Stateful Handling of Critical Events: Leveraging eBPF to Realize eXtended Finite State Machine Abstractions

Arianna Quinci[1,2,*], Giacomo Belocchi[1], Francesco Quaglia[1,2] and Giuseppe Bianchi[1,2]

[1] *University of Rome Tor Vergata, Rome, Italy*
[2] *Consorzio Nazionale Interuniversitario per le Telecomunicazioni*

Abstract

As modern security threats evolve, purely stateless defenses cannot capture the progressive nature of malicious behaviors. To address this limitation, we propose to track potentially malicious actions via stateful abstractions based on the eXtended Finite-State Machine (XFSM) paradigm, enabling not only the modeling of process evolution but also the management of system components—including physical ones like RAM frames—within the operating system kernel. By leveraging extended Berkeley Packet Filter (eBPF) technology, our approach ensures minimal performance overhead while facilitating flexible, real-time detection of complex attack patterns. We demonstrate its effectiveness through functional evaluations on real-world scenarios and confirm its practical feasibility via comprehensive performance assessments. Our solution delivers a powerful yet user-friendly defense mechanism, balancing kernel-level complexity with adaptability to contemporary security challenges.

1. Introduction

In recent years, security threats have grown increasingly complex [1], demanding defenses that, unlike stateless rules or policies, track the evolving behavior of processes and system components. More generally, it would be extremely useful to rely on security solutions where specific objects—like an I/O session, or a frame of RAM—could be associated with a state machine oriented to security management. At the same time, supporting the programmability of such state machine through a simplified in-kernel support would be a significant asset. Existing solutions often rely on system call filtering, but many are not fully stateful [2, 3], since they apply filters just based on used parameters or sequences of invoked system calls—hence the state information might be limited to keeping the history of invoked services. Hence, they do not really track the state of generic system-level objects—used at any point in the vertical hardware/software layering—which can be ultimately exploited while really executing (malicious) activities in the system.

In this context, a prominent example is related to an individual RAM frame. In fact, it could represent the backed RAM store underlying shared memory services when mapping logical pages that belong to different address spaces, whose processes could even be active in non-overlapping time intervals. In this scenario, the content of the RAM frame can actually represents a Write-Execute (WX) RAM area where malware could be un-encrypted (by one process) and then executed (by the other process) even if none of the two processes might have had both Write and Execute permissions on the logical page that was mapped on the frame. In this work we propose a system architecture for associating a state to any generic logical/physical object in the IT system, which evolves along time. The final advantage of this solution is the one of enabling the discovery of the entrance of objects—like the aforementioned frame—into states that can be critical for security. To model the evolution of the objects in the operating system architecture, we relied on the "eXtended Finite State Machine" paradigm [4, 5, 6, 7]. XFSMs are finite-state machines enhanced with state variables and condition-based transitions. While traditional finite-state machines rely solely on discrete states, XFSMs incorporate state variables that can hold multiple values. This approach reduces the need to create distinct states for every possible combination of conditions, effectively addressing the "state space explosion" problem [8, 9]. Transitions in an XFSM occur when specified conditions involving state variables are met, allowing for more dynamic and

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
*Corresponding author.

flexible behavior. Moreover, a single transition can simultaneously modify internal state variables and execute actions, streamlining the state management process.
Basing on the XFSM paradigm, in this work we make the following key contributions:

1. Stateful Abstraction Based on XFSM: We propose a stateful abstraction model based on the XFSM paradigm to effectively track and represent the evolution of system objects, in particular within the operating system kernel.
2. Implementation Using eBPF: We develop an efficient XFSM engine for Linux leveraging extended Berkeley Packet Filter (eBPF) technology [10, 11], enabling the execution of multiple XFSMs directly within the kernel with minimal performance overhead.
3. Functional Evaluation with Attack Use Cases: We conduct a functional evaluation using real-world attack scenarios to demonstrate the effectiveness of our approach in detecting complex attack patterns.
4. Performance Evaluation (Overhead Assessment): We perform a comprehensive performance evaluation to assess the overhead introduced by our solution, showing that it remains practical for deployment in production environments.

Our solution strikes a balance between the flexibility of kernel modules—which however are complex to write and manage—and the ease of use of security tools like Tetragon [3] and Falco [2]—which however offer less expressive policies. By leveraging XFSMs and eBPF, we provide a powerful yet efficient mechanism in Linux for stateful monitoring and defense against sophisticated attacks.

The remainder of this article is structured as follows. In Section 2 we discuss related work. In Section 3 we detail the design of our XFSM engine and discuss its implementation using eBPF. In Section 4 we show the system effectiveness via case studies. In Section 5 we evaluate its run-time cost. Conclusions are discussed in Section 6.

2. Related Work

Traditional operating system security has been based on Discretionary Access Control (DAC). To enhance these controls and address their limitations, Mandatory Access Control (MAC) mechanisms were introduced. In Linux, MAC is implemented through the Linux Security Modules (LSMs) framework [12], which provides security hooks into the kernel. Among the various implementations of LSMs, Security-Enhanced Linux (SELinux) [13] is widely adopted. It allows for the definition of rules to prevent critical security scenarios by making access control decisions based on labels. Each resource managed by SELinux has an associated label, and system actors referred to as "subjects", also have associated labels. Through SELinux transitions, it is possible to define the state evolution of system objects. However, a significant limitation of SELinux is its inability to track the state of generic system objects due to the constraints of the LSM hooks. For instance, SELinux cannot be relied upon if one needs to associate a state (and a state trajectory) to a physical memory frame, just because of the absence of apposite hooks [14]. Incorporating monitoring at the physical memory level would require adding custom hooks to the kernel.

Several research proposals have focused on protecting the execution context of system calls through system call filtering mechanisms. Bastion [15] is a prototype that aims to protect against the abuse of system calls by verifying whether the calling context of a system call is permissible. It achieves this by checking system call arguments and sequences. Another approach involves extending SecComp (Secure Computing Mode) with eBPF [16] to introduce a notion of state into the traditional SecComp filtering mechanism. SecComp [17, 18] is a Linux kernel feature that allows processes to be sandboxed according to predefined filters. The eBPF extension enables stateful filtering by incorporating state into the decision-making process. However, this tool is specifically designed to handle a limited set of scenarios—related to tracking the state of system calls—and has no support for monitoring and recording the state of generic system-level objects, including physical objects. To protect containerized

Solution	Stateful monitoring	Layering coverage	Custom policy support	Runtime reconfiguration
SELinux	Yes	Partial	Yes	Partial
Bastion	No	Partial	Partial	Yes
SecComp + eBPF	Yes	Partial	Partial	Yes
Phoenix	Yes	Partial	Yes	Partial
Tetragon	No	Partial	Yes	Yes
Falco	No	Partial	Yes	Yes
XFSM + eBPF (our approach)	Yes	Yes	Yes	Yes

Table 1
Comparison of security solutions and their features.

environments, Phoenix [**?**] is a research tool designed to safeguard against unpatched vulnerabilities. It relies on SecComp and Ptrace [19] to filter system call arguments and sequences, using SecComp for pre-filtering and Ptrace for deep inspection of system call parameters. Like the SecComp-eBPF extension, this mechanism has limitations in monitoring and recording generic types of logical and physical objects. There are also widely used solutions based on eBPF for containerized environments. Tetragon [3] is a runtime security enforcement and observability tool that performs filtering directly within the kernel. It provides a series of filtering policies and allows users to define their own without hard-coded filters. Users specify policies by defining the hook point, the condition to check, and the action to perform when the condition is met.
Falco [2] is another tool used in containerized environments that detects and notifies about suspicious behaviors without taking reactive measures. While Tetragon utilizes eBPF kprobes and tracepoints, Falco employs eBPF kprobes and kernel modules. Like Tetragon, Falco allows users to define custom rules in addition to the default ones provided by the engine. Both Tetragon and Falco parse YAML files to extract the policies to be executed, making them flexible and easy to configure. However, a limitation of these tools is that they do not allow filtering based on conditions applied to the state of an object; they are essentially stateless.

Overall in Table 1 we report a schematized comparison of the discussed solutions, and of what we propose, by relying on a few relevant indexes.

3. The XFSM + eBPF Approach

Our system is implemented as an XFSM executor which is based on two layered architectural components. The topmost one offers a mapping component that allows specifying the state transitions that a state machine needs to implement in order to model the state trajectory of a given object. We call this component as Management-Layer (ML). Currently, our maps kept by the management layer are statically defined by the user, although nothing prevents the possibility to modify the design for offering maps that are in their turn state machines. The lower layer in our architecture is the Per-Object XSFM Layer (POXL). This layer is in charge of maintaining for each object of interest its current state. The different system objects are of interest depending on specific events that occur within the system, which are anyhow linked to specific actions executed by some running thread, either in process context or in interrupt context. In our architecture, these events are generated via an hooking mechanism, that can be applied in a flexible manner depending on the specific points of the execution of the kernel code that needs to be intercepted—since they may have some effect on the state of particular monitored objects. The technology used for hooking is twofold, and is linked to the different ways according to which an eBPF hook can be installed in the Linuk kernel. In particular, our architecture uses both:

- eBPF Tracepoints: these are predefined hooks embedded within the Linux kernel, designed to allow eBPF programs to attach and execute custom logic at key kernel events without modifying kernel source code.

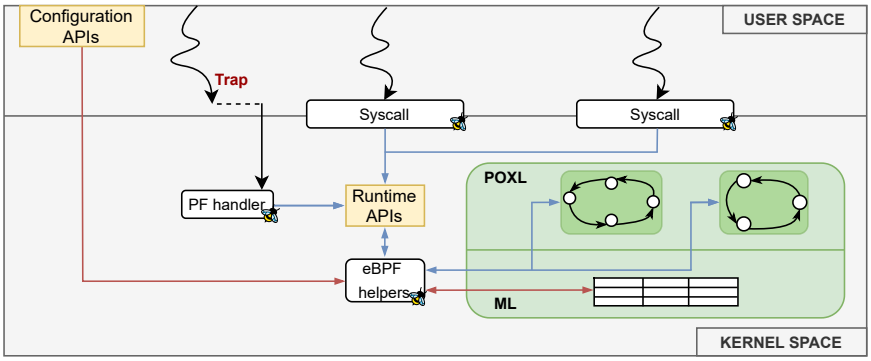

Figure 1: System architecture

- Kprobes: dynamic instrumentation mechanisms used when tracepoints are not available, allowing eBPF programs to hook into almost any kernel function.

Interceptors can be placed at both the entry and exit points of generic kernel functionalities, including system calls. Within each interceptor, activities are performed in our solution at both the levels of the architecture. In particular:

1. POXL: to handle the current object state, its access-key is retrieved and the state information is accessed.
2. ML: the object state information is passed in input to the ML layer, which determines the action to perform according to the rules the user has defined for that specific object type—hence depending on such state and the occurred event intercepted via the hooks.

One key point in the above description is related to the retrieval of the access-key for an object state. In our architecture, the access key can be defined by the user by relying on whatever information can be accessible when running the eBPF hook that traces the occurring events. For example, whatever metadata related to the currently executing thread, such as its PID, can be used. Also, for events generated by hooks installed at the entry points of services that are invoked by the thread—like for example system calls—the key can correspond to whatever parameter the service has received. Actually, we also offer in our architecture the possibility to combine different information to generate the key—like for example combining the PID with the value of a specific parameter of a service. This will enable the access to a specific object type, and to its state, at the POXL level. As for the employed data structures, both the levels of our architecture are implemented using eBPF hash maps, which require a unique identifier for each element due to their exact matching mechanism. Overall, for the two layers, we have the following binding to the hash maps, and to the $< key, value >$ pair that they manage:

- For ML we have what follows:

  Key - Composed of the current state of the object and the event that occurs (e.g., the type of system call intercepted).

  Value - Contains a set of conditions involving environment variables, such as XFSM registers and system call parameters. Based on these variables, the system decides which action to perform, allowing actions to adapt dynamically to the current state.

- For POXL we have what follows:

  Key - A unique identifier for each system object, depending on the type of object (e.g., process ID for processes, frame number for memory frames).

  Value - Represents the object's current state and includes a set of registers that store various information about the object.

The state transitions of the POXL state machines are driven by the configuration of ML. Transitions triggered under specific conditions guide the evolution of per-object XFSMs over time. Since in our monitoring architecture system objects can interact, one object's state may influence another one, allowing a per-object XFSM to trigger a state transition in another per-object XFSM. This is supported by simply maintaining in the state of an object the access key to retrieve the state of another object.

In Figure 1 we schematize tha main parts of our achitecture, also showing the relation with eBPF hooks and system calls or traps.

3.1. Actions and Reactive Engine

The actions to be performed when an object transits into a specific state constitute the "reactive engine" of our solution. These actions are selected from a set implemented using eBPF helper functions or custom kernel functions. By deciding which action to execute, the system's behavior can be customized flexibly depending on the scenario. Examples of actions that are supported in our architecture include:

- Notifying: Alerting users or administrators about suspicious behaviors via logs or user-space notifications.
- Restricting: Modifying or blocking system calls to prevent malicious activities.
- Updating States: Modifying internal states or XFSM registers to reflect changes in object behaviors.
- Interacting with Kernel Modules: Leveraging kernel functions to perform complex tasks or to extend functionality beyond eBPF's capabilities.

Custom kernel functions enable eBPF programs to interact with kernel modules, allowing the development of any required functionality that is not natively supported by eBPF.

3.2. Workflow

The workflow is executed within the interceptors of system calls, or handlers of other events, which, depending on the scenario, can be located either at the entry or the exit point of the intercepted module. It consists of two main phases:

1. **Per-Object XFSM lookup:** this phase retrieves information specific to the object being processed by the intercepted function.
2. **Management XFSM lookup:** this phase evaluates a set of conditions and determines the appropriate actions and state transitions. The management XFSM operates as follows:
    - Conditions are evaluated in priority order, starting from the most critical to the least important, similar to how rules are processed in firewalls. The last rule handled is the default one, which must be processed if no condition is met. This ensures that higher-priority conditions are handled first.
    - Each condition involves one or more checks on parameters and is matched against predefined criteria.
    - Based on the matched conditions, the XFSM executes the following steps: (a) Performs the specified action(s) - (b) Updates parameters or objects' states as needed - (c) Transits to the next state.

This modular design allows the workflow to adapt dynamically to varying conditions by using a flexible matching system for parameter evaluation, ensuring effective control over the behavior of function interceptors.

API	Behavior
`void* add_policy(enum fsm_state state, enum fsm_event event)`	Builds the key based on the given state and event, then adds an entry to the management layer map.
`void *add_combination(struct key_struct, enum action todo, enum fsm_state state, int num_cond, int combination_index)`	Adds a combination by providing the key, action to perform, transition state, number of conditions to check and combination index.
`void *add_condition(int arg_pos, enum param_type type, enum match_type match)`	Adds a condition to a combination by specifying the argument's position, parameter type and matching method.
`void *remove_policy(enum fsm_state state, enum fsm_event event)`	Builds the key based on the given state and event, then removes the entry corresponding to the key from the management layer map.
`void *remove_condition(enum fsm_state state, enum fsm_event event, int index)`	Builds the key based on the given state and event, then removes the condition at the specified index from the object associated with the key in the management layer map
`void *remove_combination(enum fsm_state state, enum fsm_event event, int condition_index, int combination_index)`	Builds the key based on the given state and event, then removes the combination at the specified index from the object associated with the key in the management layer map
`struct stateful_state *find_POXL_entry(__u64 id)`	Retrieves the POXL entry associated to a given identifier
`struct value_struct *find_ML_entry(struct stateful_state *state, enum fsm_event event)`	Builds the key based on the given state and event, then retrieves the ML entry associated to the generated key
`struct combination *retrieve_combination(struct stateful_state *state, enum fsm_event event, struct value_struct *val, union arguments *args)`	Builds the key based on the given state and event, then retrieves the combination according to the key. The combination contains conditions, state changes and registers' updates
`void update_state(struct combination *comb, __u64 id)`	Updates the state of the object with the given identifier according to the combination retrieved

Table 2
Available APIs

3.3. Available APIs

Our system offers the APIs listed in Table 2, which are divided in two families. The first family is used to configure the system. In particular it provides APIs to dynamically add and remove custom policies in ML. The first API enables the creation of state-event-based rules, where conditions and parameter combinations dictate the transition logic. The API supports the seamless integration of new policies into the management map by defining key-value structures along with the associated conditions that must be satisfied. To initialize the key, the user must choose both the state and the event from a predefined list of available options.

To initialize the associated value, the user specifies the number of combinations to evaluate and the number of conditions within each combination. The user then defines the type of parameters and the match type for each condition. For the match type, a predefined list of options is provided for the user to choose from. The user must carefully manage the order of the policies, ensuring that they are defined from the highest priority to the lowest priority.

The system also provides an API to remove specific conditions on parameters. To do so, the user must specify the key of the map and the index of the condition to be removed, ensuring precise control over the conditions associated with a given policy.

Here an example using `state1` and `event1`, the call `add_policy(state1, event1)` creates a key (`struct key_struct` with `current_state=state1` and `current_event=event1`) and a value (`struct value_struct`) holding combinations. Each combination (`struct combination`) defines parameters, conditions, action (`todo`), register updates and next state. In this example the condition is the following:

```
struct param_condition condition = {.arg_pos=0, .match=EQUALS, .value="
    example", .type=STRING};
```

Optionally you can define some register update:

```
struct reg_update update = {.reg_in=some_registry_in, .reg_out=
    some_registry_out, .reg_num=ONE, .value=42, .op=ADD};
```

The system also provides an API to remove specific conditions on parameters. To do so, the user must specify the key of the map and the index of the condition to be removed, ensuring precise control over the conditions associated with a given policy.

A similar mechanism is available for an API that allows the removal of an entire combination. The user specifies the key of the map and identifies the combination to be removed. Doing this the user can remove all the conditions associated to the given parameters' combination. Additionally, the user has the option to remove an entire entry from ML. This API eliminates all policies associated with a specific object, effectively resetting its configuration. This functionality is particularly useful when the user no longer wishes to monitor the state of specific objects. The other family of APIs enables the seamless interaction between POXL and ML. These APIs facilitate object retrieval and state management across the two layers. Specifically, two APIs allow the retrieval of objects from ML and from the POXL. Both APIs internally leverage eBPF helper functions, specifically `bpf_map_lookup_elem`, to fetch the value associated with a key in a hashmap.

Another API is used to determine the actions required based on the current state and event. This API performs a lookup in the ML hashmap for the event being processed, and retrieves the corresponding actions. According to these actions, the object's state is updated through another API, employed to modify the state of the XFSM associated with an object. This API internally leverages the eBPF helper function `bpf_map_update_elem`.

4. Functional Aspects

The designed solution was evaluated over several scenarios to test its flexibility and adaptivity, two of which are illustrate in this section.

4.1. System Calls' Sequence

The first scenario is the sequence of system calls `memfd_create` and `execve`. This is a pattern usually met in *Reflective Code Injection* attacks [20, 21, 22, 23]. Through the call to `memfd_create` the attacker creates a file, then he executes it through the `execve`. Doing so the code injection is performed without touching the hard drive. To face this attack the system intercepts the calls to `memfd_create` and `execve` through tracepoints. The tracepoints are at the entry point of the system calls. Both the tracepoints retrieve the PID of the process that invoked the system call and they update the entry of the per-object XFSM with the new state. If there is no entry associated to a process, they create a new entry.

In the `memfd_create` tracepoint the state changes from `SAFE` to `MEMFD_CREATE_ATTENTION`. If the current state is `MEMFD_CREATE_ATTENTION` in the `execve` tracepoint the critical condition is met and the process is killed.

To check if a condition is met the tracepoint looks into ML. The management map has two entries related to this system call sequence: one associated with the `SAFE` state and the other associated with the `MEMFD_CREATE_ATTENTION` state. In this use case there is no need to use the XFSM's registers, the only action to perform is the XFSM's state evolution.

When an object is no longer of interest, its associated entry must be removed. Since in this use case the process ID serves as the identifier for the entries, the system intercepts the invocations of `do_exit`, and when the corresponding PID is encountered, it deletes the associated entry from the POXL.

4.2. Physical Memory Frame vs Access Permissions

In this use case, checks are performed in relation to the used physical memory frame. The used identifier is the frame number, instead of the process ID. This proves the fact that our system can manage different types of objects. The scenario considered is a *code injection* attack carried out in a distributed manner across two processes [24]. In particular, we may have two processes, each one living in its own address space. A logical page in each of the two address spaces can map on the same physical address. This is a classical scenario when relying on shared-mapping of memory. The first process can set write permissions on its logical page, hence enabling writes on the physical frame, and then the second process can set execution permissions. This sequence of permission changes indicates a potential

cross-process *code injection* attack exploiting a specific physical frame. To properly track this scenario in our system, it is important to remember that logical pages are not immediately materialized into physical frames. In fact, the page is not immediately materialized after the call to `mmap`. Instead, it is materialized when a real access (e.g., write/execute) to the page is made. When a process attempts to access a memory page that has not yet been materialized, a page fault occurs. The operating system's page fault handler is invoked to resolve this event. In a typical scenario, the handler allocates a physical frame, retrieves the required page from disk (if necessary), and maps it into the process's virtual address space, thereby making the page accessible.

In this scenario, our system intercepts `mmap`, `mprotect` and the page fault handler. In particular there are two interceptors for each of them, at the entry and exit points. Relying solely on `mmap` and `mprotect` interceptions would not guarantee that the page has been fully materialized.

Since eBPF tracepoints are not available for the page fault handler, it is intercepted using a combination of a `kprobe` and a `kretprobe`. The `kprobe` captures the parameters passed to the `handle_mm_fault` function, while the `kretprobe` checks whether the page fault handler returns an error. If no error is returned, the `kretprobe` retrieves the parameters saved by the `kprobe` at the entry point of the page fault handler and logs an entry into the per-object XFSM at the POXL level. This log entry uses the actual frame number of the involved memory frame as the key. The `mmap` and `mprotect` system calls are also intercepted at both entry and exit points. At the entry tracepoints, the system saves the arguments of the respective calls. At the exit tracepoints it checks whether the suspicious sequence of permissions set on the memory frame is satisfied. If the condition is not met, the system updates the register that tracks the protection state of the frame. For both system calls, this update occurs at the exit tracepoint, following the evaluation of the return value.

To release the object inserted into the POXL when it is no longer needed, the system intercepts the invocation of `free_pages` through a kprobe. If `free_pages` completes successfully without returning an error, the system deletes the corresponding entry from the XFSM.

This scenario also accounts for cases where two processes share the same memory location but access it through different logical addresses. Such a situation arises when dealing with file mapping. By tracing back to the physical address of the memory frame, the system can track these cases even when the logical addresses differ.

5. Run-time Cost

To assess the run-time cost of our solution, we considered its nesting for the management of two types of operating system services (system calls): virtual file system services and address space management services. Our evaluation aimed to cover both a minimal utilization scenario of our facilities, and an intermediate one, incorporating various tracepoint implementations. Specifically, we analyzed empty tracepoints, tracepoints with map accesses (to simulate XFSM's checks), and their counterparts implemented using kprobes. To ensure the reliability of the performance measurements, we followed established guidelines [25] designed to minimize errors. The measurements were conducted on a server equipped with two `Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz` processors, distributed across two distinct *NUMA nodes*. Each processor features 8 cores, with hyper-threading disabled. The server is configured with four memory modules, each with a capacity of 32 GB, providing a total of 128 GB of system memory.

Before carrying out the measurements, the processor frequency was fixed at 2.10 GHz using the *cpufreq* command, with *turboboost* disabled, to ensure repeatability of the experiments. Additionally, CPU affinity was set to avoid interference by the Linux load balancing mechanism.

5.1. Virtual File System Services

The operations considered are `read` and `write`, and we used the disk-duplicate system tool via the command "`dd if=/dev/zero of=/dev/empty bs=... count=1000000`". This command performs a disk duplicate from the input file to the output file using a given block size. We considered several

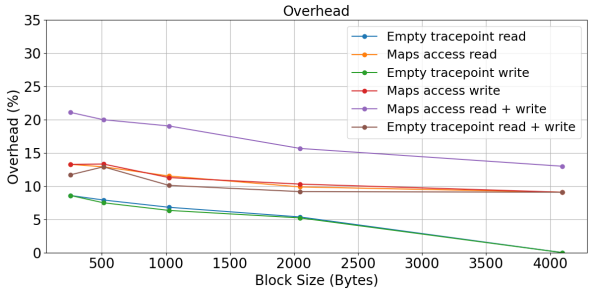

Figure 2: Tracepoints' overhead

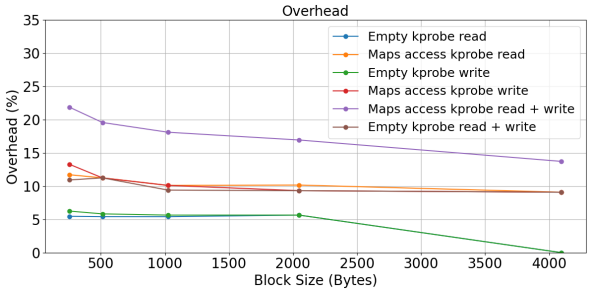

Figure 3: Kprobes' overhead

block sizes—256 B, 512 B, 1024 B, 2048 B and 4096 B—hence evaluating if the overhead was influenced by the block size choice. `/dev/zero` is the well-known device filled with 0s, while `/dev/empty` is an ad-hoc created device, not really storing received data. Hence these devices have negligile real costs of their operations, representing therefore a well case study for the assessment of the pure overhead of our solution.

For every kind of interceptors' combination we evaluated, it emerged that making the block size grow the overhead reduces. This means that, except for specific values of the service parameters, the overhead appears to be low. In particular, as shown in Figures 2 and 3, for block size of 4096—which is the common size used in Linux systems for most I/O operations—the worst overhead generated by the usage of tracepoints or kprobes over `read` and `write` is about 14%, and is 10% or less for the other tested scenarios, even with lower block size.

5.2. Address Space Management Services

To asses the overhead for address space management services we considered `mmap` and `munmap` system calls. To measure the overhead, we used the `perf` command to capture the execution time of a program performing 200000 iterations of the following sequence: a `mmap` operation, a single-byte write to materialize the memory, and a `munmap` operation. To gain reliable results we computed the average time over 100 invocations to `perf`. The results in Figures 4 and 5 show how the pure usage of tracepoints gives rise to essentially negligible costs of our solution, which appear to be more pronounced only when relying on kprobes. We note however that this case study is a real stress scenario, given the un-likelihood of the frequency of such `mmap` and `munmap` of a same area with no real usage of that area for memory accesses related to the program actual workflow.

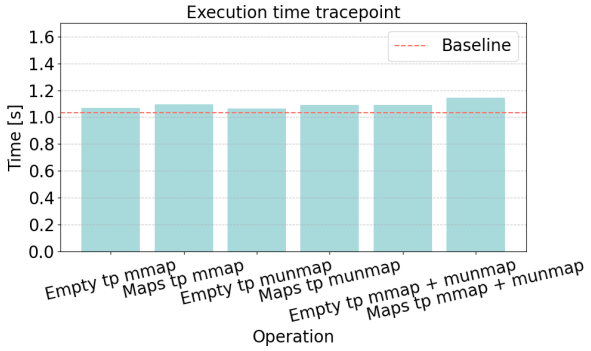

Figure 4: Tracepoints' overhead

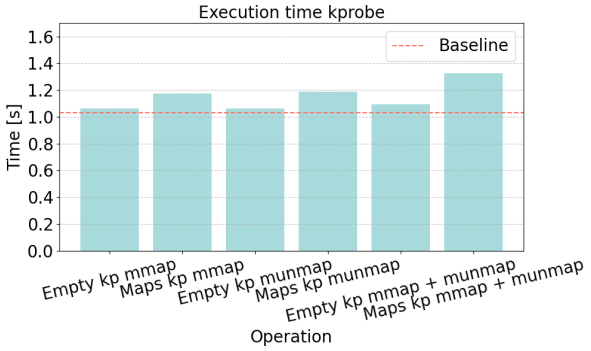

Figure 5: Kprobes' overhead

6. Conclusions and Future Work

The system proposed in this work is designed to provide eBPF developers with the capability to define policies for monitoring/handling complex security scenarios, leveraging the high expressiveness

achieved through the use of a stateful approach. Our approach enables greater expressiveness compared to other currently available solutions, making the system more versatile and suitable for a wide range of security scenarios. The system has been designed with a strong emphasis on performance optimization, ensuring low overhead when its tools are used appropriately. As demonstrated by the use cases we presented, the system can adapt to and protect against several types of attacks, proving its effectiveness and flexibility.

So far, our work has focused on the core architecture—managing state transitions and recording object states—so as to provide a foundation for stateful security-oriented detection mechanisms. However, several enhancements remain on our roadmap. A first area for improvement revolves around the introduction of more versatile key matching for state machines; replacing the current eBPF `hashmap` with a data structure supporting longest-prefix match would allow for broader and more flexible object management, overcoming the current limitation posed by exact matching. In addition, we plan to extend the available APIs, enabling more responsive actions based on user-defined conditions. Another key objective is developing a set of predefined scenarios the system can autonomously handle, minimizing manual intervention and simplifying adoption. Collectively, these improvements would simplify the adoption of the system, enhance its usability, and make it more accessible to users/programmers.

Acknowledgments

This work was partially supported by the projects SERICS (SEcurity and RIghts In the CyberSpace - PE00000014) and RESTART (Telecommunications of the Future - PE00000001) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] J. Singh, G. Singh, S. Negi, Evaluating security principals and technologies to overcome security threats in iot world, in: 2023 2nd International Conference on Applied Artificial Intelligence and Computing (ICAAIC), IEEE, 2023, pp. 1405–1410.

[2] T. F. Authors, Falco, 2023. URL: https://falco.org, [Online].

[3] T. T. Authors, Tetragon, 2023. URL: https://tetragon.cilium.io, [Online].

[4] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, S. Pontarelli, Xtra: Towards portable transport layer functions, IEEE Transactions on Network and Service Management 16 (2019) 1507–1521.

[5] K. T. Cheng, A. S. Krishnakumar, Automatic functional test generation using the extended finite state machine model, in: Proceedings of the 30th International Design Automation Conference, DAC '93, Association for Computing Machinery, New York, NY, USA, 1993, p. 86–91. URL: https://doi.org/10.1145/157485.164585. doi:10.1145/157485.164585.

[6] V. I. Ulyantsev, F. N. Tsarev, Extended finite-state machine induction using sat-solver, IFAC Proceedings Volumes 45 (2012) 236–241. doi:10.3182/20120523-3-RO-2023.00179.

[7] N. Walkinshaw, R. Taylor, J. Derrick, Inferring extended finite state machine models from software executions, Empirical software engineering 21 (2016) 811–853.

[8] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[9] F. J. Lin, P. Chu, M. T. Liu, Protocol verification using reachability analysis: the state space explosion problem and relief strategies, in: Proceedings of the ACM workshop on Frontiers in computer communications technology, 1987, pp. 126–135.

[10] L. Rice, Learning eBPF, " O'Reilly Media, Inc.", 2023.
[11] R. Davoli, M. D. Stefano, BERKELEY PACKET FILTER: theory, practice and perspectives, Ph.D. dissertation, Università di Bologna, 2019.
[12] C. Wright, et al., Linux security modules: General security support for the linux kernel, in: 11th USENIX Security Symposium (USENIX Security 02), 2002.
[13] S. Smalley, C. Vance, W. Salamon, Implementing selinux as a linux security module, NAI Labs Report 1 (2001) 139.
[14] E. de Bourges, Extending selinux to track memory-pages accesses (2011). URL: http://www.mupuf.org/media/files/selinux_memory_access.pdf.
[15] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, C. Min, Protect the system call, protect (most of) the world with bastion, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2023, pp. 528–541.
[16] J. Jia, et al., Programmable system call security with ebpf, arXiv preprint arXiv:2302.10366 (2023).
[17] The Linux Kernel Documentation, Seccomp BPF (Berkeley Packet Filter), 2018. URL: https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html, [Online].
[18] A. Arcangeli, seccomp for 2.6. 11-rc1-bk8, 2005. URL: https://lwn.net/Articles/120192/.
[19] The Linux Kernel Documentation, ptrace() API for PowerPC, 2024. URL: https://www.kernel.org/doc/html/next/powerpc/ptrace.html, [Online].
[20] R. Guo, Reflective code loading in linux — a new defense evasion technique in mitre att&ck v10, https://medium.com/confluera-engineering/reflective-code-loading-in-linux-a-new-defense-evasion-technique-in-mitre-att-ck-v10-da7da34ed301, 2021. [Online].
[21] Stuart, In-memory-only elf execution (without tmpfs), https://magisterquis.github.io/2018/03/31/in-memory-only-elf-execution.html, 2018. [Online].
[22] G. T. Bonicontro, Running elf executables from memory, https://www.guitmz.com/running-elf-from-memory/, 2019. [Online].
[23] M. Salvatori, G. Bernardinetti, F. Quaglia, G. Bianchi, Shiftyloader: Syscall-free reflective code injection in the linux operating system, in: G. D'Angelo, F. L. Luccio, F. Palmieri (Eds.), Proceedings of the 8th Italian Conference on Cyber Security (ITASEC 2024), Salerno, Italy, April 8-12, 2024, volume 3731 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3731/paper02.pdf.
[24] E. Connor, T. McDaniel, J. M. Smith, M. Schuchard, {PKU} pitfalls: Attacks on {PKU-based} memory isolation systems, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1409–1426.
[25] M. Becker, S. Chakraborty, Measuring software performance on linux, arXiv preprint arXiv:1811.01412 (2018).