a,*]

[a] Dept. of Forensic Sciences, Sungkyunkwan University, 25-2 Sungkyunkwan-ro, Jongno-gu, Seoul, 03063, South Korea
[b] CapeLabs, 16, Heungan-daero 223beon-gil, Dongan-gu, Anyang-si, Gyeonggi-do, 14106, South Korea

A R T I C L E I N F O

Keywords:
Windows eBPF
Container forensics
Kernel-level monitoring
Orchestration environments
Incident response

A B S T R A C T

Digital forensic investigations in Windows orchestration environments face critical challenges, including the ephemeral nature of containers, dynamic scaling, and limited visibility into low-level system events. Traditional event log-based approaches often fail to capture essential kernel-level artifacts such as process creation, file I/O, and registry modifications. To overcome these limitations, this paper introduces a novel DFIR framework that leverages eBPF to enable real-time kernel-level monitoring in containerized environments. Building on Microsoft's Windows eBPF project, we developed custom eBPF extensions tailored for DFIR. Aligned with NIST SP 800-61 guidelines, the proposed framework integrates unified workflows for preparation, detection, containment, and recovery through a centralized management console. Through case studies of cryptocurrency mining, ransomware, and blue screen of death attacks, we demonstrate our framework's ability to identify malicious processes that traditional event log-based methods might miss, while confirming minimal system overhead and high compatibility with existing orchestration platforms.

1. Introduction

With the increasing market share of cloud-native architectures, organizations have been rapidly adopting various container orchestration solutions. For instance, the CNCF 2023 annual survey (Cloud Native Computing Foundation (CNCF), 2023) reports that 66 % of the responding organizations already use a prominent orchestration platform in production, whereas only 15 % indicated having no plans to adopt such platforms in the future. This finding suggests that overall adoption of these technologies will likely continue to rise.

Meanwhile, the short-lived and dynamically scalable nature of containers poses significant challenges for traditional digital forensics techniques, such as performing offline analysis after disk imaging. In these orchestration environments, containers can be created and destroyed within seconds, leaving only a brief window to capture potential attack artifacts, which in turn increases the complexity of incident response. Consequently, there is a growing call for new methodologies and best practices to address these challenges (Sysdig, 2022).

In response to this need, research in Digital Forensics and Incident Response (DFIR) within orchestration environments has been steadily advancing. In the Linux ecosystem, tools based on eBPF (extended Berkeley Packet Filter), such as Falco and Tracee, are widely employed to monitor kernel-level events in real time, and their adoption for forensic analysis is growing (Sysdig, 2025; Aqua Security, 2025). By contrast, research on Windows orchestration environments remains comparatively underexplored. Current Windows-based logging approaches predominantly focus on application-level log analysis, making it challenging to track kernel-level events such as process creation or file I/O. Recently, Microsoft's Windows eBPF project (Microsoft. ebpf for windows) has introduced the potential for monitoring kernel operations within a secure sandbox environment in Windows. However, unlike Linux, which provides numerous hook points, the project is currently limited to collecting network-related information, significantly restricting its applicability for DFIR.

To address these gaps, this paper develops a novel Windows eBPF extension and proposes a DFIR framework based on this development. The proposed framework enables real-time collection and analysis of kernel events in Windows orchestration environments. Furthermore, the framework integrates these events with centralized log analysis tools, such as OpenSearch, to enhance the effectiveness of incident response. The key contributions of this paper are as follows:

* Corresponding author.
E-mail addresses: philgeun@skku.edu (P. Jin), austin@thecapelabs.com (N. Kim), doowon@skku.edu (D. Jeong).

https://doi.org/10.1016/j.fsidi.2025.301923

- We developed and open-sourced a Windows eBPF extension capable of collecting critical DFIR artifacts, including process modification events, file I/O operations, and registry changes. This open-source project is designed to evolve with contributions from the DFIR research community.
- We implemented an incident response process aligned with the NIST SP 800-61 (Cichonski et al., 2012), systematically identifying the unique requirements for incident response in Windows orchestration environments, which differ significantly from Linux-based environments.
- Through case studies, we demonstrate that our framework can detect kernel-level tampering and low-level attack indicators that traditional file log–centric forensic techniques often overlook. Additionally, We show that the framework enables rapid attack isolation and artifact preservation upon detection.

The remainder of this paper is organized as follows. Section 2 reviews relevant background information, and Section 3 details the implementation of the Windows eBPF extension. Section 4 introduces the overall framework and provides in-depth explanations of its components. In Section 5, we evaluate the performance and feasibility of the proposed approach through case studies covering three types of malicious behavior. Finally, Section 6 concludes the paper and discusses future research directions.

2. Background

2.1. Extended Berkeley Packet Filter (eBPF)

eBPF is a transformative technology that allows user-defined code to execute securely within the Linux kernel, enabling real-time monitoring and enforcement of security policies (eBPF.io. ebpf, 2025; Linux Foundation, 2024). Solutions such as Falco, Tracee, and Datadog utilize eBPF in Linux environments to trace system calls and monitor potential intrusion vectors. These tools can generate real-time alerts and collect forensic data upon detecting suspicious activities, facilitating rapid and context-aware incident response (Sysdig, 2025; Aqua Security, 2025; Fournier et al., 2021).

2.2. eBPF for windows

Microsoft's Windows eBPF project, though still in its infancy, seeks to adapt key concepts from Linux eBPF for the Windows kernel. Currently, its functionality is limited to network-layer event monitoring (Microsoft. Progress on making ebpf, 2021).

Simultaneously, there is an industry-wide shift toward tightening access to the Windows kernel. A notable example is the recent Blue Screen of Death (BSOD) incident caused by a CrowdStrike Falcon Sensor update (Microsoft Support, 2025; Warren, 2019), which underscores Microsoft's increasing emphasis on stricter control over third-party drivers and kernel modifications. As these restrictions grow, integrating external security modules or drivers to directly monitor kernel-level events may become more challenging. Consequently, eBPF-based event collection within secure sandboxes is likely to gain prominence as a viable alternative.

However, the early-stage development of Windows eBPF leaves a significant gap in guidelines or best practices for its use in forensic and security-monitoring contexts. This limitation highlights the challenges in capturing and analyzing kernel-level events within Windows container environments, presenting an open research problem.

2.3. DFIR in orchestration environments

Orchestration environments manage and automate tasks across distributed systems, applications, and services (Red Hat, 2025). This paper focuses on Kubernetes as a representative orchestration platform. As illustrated in Fig. 1, key concepts such as Cluster, Node, Pod, and Container provide essential context for understanding the framework proposed in subsequent sections.

In the era of cloud-native computing, containers and orchestration environments have become foundational components of IT infrastructure. Due to their ephemeral nature and dynamic scaling under varying workloads, traditional forensic approaches such as disk imaging are often infeasible. Once a container is terminated, retrieving events or artifacts generated within it becomes exceedingly difficult.

In Linux-based Kubernetes environments, eBPF is leveraged to collect kernel-level events across nodes in real time (Timothy, 2022; Amin et al., 2023). However, a comparable approach for Windows orchestration environments remains underexplored, emphasizing the urgent need for research in this area.

3. Implementation of windows eBPF extension for DFIR

Leveraging eBPF for DFIR in Windows orchestration environments requires addressing existing limitations and developing practical deployment strategies. While eBPF in Linux is widely adopted for forensics and security purposes, its implementation in Windows is still in its early stages, limiting critical functionalities such as kernel-level event detection and real-time data collection. These constraints hinder the efficient collection and analysis of forensic data from Windows nodes, reducing the overall effectiveness of digital forensics in Windows-based orchestration setups.

To overcome these challenges, we designed and implemented a novel eBPF extension capable of detecting four important types of Windows kernel events, such as process creation, file I/O, registry changes, and network activities, in real time. This extension was developed with a focus on extensibility and efficiency, enabling the addition of new forensic capabilities with minimal modifications as requirements evolve in various environments. The Windows eBPF extension developed in this study, referred to as `eBPF-for-DFIR`, has been released as open-source,[1] allowing professionals and researchers to adopt and enhance it collaboratively.

3.1. Setting up the windows testing environment for `eBPF-for-DFIR` *development*

Developing kernel-level components such as drivers poses risks to system stability, making robust isolation a critical consideration. To mitigate these risks, we configured a Hyper-V virtual machine (VM) as the primary development and testing environment, ensuring that issues like kernel panics (e.g., Blue Screen error) would not affect the host system.

The development environment was established using Microsoft

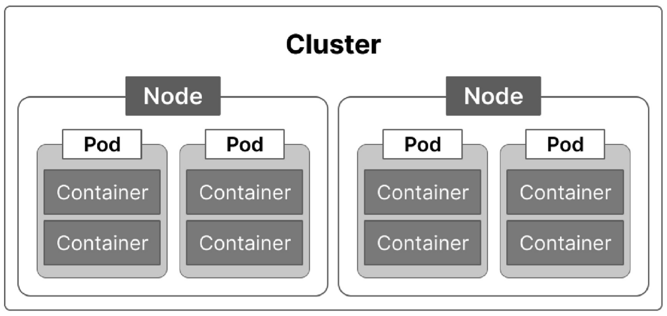


Fig. 1. Key components of kubernetes.

[1] url: https://github.com/capelabs/eBPF-for-DFIR.

Hyper-V to create a Windows 10–based VM, with the host system equipped with an AMD Ryzen 7900 processor and 32 GB of RAM. The VM was allocated 4 cores and 8 GB of RAM, running Windows 10 (version 22H2). Visual Studio 2022 was used as the primary development tool.

The eBPF development environment was configured based on Microsoft's eBPF for Windows GitHub project (Microsoft. ebpf for windows). This project was installed on a pre-configured virtual environment to enable the use of eBPF extension. To customize eBPF hooks, we analyzed Microsoft's `ntosebpfext` (Microsofta). Based on the results of this analysis, we added hook points and modified the system to enable real-time monitoring of Windows kernel events.

3.2. *Development of* `eBPF-for-DFIR`

The development of `eBPF-for-DFIR` tailored for DFIR demands an efficient and extensible design to handle the real-time collection of kernel-level events (e.g., process creation, file I/O, registry modifications). Fig. 2 illustrates an overview of Windows eBPF integrated with `eBPF-for-DFIR`. The section labeled as `eBPF-for-DFIR`, `eBPF Shim` and `Collector` in the figure represents the component specifically developed in this study.

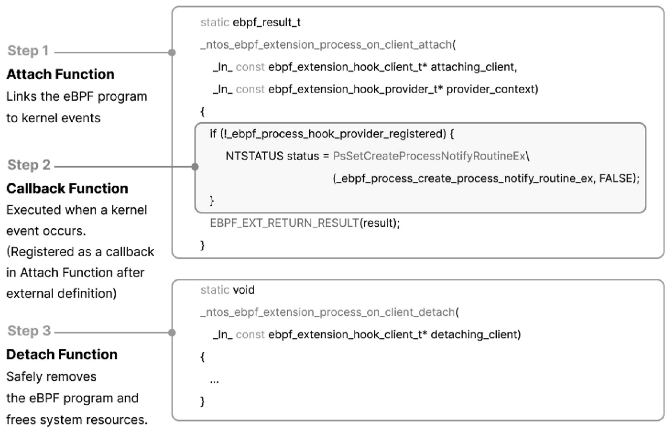
3.2.1. *Design principles*

In this research, the design of `eBPF-for-DFIR` follows three principles:

- Modularization: Each data-collection module operates independently while sharing common structures and interfaces, enabling seamless maintenance and future scalability.
- Efficient Data Management: Forensic-relevant events detected via eBPF hooks are stored in a structured context, ensuring consistency and preventing redundant data storage.
- Performance Optimization: By filtering unnecessary data at the kernel level and transmitting only essential information to the user space, the system enhances efficiency while minimizing CPU and memory overhead.

3.2.2. `eBPF Shim`

During the eBPF extension development process, the Microsoft eBPF for Windows (Microsoft. ebpf for windows) project was utilized to conduct an in-depth analysis of Windows kernel APIs and identify suitable hook points. Based on the existing network sample code (`neteventebpfext` (Microsofta), we implemented and refined the `Attach`, `Detach`, and `Callback` functions, enabling the eBPF program to dynamically attach to or detach from specific events and process incoming data. Fig. 3 highlights the code responsible for performing the three key operations of the `eBPF Shim`.


Step 1
Attach Function
Links the eBPF program to kernel events
Step 2
Callback Function
Executed when a kernel event occurs.
(Registered as a callback in Attach Function after external definition)
Step 3
Detach Function
Safely removes the eBPF program and frees system resources.

```
static ebpf_result_t
_ntos_ebpf_extension_process_on_client_attach(
   _In_ const ebpf_extension_hook_client_t* attaching_client,
   _In_ const ebpf_extension_hook_provider_t* provider_context)
{
   if (!_ebpf_process_hook_provider_registered) {
      NTSTATUS status = PsSetCreateProcessNotifyRoutineEx\
                        (_ebpf_process_create_process_notify_routine_ex, FALSE);
   }
   EBPF_EXT_RETURN_RESULT(result);
}

static void
_ntos_ebpf_extension_process_on_client_detach(
   _In_ const ebpf_extension_hook_client_t* detaching_client)
{
   ...
}
```

Fig. 3. Source Code of the 3 Core Functions of the `eBPF Shim`

- `Attach` Function: Links the eBPF program to specific kernel events or hook points, enabling real-time monitoring of process creation, file access, registry changes, and other kernel activities.
- `Detach` Function: Safely removes the eBPF program when it is no longer required, freeing system resources and preventing unnecessary overhead
- `Callback` Function: Executes when a kernel event occurs, processing the event data, forwarding it to user space, or performing additional tasks (e.g., logging and alerting).

To enable seamless interaction between eBPF programs and kernel events, driver development was essential for providing the necessary interface and real-time event handling capabilities. Driver development was based on the Kernel Mode Driver (KMDF) template in Visual Studio, extending the example code provided in the `ntosebpfext` (Microsofta).

Compiling the driver code produces three primary files: `.sys`, `.inf`, and `.cat`. The `.sys` file contains the key functionality, the `.inf` file provides installation guidelines, and the `.cat` file verifies the integrity of the driver package. These files ensure that the driver can be safely installed and operated in a controlled environment. After deployment in the Hyper-V virtual machine, the driver successfully enables a real-time eBPF-based data-collection pipeline.

3.2.3. `Collector`

To effectively capture and process data, the `eBPF-for-DFIR` was designed to transfer event information from kernel space to user space.

In the kernel space, an `eBPF shim` hooks into kernel data to detect specific events such as process creation/termination, file access, and registry modifications. When an event is triggered, the eBPF program

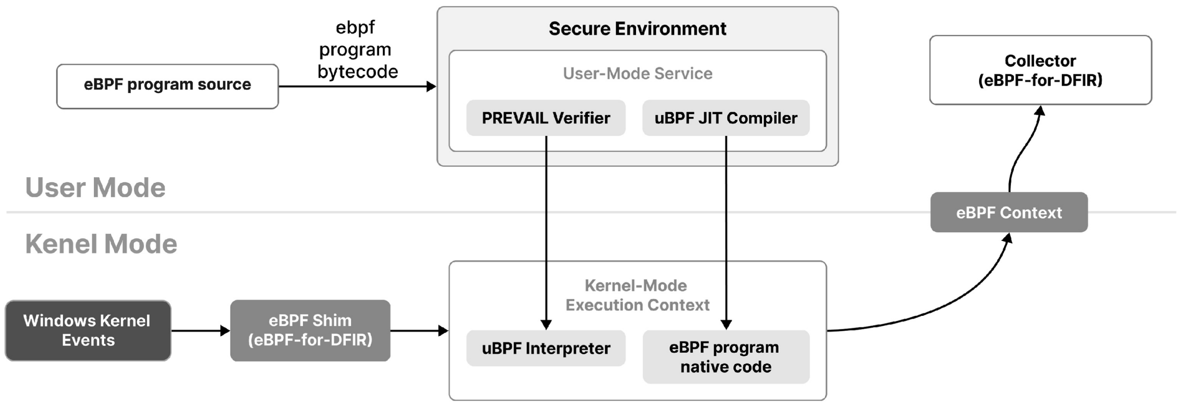


Fig. 2. The Diagram of Windows eBPF with `eBPF-for-DFIR`

employs helper functions to store essential details in a `eBPF Context`, which then organizes and manages this information.

Listing 1 illustrates an example of the `eBPF Context` related to the process.

```
typedef struct _process_notify_context
{
    EBPF_CONTEXT_HEADER;
    process_md_t process_md;
    PEPROCESS process;
    PPS_CREATE_NOTIFY_INFO create_info;
    UNICODE_STRING command_line;
    UNICODE_STRING image_file_name;
}process_notify_context_t;
```

Listing 1: Example of the `eBPF Context` Related to Process

The `collector` interprets the `eBPF Context` received from the kernel space, as described earlier, and converts them into an understandable format for extracting forensic artifacts. Since the structure of `eBPF Context` and the nature of the data they contain vary depending on the event type, it is essential to define data structures through `eBPF Context` analysis and implement specialized parsing modules tailored to each event type. The processed information for each event type is transformed into JSON format and forwarded to external systems, such as log analysis platforms. Section 3.3 provides the detailed explanation of the data specifications for each event type.

3.3. Data specifications

The `eBPF-for-DFIR` collects multiple categories of kernel-level forensic data in a Windows environment, including process, file, network, registry, and command-line information. This subsection outlines the types of data collected and their detailed specifications.

3.3.1. Process creation and termination

The `eBPF-for-DFIR` collects process-related information as shown in Table 1. This data can be obtained by parsing the `eBPF Context`. For example, referring to Listing 1, the `PID` is retrieved from `PEPROCESS`, while the process creation timestamp is obtained from `PPS_CREATE_NOTIFY_INFO`. In addition to the PID and creation time, other critical information is captured for incident investigation, including the executable path, parent-child relationship, and command-line parameters executed via CMD or PowerShell.

The operation types related to processes are categorized into two types: creation and termination. These are identified by the enum values of the operation field, where 0x00 represents creation and 0x01 indicates termination. For each event occurrence, timestamps such as `created_at` and `destroyed_at` can be referenced. In particular, for process termination events, the exit code can also be captured through `exit_code`.

Table 1
Data specification for process creation and termination.

Key	Type	Description
operation	unsigned int32	Types of Process Event Collection (Enum)
pid	unsigned int	Process PID
ppid	unsigned int	Parent process PID (available only at creation)
image_file_name	unicode string	Name of the process file
command_line	unicode string	Command-line arguments passed during process execution
created_at	unsigned int	Process creation timestamp (available only at creation)
destroyed_at	unsigned int	Process termination timestamp (available only at termination)
exit_code	unsigned int32	Exit code returned when the process terminates (available only at termination)

3.3.2. File I/O

In the Windows environment, file operations are handled through IRP_MJ (Interrupt Request Packet Major Function) codes. Based on the official Microsoft documentation (Microsoft Corporation, 2024), we analyzed the key IRP_MJ codes to define and collect crucial file I/O events. Similar to processes, the operation types of file events can be identified through the enum value of the operation key (see Tables 2 and 3).

Whenever the file I/O events listed in Table 3 are triggered, related information such as the file path and timestamps, as shown in Tables 2 and is collected. Notably, the PID of the process that triggered the file event is also recorded, providing critical data for identifying malicious activities, such as malware creating infected files or unauthorized modification of sensitive data.

3.3.3. Network I/O

The network I/O event collection is developed by enhancing the network collection function provided by Microsoft's eBPF for Windows. In the original Microsoft's implementation, network logs are recorded only after a session has terminated, due to the duration-based logging approach. This means that if a malicious process remains active and does not close its network session, no relevant network logs will be generated, making it unsuitable for DFIR. To overcome this limitation, we analyzed the code and removed the duration constraint, enabling continuous monitoring and real-time recording of network activity.

The `eBPF-for-DFIR` collects real-time network artifacts, including IP addresses, port numbers, interfaces, and protocol types (see Table 4). It captures network activity for protocols, identified by their corresponding enum values: 0x01 for ICMP, 0x06 for TCP, and 0x11 for UDP. By integrating the collected data with network anomaly traffic detection system, it can effectively identify malicious communications with external attacker-controlled servers and attempts at data exfiltration.

3.3.4. Registry change

In the Windows environment, registry modifications are frequently exploited by malware to establish persistence. To address this, `eBPF-for-DFIR` collects forensic artifacts whenever a registry change event occurs. Specifically, it captures the registry key, value, data type, and the process ID that triggered the change (see Table 5).

We enumerated the operation types of registry change events based on the `REG_NOTIFY_CLASS` value in the `Ex_callback_function` of the Windows driver (Microsoft, 2023a), as shown in Table 6. The `REG_NOTIFY_CLASS` values provide both `pre-` and `post--` notifications, allowing the system to log registry state information before and after a change occurs.

3.4. Mapping Processes to pods and containers

In an orchestration environment, it is crucial from a DFIR perspective to accurately identify the specific Pod or container involved in a threat incident and take immediate action. In Linux environments, cgroups provide an easy way to identify container IDs (Hoang et al., 2023), but

Table 2
Data specification for file I/O

Key	Type	Description
operation	unsigned int32	Types of File Event Collection(Enum)
file_name	unicode_string	File path
create_time	long long	Create Time
last_access_time	long long	Last Access Time
last_modified_time	long long	Last Modified Time
file_size	long long	File size
pid	unsigned long	PID of the process that triggered the file event

Table 3
Operation types of file event.

Enum	Major Function Code	Description
0x00	IRP_MJ_CREATE	File or directory creation
0x02	IRP_MJ_CLOSE	Close file handle
0x03	IRP_MJ_READ	Read data from file
0x04	IRP_MJ_WRITE	Write data to file
0x06	IRP_MJ_SET_INFORMATION	Set file information
0x09	IRP_MJ_FLUSH_BUFFERS	Flush file buffers
0x0C	IRP_MJ_DIRECTORY_CONTROL	Set directory control
0x0D	IRP_MJ_FILE_SYSTEM_CONTROL	Perform file systemcontrol operations
0x12	IRP_MJ_CLEANUP	File cleanup operations

Table 4
Data specification for network I/O

Key	Type	Description
operation	unsigned int32	Types of Network Event Collection(Enum)
src_ip	ip_address_t	Source IP address
src_port	uint16	Source port number
dst_ip	ip_address_t	Destination IP address
dst_port	uint16	Destination port number
interface_name	string	Interface used during communication

Table 5
Data specification for registry change.

Key	Type	Description
operation	unsigned int32	Types of Registry Event Collection(Enum)
key_path	unicode string	Path of the registry key
value	unicode string	Name of the registry value
type	ulong	Data type
pid	unsigned long	PID of the process that triggered the registry event

Table 6
Operation types of registry change.

Enum	REG_NOTIFY_CLASS value	Description
0x01	RegNtPreSetValueKey	Set registry value
0x02	RegNtPreDeleteValueKey	Delete registry key
0x0F	RegNtPostDeleteKey	Delete registry key
0x11	RegNtPostDeleteValueKey	Delete registry key value
0x10	RegNtPostSetValueKey	Set registry key value
0x1A	RegNtPreCreateKeyEx	Create registry key
0x1B	RegNtPostCreateKeyEx	Create registry key
0x04	RegNtPreRenameKey	Rename registry key
0x13	RegNtPostRenameKey	Rename registry key
0x03	RegNtPreSetInformationKey	Set registry key information
0x12	RegNtPostSetInformationKey	Set registry key information
0x08	RegNtPreQueryValueKey	Query registry key value
0x17	RegNtPostQueryValueKey	Query registry key value
0x09	RegNtPreQueryMultipleValueKey	Query multiple registry key values
0x18	RegNtPostQueryMultipleValueKey	Query multiple registry key values

Windows lacks an equivalent feature, making Pod-level forensic analysis more challenging. To overcome this limitation, we developed a technique that identifies the specific Pod or container where an event occurred by analyzing the process hierarchy of Windows nodes and integrating kernel-level forensic data.

Fig. 4 presents an example of a process hierarchy observed in a Windows node. The process `containerd-shim-runhcs-v1.exe` represents a Pod, while each `cmd.exe` instance beneath it corresponds to a container running within that Pod. Below `cmd.exe`, a `powershell.exe` process indicates that PowerShell is executing inside the container, which then launches `calico-code.exe`. Since `eBPF Shim` and `Collector` capture the PID of a process triggering an event such as a file I/O or a registry change, we can accurately map a process to its corresponding Pod or container. For instance, if an anomalous activity is

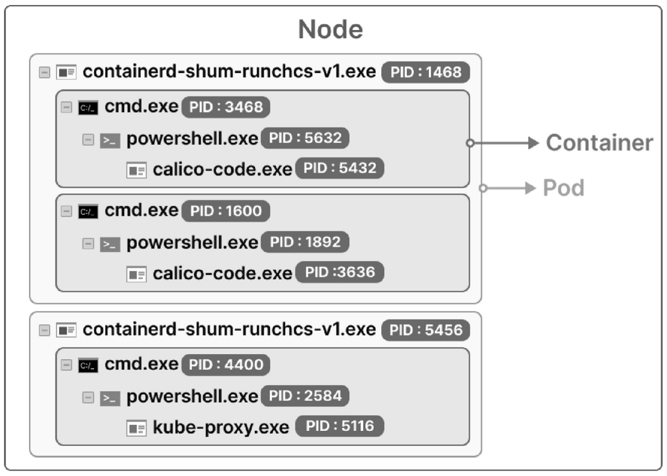

Node
containerd-shum-runchcs-v1.exe PID : 1468
cmd.exe PID : 3468
powershell.exe PID : 5632
calico-code.exe PID : 5432
cmd.exe PID : 1600
powershell.exe PID : 1892
calico-code.exe PID :3636
containerd-shum-runchcs-v1.exe PID : 5456
cmd.exe PID : 4400
powershell.exe PID : 2584
kube-proxy.exe PID : 5116
Container
Pod

Fig. 4. Example of mapping processes to their corresponding pods and containers in a windows node.

detected in `calico-code.exe` (PID 5432), identifying the associated Pod (PID 1468) or container (PID 3468) allows security teams to rapidly localize the incident and take immediate remediation actions, such as isolation or further forensic investigation.

4. eBPF-based DFIR framework

This section outlines a comprehensive framework for conducting DFIR activities in orchestration environments using the custom-developed `eBPF-for-DFIR` extension. As depicted in Fig. 5, the proposed architecture aligns with the four major phases of the NIST SP 800-61 Revision 2 incident response lifecycle: Preparation, Detection & Analysis, Containment, Eradication & Recovery, and Post-Incident Activity. The proposed framework is based on the `eBPF-for-DFIR` extension developed in Section 3, and is designed to operate independently of any specific container orchestration platform. It is applicable not only to Kubernetes-based deployments, but also to general Windows-based container environments, including non-orchestrated hosts.

- **Preparation:** Deploy `eBPF-for-DFIR` and configure a DaemonSet for Windows nodes, ensuring seamless collection and processing of kernel-level events.
- **Detection & Analysis:** Monitor real-time kernel events to identify indicators of compromise, such as suspicious processes, registry changes, and anomalous network activity.
- **Containment, Eradication & Recovery:** Upon detecting malicious activity, isolate affected containers (Pods), restrict network communication, and collect memory and volume artifacts to enable rapid recovery.
- **Post-Incident Activity:** Analyze collected evidence to reconstruct the attack chain and improve detection rules, enhancing future resilience.

4.1. Preparation

The Preparation phase establishes the infrastructure for gathering and analyzing forensic data from Windows orchestration nodes. Two critical components of this phase are the DaemonSet-based deployment of `eBPF-for-DFIR` and centralized log collection.

4.1.1. Deployment of `eBPF-for-DFIR` as DaemonSets

In traditional Windows orchestration environments, DFIR preparations often rely on a sidecar approach, wherein data changes at the

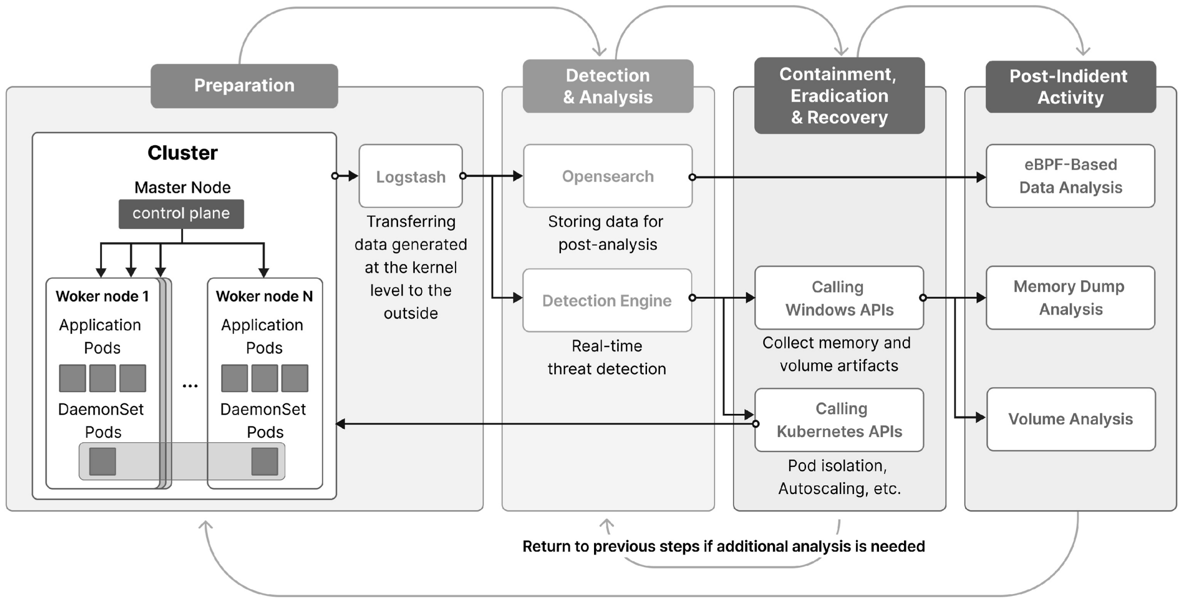


Fig. 5. Proposed eBPF-based DFIR architecture.

application level—such as event logs—are collected by reading log files within each container. This method requires deploying a dedicated collector to every container, as well as synchronizing sidecar configurations or versions based on each container's application version, creating additional overhead.

In contrast, the DaemonSet approach deploys kernel-level data collection programs, such as eBPF, on a per-node basis. More specifically, leveraging eBPF in a DaemonSet-based deployment for node-level log collection offers multiple advantages. First, only one eBPF Pod runs per node, thereby reducing overall resource overhead. Second, since the DaemonSet approach operates at the kernel level, compatibility issues are less pronounced. Given these benefits, deploying `eBPF-for-DFIR` as a DaemonSet provides a stable and scalable infrastructure for real-time log collection.

Furthermore, configuring the system to detect whether a node's operating system is Windows or Linux allows `eBPF-for-DFIR` to be deployed solely where it is needed, thereby accommodating hybrid orchestration environments without issue. Specifically, Linux nodes can continue using their existing eBPF programs, while Windows nodes can receive separate `eBPF-for-DFIR` deployments—maintaining consistency in the overall log collection process. This approach enables efficient security monitoring in clusters that host multiple operating systems.

4.1.2. *Centralized log collection*

Collected events are transmitted from each node to a central repository for analysis. To evaluate transmission efficiency, three protocols—UDP, TCP, and HTTP—were tested for delivering approximately 28,000 logs to a central Logstash server, which subsequently forwarded them to OpenSearch.

The experiments revealed no data loss across all protocols; however, significant differences in transmission time were observed. UDP was the fastest, completing the transfer in 7.5 s, followed by TCP (203 s) and HTTP (487 s). Given the high volume of kernel event logs, UDP proved to be the most suitable for this architecture, balancing speed and reliability.

4.2. *Detection & Analysis*

In this section, the focus is on how kernel events collected through `eBPF-for-DFIR` can be used to detect threats in real time. Rather than delving into the detection logic itself, the discussion centers on how meaningful indicators (e.g., process creation, file I/O) are processed and correlated to identify abnormal behavior. Specifically, once events from each node reach the central detection engine, they are evaluated by predefined rules (e.g., whitelist/blacklist, path monitoring) or an IoC (Indicator of Compromise) matching engine to determine the likelihood of a threat.

For instance, when a new process creation event is detected, if the path or name of the process is not included in a known whitelist (i.e., legitimate applications), it is flagged as "suspicious". At the same time, any registry modification events are examined to check for evidence of persistence or auto-run techniques. This rule-based approach is straightforward yet effective for establishing an initial, rapid detection layer. Subsequently, if a suspicious event surpasses a certain threshold, or if it is combined with network connection events that indicate an excessive number of outbound TCP connections—signaling a potential attack—the central management console automatically calls the orchestration API—such as the Kubernetes API—to quarantine the affected Pod. At that point, additional artifacts associated with the relevant process or file are gathered. Over time, refining the detection engine through post-incident analysis ensures alignment with organizational requirements.

4.3. *Containment, Eradication & Recovery*

Aligned with NIST SP 800-61, this phase involves swiftly neutralizing threats while preserving forensic evidence. Key objectives include isolating compromised Pods, ensuring service continuity across the cluster, and securing critical artifacts such as volume and memory dumps for forensic analysis.

4.3.1. *Pod Isolation*

Pod isolation is a critical step in mitigating attacks while maintaining service continuity and collecting forensic artifacts. Once the detection engine identifies a threat, it utilizes the orchestration API—such as the Kubernetes API—to relabel the compromised Pod and enforce isolation measures, as illustrated in Fig. 6. This automated workflow ensures an efficient response through the following steps:

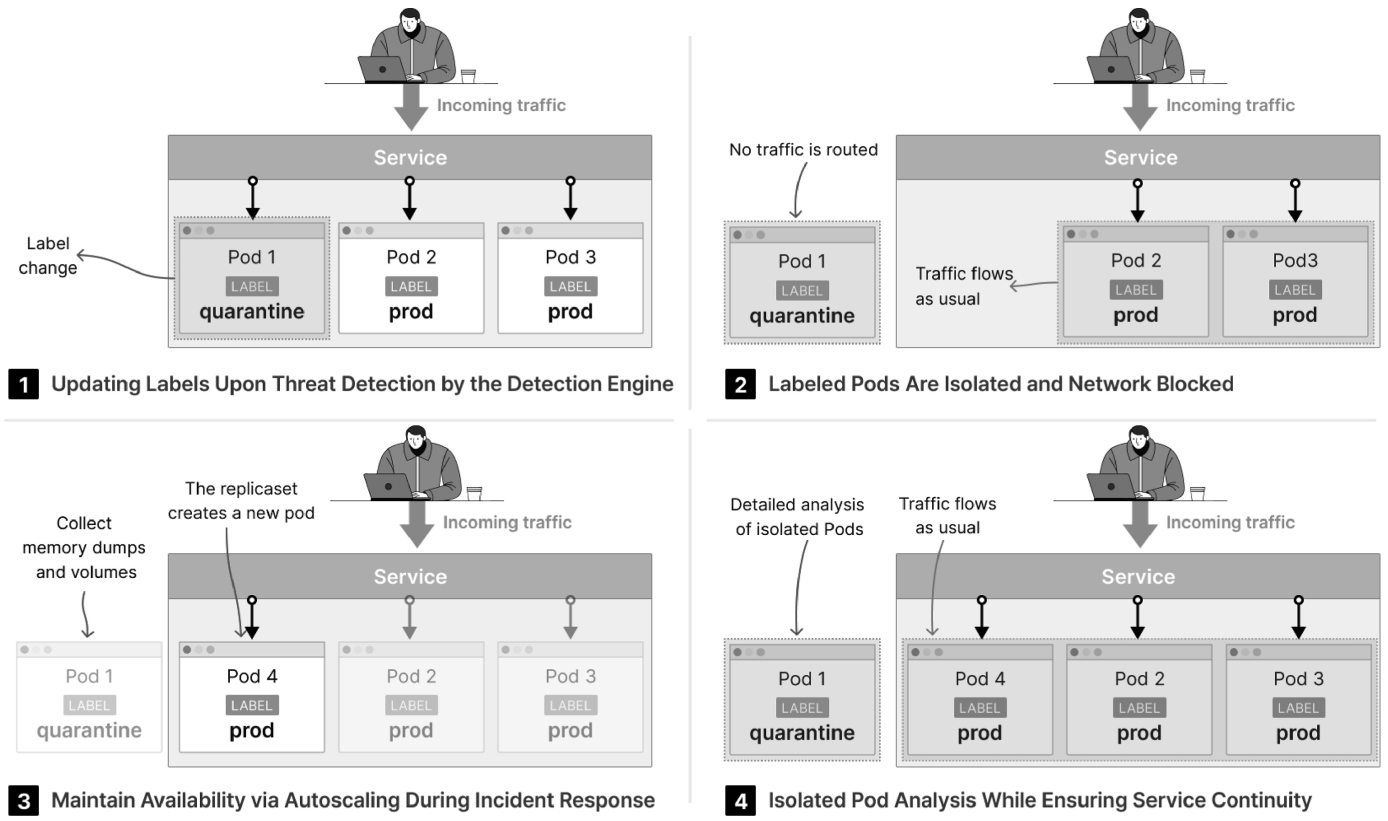

Incoming traffic
Service
Label change
Pod 1 LABEL quarantine
Pod 2 LABEL prod
Pod 3 LABEL prod
1 Updating Labels Upon Threat Detection by the Detection Engine
No traffic is routed
Traffic flows as usual
Pod3 LABEL prod
2 Labeled Pods Are Isolated and Network Blocked
Collect memory dumps and volumes
The replicaset creates a new pod
Pod 4 LABEL prod
3 Maintain Availability via Autoscaling During Incident Response
Detailed analysis of isolated Pods
Traffic flows as usual
4 Isolated Pod Analysis While Ensuring Service Continuity

Fig. 6. Procedures for pod isolation and artifact collection.

1. Updating Pod Labels Upon Threat Detection: The detection engine updates the label of the compromised Pod (e.g., to “quarantine”), distinguishing it from normal Pods labeled as “prod”.
2. Isolating and Blocking Network Traffic: A NetworkPolicy is applied to prevent any external or internal traffic to the quarantined Pod, while unaffected Pods continue operating normally to ensure service continuity.
3. Maintaining Availability via Autoscaling: A new Pod is automatically created by the replicaset to replace the isolated Pod, ensuring traffic flow and availability are not disrupted. Simultaneously, volume and memory dumps of the quarantined Pod are collected for forensic analysis.
4. Analyzing Quarantined Pods: The isolated Pod is analyzed in detail, providing crucial insights for the investigation, while unaffected Pods continue serving incoming traffic.

By isolating the malicious Pod, controlling network access, and collecting vital evidence, the system effectively contains the threat while preserving service continuity in orchestration environments.

4.3.2. Memory dump analysis

To facilitate more in-depth analysis, it is possible to capture a container’s memory dump. As previously discussed, containers can be classified based on PID, making it feasible to generate a process dump for any PID associated with malicious activity. Leveraging Microsoft’s `MINIDUMP_TYPE` (Microsoft Learn, 2022) with the `MiniDumpWithFullMemory` option allows the complete memory state of the process to be dumped. This approach employs the `MiniDumpWriteDump` API to create a comprehensive memory dump file containing the running process’s code, data, stack, heap, and loaded modules.

4.3.3. Volume analysis

Volume analysis is essential for determining the extent of an attack and for safely preserving forensic artifacts.

Step 1: Identify Volume Usage

Establish whether the compromised container relies exclusively on a container-specific volume or also utilizes shared volumes (e.g., `PersistentVolumeClaims` (Kubernetes, 2025)). This classification dictates subsequent analysis procedures.

Step 2: Container-Specific Volume Analysis

For containers that use only container-bound volumes, use the container’s PID to locate the corresponding container ID within the node’s `metadata.db` file (commonly used by containerd). Windows containers typically maintain a “base snapshot” (`.vhdx` file) capturing the initial container state, along with “delta snapshots” generated during events such as garbage collection, image saving, or container shutdown. Although specialized tools for parsing these snapshots are limited, general-purpose utilities like `strings` can provide partial insights.

Step 3: Shared Volume Analysis

Shared volume can be categorized into three types: volume mounted at the Pod level, volume mounted at the node level, and persistent storage volume for Pods (PersistentVolumes). While Pod-specific volumes pose a lower risk of extensive compromise, node-level or PersistentVolumes—accessible by multiple Pods—enable lateral movement. PersistentVolumes, in particular, persist beyond Pod termination, calling for more comprehensive forensic measures. Because shared volumes often lack native snapshot capabilities, direct access to the volume path typically suffices for an initial data examination.

4.4. *Post-Incident Activity*

In the final phase, investigators correlate events stored in a central repository (e.g., OpenSearch) with memory dumps and volume artifacts to reconstruct the attack chain and identify its root causes. While this step aligns closely with traditional incident response methodologies, the orchestration environment requires special attention to quickly assess whether the compromise extended to additional services or resources. Timely containment and strategic improvements to detection rules and defensive measures are essential to minimizing potential reinfection and strengthening future resilience.

5. Case studies

To evaluate the practical applicability and effectiveness of the proposed eBPF-based DFIR framework in a real-world Windows environment, we conducted three case studies involving malicious activities: cryptocurrency mining, ransomware execution, and blue screen of death (BSOD) events. The case studies were conducted on a Windows Server 2022 host equipped with 4 CPU cores and 16 GB of RAM. The Windows container base image used was Windows Server Core LTSC 2019 deployed on an infrastructure composed of `kubelet` and `containerd`. Each case study illustrates a distinct type of malicious behavior, and Section 5.4 provides a detailed analysis of each scenario.

5.1. *Rationale for case selection*

These case studies were designed to cover a broad range of kernel-space event types, including process creation, network connections, and file or registry modifications. Each case was modeled using well-known representative programs, selected for their relevance and impact in real-world attack scenarios.

First, XMRig has been reported to account for approximately 43 % of cryptocurrency mining attacks, making it one of the most prevalent threats in this domain (Check Point Software Technologies). Its profitability has motivated attackers to deploy it through various exploits and vulnerabilities (Labs; Micro; Uptycs).

Second, WannaCry remains one of the most impactful ransomware strains, reportedly affecting more than 200,000 computers in 150 countries. Major organizations such as FedEx, Honda, and the UK National Health Service (NHS) were among the affected (Cloudflare, 2017).

Finally, BSOD represents a critical example of severe system failure in Windows environments. Identifying the root cause is essential to determine whether the failure stems from internal errors or external attacks. Internal causes may include misconfigured infrastructure or bugs in proprietary code, whereas external causes can result from malicious behavior targeting system components. To simulate and analyze BSOD conditions, we employed Microsoft's NotMyFault utility (Microsoft, 2022a).

5.2. *Attack scenario*

Company A, a gaming service provider, operates in an orchestration environment utilizing Windows containers. One day, due to an operational mistake, the access credentials for a specific Windows container were leaked, allowing an attacker to infiltrate the container. Exploiting this vulnerability, the adversary accessed the container and executed three different malicious behaviors via command-line instructions, each carried out separately.

Fortunately, Company A's security team had already deployed the eBPF-based DFIR framework. It promptly detected the execution of the malware process. Using an orchestration environment with multiple Pods, the detection engine efficiently identified the compromised container by correlating its PID with the active container instance.

5.3. *Automated containment workflow*

First, the label of the Pod containing the compromised container was changed from `app=prod` to `app=quarantine`, clearly marking its isolation. Next, network isolation was enforced to block external traffic, effectively preventing further propagation of the attack. Concurrently, the orchestration platform's autoscaling feature launched new Pods as needed, rerouting traffic to maintain service availability.

Within the isolated Pod, a memory dump was performed, and its associated storage volume was switched to a read-only mode to preserve digital evidence for further analysis. These actions ensured both containment and the safe collection of forensic artifacts.

Finally, the security team conducted an in-depth analysis of the collected logs and artifacts, thoroughly examining the processes before and after the incident. Using these findings, they enhanced the organization's future incident response and prevention measures.

5.4. *Data analysis of case studies*

This subsection presents detailed findings from the analysis of collected eBPF data, OpenSearch logs, and additional insights obtained from a memory dump.

5.4.1. *Cryptocurrency miner - XMRig*

The investigation commenced upon detecting the suspicious process `xmrig.exe` (PID 9564), leading to further analysis of its command-line arguments, such as `xmrig.exe –cpu-max-threads-hint=1 –threads=1 –cpu-priority=2`. These parameters are commonly associated with cryptomining activities, often involving a wallet address. If a wallet address can be extracted, blockchain-tracing techniques may facilitate further analysis.

By correlating the execution timeline of `xmrig.exe` with network logs, we identified traffic directed to a known cryptomining host `199.247.27.41` over port 3333, reinforcing the hypothesis that cryptomining activity was occurring. Registry analysis revealed traces indicating that the `RUN` key had been queried or inspected, though no concrete modifications were detected. While this does not confirm the presence of a persistence mechanism, it suggests potential reconnaissance activity or an attempt to verify startup entries.

Further analysis of file creation events allowed us to determine the exact time when the cryptominer and its associated DLL files were introduced into the system. Notably, artifacts such as `xmrig-6.22.2-msvc-win64/config.json` provided insight into the miner's configuration settings, enabling a more detailed understanding of its operational parameters.

In parallel, memory dump analysis provided additional forensic insights into the compromised host. Examination of dumped file system data facilitated the recovery of the original cryptominer executable along with its configuration details. The analysis further revealed the DLLs loaded by `xmrig.exe`, clarifying its cryptomining workflow by exposing specific functions and library calls utilized during execution.

5.4.2. *Ransomware - WannaCry*

Ransomware activity was first identified when encrypted files with the `WNCRYT` extension, indicative of the WannaCry ransomware family, were detected. This served as a key indicator of the encryption phase initiated by the malware.

A suspicious executable file named ed01ebfbc9eb5bbea5 45af4d01bf5f1071661840480439c6e5babe8e080e41aa.exe was traced to the initial process (PID 9496) responsible for triggering the encryption. This filename, appearing to be a hash, was submitted to VirusTotal and was labeled as WannaCry by 68 vendors. This hash-based detection further strengthened the association to the WannaCry ransomware variant.

Further investigation revealed that along with the ransomware-related file `@Please_Read_Me@.txt`, additional artifacts commonly associated with WannaCry were generated, including `.eky`, `.pky`, `taskdl.exe`, and `taskse.exe`. These files are typically generated during ransomware execution.

Registry analysis found evidence that the `MachineGuid` registry key had been accessed, suggesting a likely attempt to collect system-specific identifiers for victim tracking or system fingerprinting. Overall, the forensic evidence confirms that the system was successfully compromised by WannaCry, resulting in the encryption of files and the presence of several characteristic IOCs.

5.4.3. Windows BSOD - NotMyFault

When a Blue Screen of Death (BSOD) occurs in a Windows environment, it triggers a kernel panic that halts all subsequent logging activity. Upon confirming that there were no incoming logs, an investigation was initiated to determine the root cause.

Reviewing the last logged event, analysis revealed that just before the abrupt halt, the `notmyfault.exe /accepteula /crash` command had been executed, and the corresponding process `notmyfault.exe` had started, strongly suggesting that an intentional BSOD call caused the complete suspension of system-level monitoring.

This case highlights the importance of analyzing recent process activity when log interruptions are observed, as such behavior may indicate a critical system failure.

5.5. Discussion

These case studies demonstrate that the proposed eBPF-based DFIR workflow enables rapid incident response in Windows orchestration environments by capturing critical kernel-level events—such as process creation, registry modifications, and network connections—even under high CPU load caused by cryptocurrency mining and ransomware activity. While XMRig typically consumes 100 % of CPU resources, eBPF-based kernel event collection operated seamlessly without performance degradation. Notably, kernel events were successfully captured up to the point of a BSOD, enabling postmortem analysis in the event of abrupt system failures at the node, pod, or container level.

Unlike traditional Windows Event Log-based analysis, which may experience delays or event loss, this approach collects data directly from the kernel, ensuring more reliable and timely threat intelligence. In dynamic containerized environments, where containers are frequently recreated or scaled, maintaining stable observation points at the kernel level is crucial for accurate detection and rapid containment.

At present, Windows eBPF provides fewer hook points compared to its mature Linux counterpart, which may limit the scope of system call monitoring and internal event tracking. However, the case studies indicate that `eBPF-for-DFIR` can effectively detect basic malicious activities. With continued research and community-driven development, the framework has the potential to close the feature gap between Windows and Linux eBPF implementations. As the Windows eBPF ecosystem continues to mature, it is expected to offer enhanced support for large-scale and hybrid orchestration environments through low-overhead, kernel-level instrumentation.

6. Conclusion

This paper presents an eBPF-based DFIR framework, developed to address the limitations of real-time kernel-level data collection and analysis in Windows orchestration environments. The proposed framework is built on `eBPF-for-DFIR`, a Windows eBPF extension we developed, which is deployed via DaemonSets and integrated with centralized log analytics. Aligned with NIST SP 800-61 guidelines, it enables a systematic approach to incident detection, analysis, containment, and recovery. Our case studies show how our eBPF-based monitoring system enables rapid identification of suspicious processes, automatic Pod isolation, and reliable forensic artifact preservation, even under high CPU loads.

Although Windows eBPF remains less mature than its Linux counterpart, this study confirms the feasibility of real-time kernel event collection for effective incident response. As community contributions and Windows eBPF hook expansions continue, forensic capabilities will improve, enhancing system-level visibility. The proposed framework contributes to strengthening security in cloud-native services and containerized environments, ultimately enhancing organizational resilience against fast-evolving threats in Windows-based orchestration environments.

As a priority for future work, we aim to enhance the framework's capability to detect critical threats commonly encountered in Windows environments, such as DLL injection and DLL sideloading. To this end, we plan to develop eBPF-based extensions capable of identifying the exact timing, path, and signature status of dynamically loaded DLLs in real time at the process level. Additionally, we intend to incorporate mechanisms for monitoring mutex object creation and usage patterns, thereby improving the detection coverage of stealthy behaviors frequently associated with advanced persistent threats (APTs).

Acknowledgements

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIP) (No. 2022-0-00281, Development of digital evidence analysis technique using artificial intelligence technology).

Appendix A. Extending `eBPF-for-DFIR`

Extending the functionality of `eBPF-for-DFIR` can be demonstrated by adding support for capturing process creation events. The first step is to verify that the target event provides an appropriate callback interface in the Windows Kernel API (Microsoft, 2023b). One such interface is the `PsSetCreateProcessNotifyRoutineEx` function, declared in `ntddk.h`, which provides interfaces used by the Windows NT kernel. The function declaration is shown in Listing 2.

```
NTSTATUS PsSetCreateProcessNotifyRoutineEx(
  [in] PCREATE_PROCESS_NOTIFY_ROUTINE_EX
    NotifyRoutine,
  [in] BOOLEAN Remove
);
```

Listing 2: Registering a callback for process creation

As shown in Fig. 3, the callback is registered in the `Attach` and must be properly unregistered during `Detach` to ensure appropriate resource cleanup. The registered callback invokes a data transfer routine that forwards event data to the `collector`. Once the callback is enabled, it is automatically triggered upon each process creation.

To properly route captured data to the processing module, the `eBPF context` must include a properly defined `_ebpf_program_section_info` structure (Microsoftb). Specifically, the `bpf_attach_type` and `bpf_program_type` fields must be defined as unique identifiers within the `program_info` header definition. The required values for each field are listed in Table A.7.

Table A7
Unique values for data identification in the `collector`

Type	Unique Value
bpf_attach_type	guid
bpf_program_type	guid, section_name

Appendix B. Performance Evaluation of `eBPF-for-DFIR`

The `eBPF-for-DFIR` is designed for collecting real-time forensic data in orchestrated environments. In such environments, high user concurrency can significantly increase CPU and memory usage, especially when security tools are actively collecting and processing events (Microsoft, 2025).

To evaluate the runtime performance of the `eBPF-for-DFIR`, we conducted a comparative evaluation with an endpoint detection and response (EDR) solution, which is widely used for threat detection and incident response in server environments. We selected OpenEDR (Comodo Security, 2022), a representative open-source EDR solution with a data collection scope most comparable to that of `eBPF-for-DFIR`, as the basis for comparison.

Both tools (`eBPF-for-DFIR` and OpenEDR) were evaluated in the same development environment described in Section 3.1. Performance measurements were collected under three conditions: (1) an idle system, (2) 1,000 concurrent connections simulated using JMeter with Nginx installed, and (3) 10,000 concurrent connections under the same configuration. Each test was conducted three times for approximately 3 min, and the results were averaged.

Performance metrics were collected using Windows Performance Recorder (Microsoft, 2022b), Windows Performance Analyzer (Microsoft, 2020), and Process Explorer (Microsoft, 2024). The primary metrics evaluated were CPU and memory usage. The quantitative comparison is presented in Table B.8. `eBPF-for-DFIR` consistently consumed approximately 1.5 MB of memory across all test conditions, whereas OpenEDR used about 31.4 MB.

Table B8
CPU usage (time) & Memory comparison based on experiments

Tool	CPU (ms)			Memory (MB)
	IDLE	**1,000 Users**	**10,000 Users**	
eBPF-for-DFIR	26.70	36.86	63.78	1.5
OpenEDR	102.27	8,893.71	9,310.93	31.4

Overall, `eBPF-for-DFIR` exhibited significantly lower CPU and memory consumption than OpenEDR across all load scenarios. However, as an early-stage framework, `eBPF-for-DFIR` lacks certain features that OpenEDR supports, such as DLL injection detection, system monitoring, and self-protection mechanisms. If additional extensions are developed to overcome these limitations, `eBPF-for-DFIR` could serve as a practical alternative for resource-constrained environments, given its lightweight architecture and reliable data collection capabilities.

References

Amin, Sadiq, Syed, Hassan Jamil, Ansari, Asad Ahmed, Ibrahim, Ashraf Osman, Alohaly, Manar, Elsadig, Muna, 2023. Detection of denial of service attack in cloud based kubernetes using ebpf. Appl. Sci. 13 (8), 4700.

Aqua Security, 2025. Aqua tracee: runtime ebpf threat detection engine. https://www.aquasec.com/products/tracee/. (Accessed 27 January 2025).

Check Point Software Technologies. Xmrig malware. https://www.checkpoint.com/cyber-hub/threat-prevention/what-is-malware/xmrig-malware/. (Accessed 27 January 2025).

Cichonski, P., Millar, T., Grance, T., Scarfone, K., 2012. In: Nist Sp 800-61 Rev. 2 Computer Security Incident Handling Guide, Revision second ed. National Institute of Standards and Technology, US Department of Commerce, Gaithersburg, MD, pp. 1–79.

Cloud Native Computing Foundation (CNCF), 2023. Cncf annual survey 2023. https://www.cncf.io/reports/cncf-annual-survey-2023/. (Accessed 27 January 2025).

Cloudflare, 2017. What Was the WannaCry Ransomware Attack? (Accessed 11 April 2025).

Comodo Security, 2022. OpenEDR. https://github.com/ComodoSecurity/openedr. (Accessed 11 April 2025).

eBPF.io. ebpf. https://ebpf.io/, 2025–. (Accessed 27 January 2025).

Fournier, Guillaume, Afchain, Sylvain, Baubeau, Sylvain, 2021. Runtime security monitoring with ebpf. In: 17th SSTIC Symposium sur la Sécurité des Technologies de l'Information et de la Communication.

Hoang, Varik, Hung, Ling-Hong, Perez, David, Deng, Huazeng, Schooley, Raymond, Arumilli, Niharika, Yeung, Ka Yee, Lloyd, Wes, 2023. Container profiler: profiling resource utilization of containerized big data pipelines. GigaScience 12, giad069.

Kubernetes, 2025. Persistent volumes. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. (Accessed 27 January 2025).

Labs, Huntress. Threat advisory: xmrig cryptomining by way of teamviewer. https://www.huntress.com/blog/threat-advisory-xmrig-crypto-mining-by-way-of-teamviewer. (Accessed 27 January 2025).

Linux Foundation, 2024. Threat model and independent verifier audit examine the security of ebpf. https://www.linuxfoundation.org/press/threat-model-and-independent-verifier-audit-examine-the-security-of-ebpf. (Accessed 27 January 2025).

Micro, Trend. Examining water sigbin's infection routine leading to an xmrig cryptominer. https://www.trendmicro.com/en_us/research/24/f/water-sigbin-xmrig.html. (Accessed 27 January 2025).

Microsoft, 2020. Windows performance analyzer. https://learn.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer. (Accessed 11 April 2025).

Microsoft, 2022a. NotMyFault. https://learn.microsoft.com/en-us/sysinternals/downloads/notmyfault. (Accessed 11 April 2025).

Microsoft, 2022b. Windows performance recorder. https://learn.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder. (Accessed 11 April 2025).

Microsoft, 2023a. Ex_callback_function (wdm.h) - windows drivers. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nc-wdm-ex_callback_function. (Accessed 27 January 2025).

Microsoft, 2023b. Kernel. https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/_kernel/. (Accessed 11 April 2025).

Microsoft, 2024. Process explorer. In: https://learn.microsoft.com/sr-latn-rs/sysinternals/downloads/process-explorer. (Accessed 11 April 2025).

Microsoft, 2025. Troubleshoot performance issues related to real-time protection. https://learn.microsoft.com/en-us/defender-endpoint/troubleshoot-performance-issues. (Accessed 11 April 2025).

Microsoft Corporation, 2024. !irp. https://learn.microsoft.com/ko-kr/windows-hardware/drivers/debuggercmds/-irp. (Accessed 27 January 2025).

Microsoft. ebpf for windows. https://github.com/microsoft/ebpf-for-windows. (Accessed 27 January 2025).

Microsoft Learn. Minidump_type enumeration. https://learn.microsoft.com/en-us/windows/win32/api/minidumpapiset/ne-minidumpapiset-minidump_type, 2022–. (Accessed 27 January 2025).

Microsoft. Progress on making ebpf work on windows. https://opensource.microsoft.com/blog/2021/11/29/progress-on-making-ebpf-work-on-windows/, 2021–. (Accessed 27 January 2025).

Microsoft Support, 2025. Kb5042421: Crowdstrike issue impacting windows endpoints causing an 0x50 or 0x7e error message on a blue screen. https://support.microsoft.com/en-us/topic/kb5042421-crowdstrike-issue-impacting-windows-endpoints-causing-an-0x50-or-0x7e-error-message-on-a-blue-screen-b1c700e0-7317-4e95-aeee-5d67dd35b92f. (Accessed 27 January 2025).

Microsoft. Ntos ebpf extensions. https://github.com/microsoft/ntosebpfext. (Accessed 27 January 2025).

Microsoft. _ebpf_program_section_info struct reference. https://microsoft.github.io/ebpf-for-windows/struct__ebpf__program__section__info.html. (Accessed 11 April 2025).

Red Hat, 2025. What is orchestration? https://www.redhat.com/en/topics/automation/what-is-orchestration. (Accessed 27 January 2025).

Sysdig, 2022. Practical guide for dfir kubernetes. https://sysdig.com/blog/guide-kubernetes-forensics-dfir/. (Accessed 27 January 2025).

Sysdig, 2025. Falco. https://falco.org/. (Accessed 27 January 2025).

Timothy, D Zavarella, 2022. A Methodology for Using eBPF to Efficiently Monitor Network Behavior in Linux Kubernetes Clusters. Massachusetts Institute of Technology. PhD thesis.

Uptycs. New threat detected: inside our discovery of the log4j campaign and its xmrig malware. https://www.uptycs.com/blog/threat-research-report-team/log4j-campaign-xmrig-malware. (Accessed 27 January 2025).

Warren, Jeff, 2019. Falcon update for windows hosts – technical details. https://www.crowdstrike.com/en-us/blog/falcon-update-for-windows-hosts-technical-details/. (Accessed 27 January 2025).