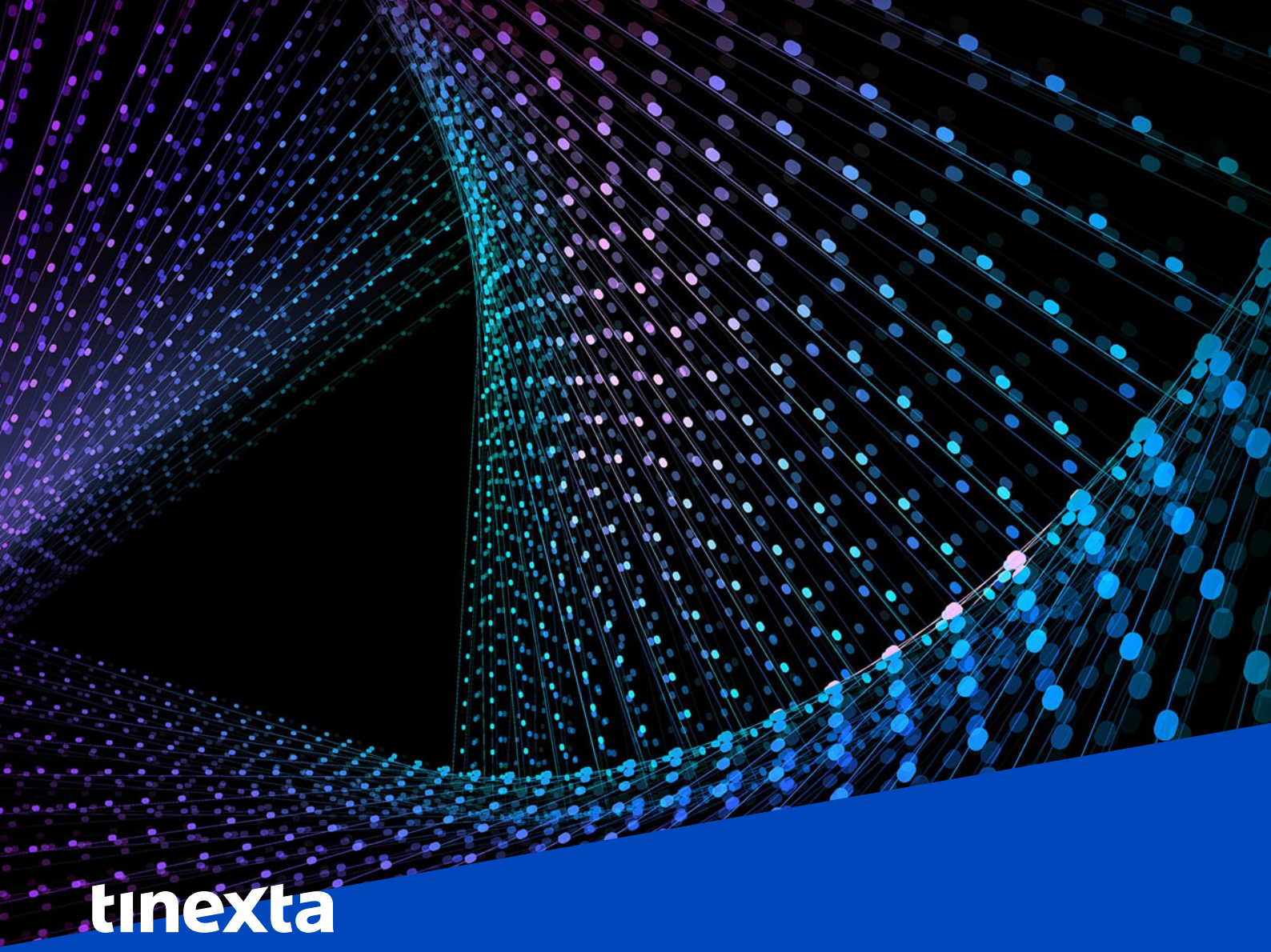


tinexta
defence

# eBPF for Digital Forensics & Incident Response:
## observability and proactive actions at kernel level

#TinextaDefenceBusiness

DFIR

The DFIR Group of Tinexta Defence is a Threat Response Unit specialising in digital forensics and incident response. The Group supports businesses and public administrations in managing security incidents and producing digital evidence with probative value.

The Group's activities integrate multidisciplinary skills and are divided into four main areas:

- **Incident Response:** rapid response capabilities to contain, eradicate and mitigate incidents, reducing operational impact;
- **Court-Appointed and Party-Appointed Technical Expert Services:** IT expertise in line with chain of custody best practices, in support of judicial context;
- **Forensic Readiness:** preventive preparation of processes, technologies, and standards to ensure accurate, complete, and verifiable data collection;
- **Research and Innovation:** experimentation with advanced technologies such as eBPF, kernel telemetry, AI for anomaly detection and container forensics to anticipate threats and develop next-generation tools.

As a Threat Response Unit, the DFIR Group supports Security Operations Centers (SOCs) and organisations in proactive detection, threat hunting and cyber crisis management, as well as post-event investigation.

The Group's mission is to raise the level of security and resilience of critical infrastructures and information systems by combining scientific rigour, technological innovation and operational capabilities to support digital defence and support the judicial process.

# Summary

# Abstract

The growing complexity of cyber attacks requires solutions capable of ensuring deep and timely visibility at the operating system level, minimising performance impact and preserving the forensic value of the data collected.

In this context, *eBPF (extended Berkeley Packet Filter)* technology is establishing itself as an innovative tool for Digital Forensics & Incident Response (DFIR), enabling the secure execution of custom programs directly in the Linux kernel.

This paper explores the potential of eBPF in DFIR, with a focus on two main areas:

- Advanced observability, through the tracking of critical activities such as SSH sessions, executed commands and user changes, ensuring the completeness and integrity of telemetry;
- Proactive defence, through control mechanisms capable of blocking connections to malicious IP addresses, transforming event collection into a tool for immediate reaction.

The proposed solution integrates aspects of forensic readiness (temporal accuracy, integrity and RFC 5424 log format) with operational security features, laying the foundations for a faster, more reliable DFIR approach that can be integrated with SOC and SIEM tools.

The results highlight how eBPF can be a strategic technology for the next generation of investigation and response tools, opening research and development scenarios related to container security, multi-event correlation and Machine Learning applications for kernel-level anomaly detection.

## Authors:

- Marco Tinti: Team Leader DFIR
- Riccardo Luzi: Forensics Analyst
- Andrea Mura: Malware Analyst

# Introduction

In DFIR (*Digital Forensics and Incident Response*), having sufficient high-quality data available is crucial for conducting effective analysis.

The greater the visibility of operations within a system, the greater the ability to carry out accurate and comprehensive investigations. Similarly, the timeliness of information gathering has a decisive impact on the outcome of activities.

In Linux-based systems, numerous approaches and technologies exist for monitoring activities in a timely and thorough manner. These often involve programs running at the operating system level that exploit the kernel's privileged position to observe and control the entire environment.

One such solution is *eBPF* (*extended Berkeley Packet Filter*)[1], which is a particularly powerful tool for improving the observability of a Linux system. This technology enables custom code to be executed in response to specific kernel or user space events without the need to modify or recompile the kernel itself. The code is first compiled into *bytecode*, then executed within a virtual machine integrated into the kernel. A *Verifier* then ensures the code's security and stability.

Unlike traditional kernel modules:

- it does not access kernel memory directly, which reduces the risk of crashes and vulnerabilities;
- it can be loaded and removed dynamically without the need to restart the system;
- it is *event-driven* and activated by predefined or custom hook points (e.g. *syscalls*, network events or *tracepoints*).

Thanks to these features, eBPF offers advanced observability, minimal performance impact and proactive capabilities, such as blocking connections or modifying system behaviour in response to events.

[1] https://ebpf.io/

The proposed system demonstrates a potential application of eBPF technology in this field by collecting information about activities performed via SSH connections to a Linux server. It also demonstrates how this technology can be used for proactive system protection, proposing a solution to block network connections directed to a specific IP address in a user-configured list.

# Observability on Linux systems

Observability is now a fundamental capability both for assessing the performance and status of a system, and for enabling effective security solutions.

It refers to the ability to understand the internal state of a complex system based on its external outputs. Enhancing this capability provides a more comprehensive view of ongoing activities, enabling the identification of threats in real time while giving a robust foundation for post-incident analysis. Thanks to its privileged position that allows it to monitor and control the entire environment, the operating system is the ideal place to implement observability tools.

In Linux, there are various methods of extending the kernel's capabilities according to operational requirements.

One approach is to modify the kernel source code directly. While this solution enables intervention in any part of the kernel to adapt it to specific needs, it requires an extremely in-depth knowledge of the codebase.

It also entails limitations in terms of portability between different Linux distributions or versions, and since each change must be approved by the development community, there can be waiting times of months or even years.

Alternatively, kernel modules can be used. These are object files containing code that extend the kernel's functionality at *runtime* and can be loaded and removed dynamically, on-demand. Many hardware device drivers, for example, are implemented as kernel modules.

Unlike direct kernel modification, using modules allows you to extend the kernel's functionality immediately without waiting for community approval, and there are fewer portability issues.

However, if a module's code is not carefully developed and reviewed with respect to the kernel version in which it is to operate, it can introduce vulnerabilities or cause system crashes. This is where eBPF comes in: a technology that enables custom code to be written, loaded, and removed dynamically to expand the capabilities of the kernel.

Although eBPF and modules share the ability to be executed at runtime, there is one fundamental difference: programs are executed within the eBPF *Virtual Machine*, which is an isolated environment operating on dedicated virtual registers.

This approach drastically reduces the risk of introducing vulnerabilities or instability, since eBPF code does not interact directly with kernel memory.

Furthermore, the Verifier reduces the risk of kernel crashes during execution. This component analyses the program before loading, evaluating all possible execution paths and examining the instructions in logical order. The main checks performed include:

- **Memory access control:** it ensures that BPF programs only access the memory they are authorised to access and do not exceed specific limits. For example, when accessing an array, it is necessary to ensure that the index does not exceed the array's limits;
- **Checking pointers before dereferencing:** one way to cause a program to crash is to dereference a pointer with a value of 0 (also known as *NULL*). To prevent this, the Verifer requires an explicit check before actually accessing the pointed value;
- **Run to completion:** it ensures that the eBPF program runs to completion and does not consume resources indefinitely by placing a limit on the total number of instructions it will process;
- **Invalid instructions:** it verifies that all instructions in a program are valid bytecode instructions (e.g. recognised *opcodes*);
- **Unreachable instructions:** it rejects programs that contain unreachable instructions.

# Main features of eBPF

eBPF programs are known as “event-driven”: once they have been dynamically loaded and associated with a specific event, they are only ever executed in response to that event, without the system needing to be restarted.

In practice, execution occurs when the kernel or an application reaches a predefined or customised hook point, depending on the use case.

## Hook points

The predefined hook points include system calls, function entry and exit points (*fentry/fexit*), network events and kernel tracepoints, among others.

If no predefined hook points satisfy a particular use case, eBPF programs can be anchored almost anywhere in the kernel or user space applications using kernel probes (*kprobe*) or user probes (*uprobe*), respectively.

Let's take a closer look at the **tracepoints**, **raw tracepoints** and **lsm** used in the demonstration in this article.

**Tracepoints** are marked points in the kernel code that remain stable between different versions of the kernel, although an older version may not have the complete set of tracepoints added in a newer version. The format of each tracepoint describes the fields that it traces and therefore the information that can be retrieved from it. This information can then be used to create a data structure associated with a set of raw arguments when using a tracepoint-type program.

For better performance, **raw tracepoints** can be used, which are hook points that allow access to raw arguments without the need to create a data structure or perform the association.

In addition to collecting information from kernel functions, eBPF programs can influence the kernel's behaviour in response to specific events to which the program is attached.

This is possible thanks to the **LSM interface**, which provides a series of hooks that can be activated just before the kernel acts on a data structure or critical path related to system security. The function called by one of these hooks can then decide whether to allow the action to be performed. For example, it can prevent access to a specific file, change the permissions of a folder or contact an IP address. It is important to note that BPF LSM is supported from kernel version 5.7 onwards, and that specific kernel capabilities must be enabled to use this type of hook.

## Life cycle

Once the relevant hooks have been identified, an eBPF program can be created using an existing toolchain that supports languages such as C and Rust. This program can then be converted into bytecode, which is the format expected by the kernel.

This program is then loaded into the kernel using the **bpf** system call. Typically, this is done using one of the many available eBPF libraries that allow the user space part of the system to be developed in languages such as C, Rust, Python and Go.

Once loaded, the eBPF program undergoes the checks performed by the verifier described in the previous chapter. If these checks are passed, the **Just-in-Time Compiler** converts the program's generic bytecode into the machine-specific instruction set. This makes the program as efficient as natively compiled kernel code or code loaded as a kernel module.

## Maps and data structures

The concept of **eBPF Maps** is used to make the collected data usable by both other eBPF programs and applications in user space. These data structures are used to store and share data in the eBPF ecosystem and include *hash tables, arrays, ring buffers* and *LRU (Least Recently Used)* structures, among others.

To interact with these maps and specific kernel data structures, eBPF programs must use dedicated **helper functions**. These helper functions provide a stable API (Application Programming Interface) that avoids compatibility issues between programs and different kernel versions.

## Security

Due to the capabilities and potential of eBPF programs, security is of paramount importance.

For this reason, several security measures are in place:

- Generally, an eBPF program can only be loaded into the kernel if the process loading it is running in privileged mode (*root*), or if the *CAP_BPF* capability is enabled. Alternatively, **unprivileged eBPF** can be enabled to allow non-privileged processes to load eBPF programs. These programs are still subject to restrictions in terms of functionality and kernel access;
- Every eBPF program loaded must pass the checks performed by the Verifier;
- Once these checks have been passed, the program undergoes a hardening process that includes setting the memory used by the program to read-only mode and obscuring the constants in the code;
- While running, an eBPF program cannot access arbitrary areas of the kernel's memory directly. Access to data and structures outside the program's context is only permitted via helper functions, ensuring consistent and secure access and changes.

# Tracking activities performed via SSH

A system for tracking SSH connections on a Linux server is presented as a practical example of a potential application of eBPF programs in the DFIR field.

The aim is to monitor activities performed via SSH connections from initial connection to disconnection, paying attention to the commands executed by connected users.

Some information is also correlated to enable continued tracking of activities with the original addresses and users, even in the event of a user change made with dedicated commands or via an SSH connection to *localhost*.

Finally, connections to specific IP addresses are blocked if they are present in a user-configured blacklist (e.g. a list of IP addresses known to be linked to malicious C2 servers).

## Tracking SSH sessions

When an SSH client connects to a server, an **sshd** process invokes the **getpeername** system call to retrieve information about the address of the peer connecting to a specific *socket*.

Using a tracepoint on this system call makes it possible to retrieve the IP address and port of the connected client.

After a series of operations, a clone of the **sshd** process creates a shell by invoking the **execve** system call to execute **/bin/bash**. From this point onwards, every command executed using this shell will have the PID (Process ID) of the process that created the shell as its PPID (Parent PID).

The **sched_process_exec** tracepoint is activated every time a program is executed and can be used to collect information about the executed command, the path of the executed file, and any arguments passed to it.

Furthermore, eBPF helper functions can be used to obtain information about the **PID, Parent PID** and **UID** of the process that activated the eBPF program.

## Changing user via command

When a user switch is performed using the **su** command, several processes are created in succession before the new login shell is created. This causes the reference to the PID of the process that created the initial shell to be lost.

This results in a loss of correlation between the new commands executed and the SSH session information relating to the initial source IP address, source port and user.

To address this issue, the **sched_process_fork** tracepoint is employed to monitor the relationships between the parent process and each new process as they are created, enabling any new shells to be linked to the original. The fork event chain is tracked using an LRU map to limit memory consumption and avoid overflow in high-load environments.

Consequently, even when a user switches via the **su** command, it is possible to trace back to the initial user, together with the IP address and port of the connected client.

## Changing user via local SSH connection

In the event of a user change initiated by starting a new SSH session to *localhost* (e.g. *user2@localhost*), tracing the creation of processes is insufficient for tracing the original session's information.

A new **sshd** process is created to manage the new session, and this process is not in the child chain of the process that created the initial shell.

To trace the desired data, a tracepoint is used on the **getsockname** system call. When a user starts a new session locally, an **ssh** process uses this system call to retrieve information about its IP address and port.

Conversely, the new **sshd** process uses **getpeername** to retrieve the IP address and port information used by the connected client.

This information can be used to obtain the PID of the **ssh** process and trace the process chain to find the original session information.

Therefore, even if the user switches to a new SSH connection, it is possible to trace the original user, along with the IP address and port of the connected client.

## Blocking IP addresses

Unlike other systems, the proposed system does not passively collect information of interest; it also allows users to proactively block connection attempts to specific IP addresses.

A dedicated map is used to pass a list of configured IP addresses from the application in user space to the eBPF program.

The eBPF program then uses a hook provided by the LSM interface, which is activated when processing an invocation of the **connect** system call.

The **connect** arguments can be used to trace the IP address to which the calling process is trying to connect. If this IP address is present in the configured list, the eBPF program alters the return value of the **connect** system call, thus blocking its execution. It also instructs the application in the user space to generate an alert to indicate possible suspicious activity.

The following code snippet illustrates a basic example: the **socket_connect** type LSM hook uses the **-EPERM** return code to block the connection to a specific destination IP address configured by the user and, at the same time, track this action with a kernel-level debug message via the **bpf_printk** function.

```
SEC("lsm/socket_connect")
int BPF_PROG(block_connection, struct socket *sock, struct
sockaddr *addr, int addrlen) {
    struct sockaddr_in *addr_in = (struct sockaddr_in *)addr;
    __u32 dst_ip = addr_in->sin_addr.s_addr;
    if (dst_ip == <IP_TO_BLOCK>) {
        bpf_printk("Blocking connection to %d\n", dst_ip);
        return -EPERM;
    }
    return 0;
}
```

## Logging and forensic validity of data

The telemetry produced by the system is designed to have probative value and Security Operations Center (SOC) operability. This considers three key principles: temporal accuracy, integrity, and verifiable provenance.

For each command executed via SSH, an event is recorded containing:

- **Dual timestamp:** monotonic time derived from **bpf_ktime_get_ns()** to ensure the sequential ordering of events, as well as wall-clock time in UTC format for external correlations;
- **Boot ID:** a unique system identifier used to distinguish reboots. This is retrieved by reading */proc/sys/kernel/random/boot_id* from user space and passed to the kernel via an eBPF map;
- **Sequence per CPU:** an incremental counter that sorts events by core, which is essential in multicore environments;
- **Missed event counters:** information on any data loss due to buffer overflows, retrieved via **bpf_ringbuf_query()** to ensure transparency regarding log completeness;
- **Original username and UID:** the username and UID of the **sshd** process that created the first shell;
- **Current username and UID:** the username and UID of the process that executed the current command;
- **Source address:** the IP address and port of the client that initially connected;
- **Parent PID and PID:** identifiers of the process and its parent;
- **Executed command:** the complete command, including the path to the binary file;
- **Arguments:** parameters passed to the command.

Logs are generated for connection attempts to IP addresses on the configured blacklist, containing the same set of information as above, but with an additional flag to highlight suspicious activity.

This information is saved in a dedicated log file in RFC 5424 syslog format, which supports structured data and enables integration with SIEM systems such as ELK or Splunk. This format enables advanced queries on fields such as the original UID or IP address, facilitating forensic analysis and real-time correlations.

# Conclusions and future developments

eBPF technology has proven to be highly effective and promising in the fields of system defence and incident investigation. The great freedom it gives users in developing eBPF programs means it can be used in numerous vital cases for protecting Linux systems.

However, it is important to bear in mind that some useful features of this technology are only available in more recent versions of the Linux kernel. Furthermore, kernel capabilities must be enabled to use the technology, and these capabilities may not be compatible with the security policies of some organisations.

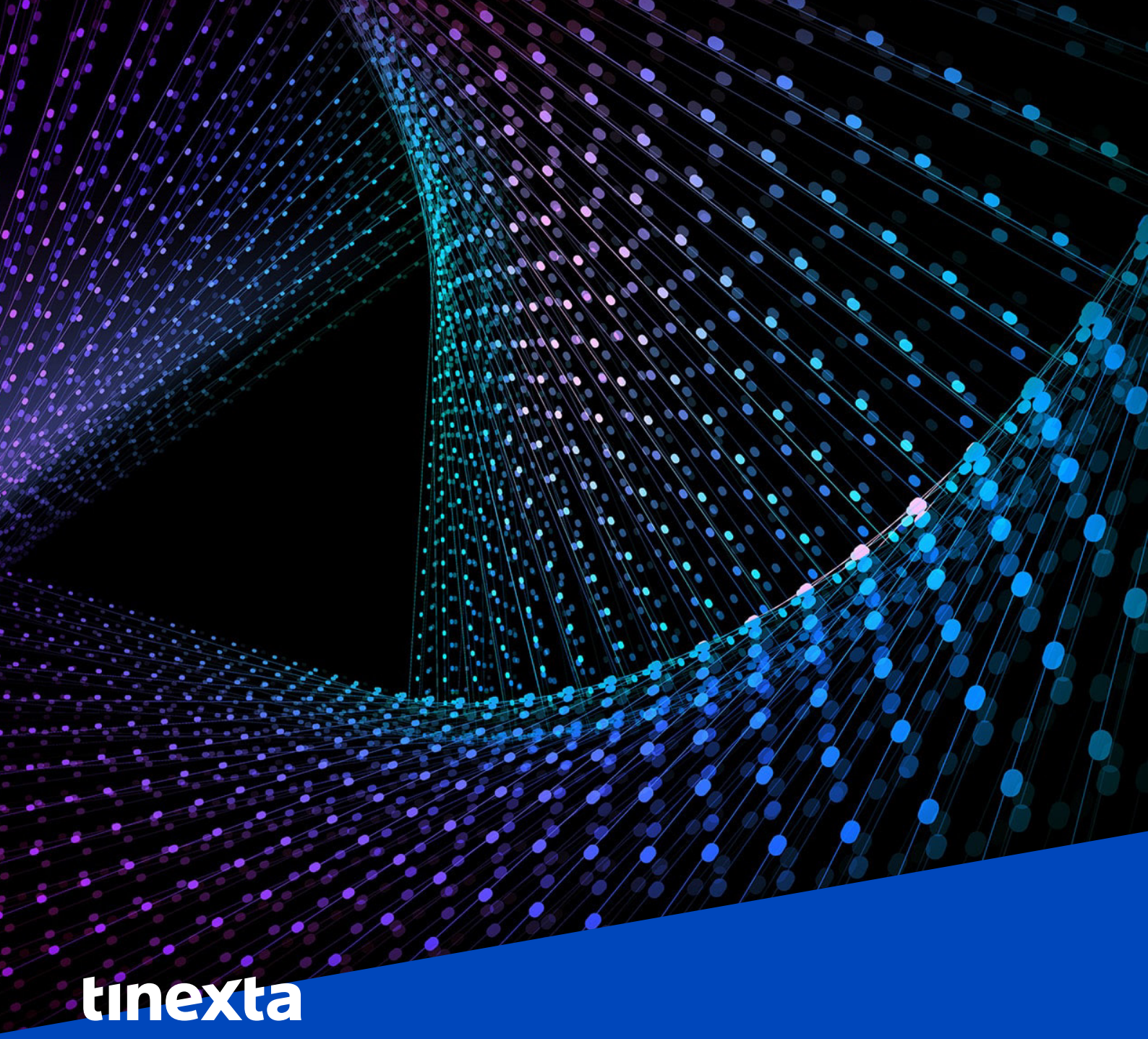
Some future developments of the system described are:

- **Proactive actions:** taking further proactive actions towards suspicious processes, such as connection blocking;
- **Container visibility:** given the growing and widespread use of containers and microservices, information will be added about the containers within which commands are executed;
- **Correlation of other events:** collecting information about other types of events related to commands executed by connected users;
- **Alert rules:** adding a system of rules to generate alerts in certain situations.

Integrating Machine Learning techniques for anomaly detection based on eBPF metrics is a promising area of future research. Open-source projects, such as Falco (CNCF) , and recent academic research suggest that applying machine learning (ML) to kernel-level telemetry can significantly improve the signal-to-noise ratio. Some studies report reductions in false positives of 30–40% in specific contexts.

However, it is important to note that the system has limitations on kernels prior to version 5.10, where some features, such as raw tracepoints and ring buffers, may not be available. This requires alternative implementations with greater overhead.

[2] https://falco.org/



tinexta
defence

**Defence Tech | Next |**
**Donexit | Foramil | Innodesi**

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.tinextadefence.it

#TinextaDefenceBusiness