


Enforcement of In-Kernel Stateful Security Policies via eBPF

Letterio Galletta
IMT School for Advanced Studies Lucca
Lucca, Italy
letterio.galletta@imtlucca.it

Abstract

Many attacks against workloads running on multi-tenant systems are multi-step and history-dependent: sequences of innocuous operations whose malicious nature emerges only over an execution trace. Defending against them requires security policies that are *stateful*, are enforced *within the kernel*, and have a *precise semantics*. Currently deployed proposals fail on at least one count: kernel's built-in syscall filtering and classical MAC frameworks are stateless, while current eBPF-based tools express their policies through ad hoc YAML rules that cannot capture temporal relations among events, and whose semantics is defined only by the implementation.

We present BPFence, an in-kernel runtime-verification framework that satisfies the properties above. BPFence provides a policy language with a formal semantics that can express temporal relations among events. It also provides a type system that statically distinguishes events the kernel can *control* from those it can only *observe*. Every well-typed policy is compiled into a finite-state monitor proved correct with respect to its semantics, and then into eBPF programs that run inside the kernel. We evaluate BPFence on seven case studies drawn from real-world attack patterns, and on a set of micro- and macro-benchmarks to show that the enforcement overhead remains compatible with production deployment.

CCS Concepts

• **Security and privacy → Formal methods and theory of security**; **Operating systems security**; • **Software and its engineering → Domain specific languages**.

Keywords

Runtime verification, formal methods for security, eBPF

ACM Reference Format:
Letterio Galletta. 2026. Enforcement of In-Kernel Stateful Security Policies via eBPF. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846689

1 Introduction

Cloud-native deployments rely on a single shared kernel to host hundreds of mutually distrustful workloads: containers of different tenants, microservices managed by an orchestrator, and short-lived sandboxes spawned by build pipelines. In this setting, the kernel is a critical isolation boundary, and a single compromised process can become a foothold for container escape, lateral movement across tenants, or persistent access to host resources. Operators rely on *security policies* to constrain which operations a workload may perform and keep that boundary effective. However, vulnerabilities such as Dirty Pipe [42] witness how attackers can exploit the system calls allowed by default container profiles to escalate from a compromised application to host privileges by invoking operations that look innocuous in isolation. What distinguishes such attacks is that the malicious behavior is *multi-step* and *history-dependent*: a write to a sensitive file followed by an `exec` syscall, or a successful invocation of `ptrace` followed by a shell spawn. Containing these attack chains or simply detecting them reliably requires security policies that talk about *execution traces*, not just about individual syscalls or system events.

In this setting, effective security policies must satisfy *three* complementary requirements. First, they must be *stateful*: they need to remember the relevant history of the monitored entity, e.g., a process or a container, and react to sequences of events and time windows. Second, the policy language must have a *well-defined semantics*: the operator must be able to say what a policy denotes independently of its implementation, and the toolchain must be able to reject policies it cannot enforce, rather than silently accepting them and producing a monitor whose behavior diverges from the operator's intent. Third, enforcement must happen *inside the kernel*: a userspace monitor sits on the wrong side of the syscall boundary, pays a context switch on every relevant event, and is exposed to time-of-check/time-of-use races. In-kernel enforcement, by contrast, mediates the operation synchronously, on the same execution path that performs it, and observes exactly the arguments the kernel will use. Unfortunately, the mechanisms practitioners actually deploy fail to meet at least one of these three requirements.

On the one hand, system-call filters such as `seccomp-bpf` [26] run entirely in the kernel, but their policies are stateless: a filter cannot remember that the same process has just opened `/etc/shadow`. Most production seccomp profiles, therefore, are simple static allow lists. Mandatory access control frameworks such as SELinux [45] and AppArmor [7] are more expressive, but their decisions are still stateless with respect to recent history and their policies are notoriously hard to author and audit. On the other hand, modern eBPF-based runtime security tools such as Falco [13] and Tetragon [8] leverage programmable in-kernel storage and can react to security-relevant events, providing the mechanisms for stateful monitoring. However, they miss the second requirement: their policies are expressed through ad hoc YAML rules whose semantics is defined only by the implementation: there is no specification of *what* a rule denotes, no static check that a rule asks the kernel to enforce something the kernel can in fact enforce. In addition, multi-step and bounded-time properties must be encoded as collections of correlated rules whose joint behavior is left to the operator's intuition.



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846689

These previous efforts show that the in-kernel machinery to enforce programmable policy is already available, but what is still missing is a policy layer that meets *all* requirements at once: expressive enough to talk about execution traces and time windows, equipped with a semantics the toolchain and the operator can reason about, and enforced inside the kernel.

Our proposal. This paper presents BPFence, an in-kernel runtime-verification framework that meets the three requirements above. BPFence offers a policy language with a formal trace semantics whose constructs capture recurring classes of stateful security properties: safety obligations, bounded-response liveness, historical predicates, sliding-window counts, and cross-event correlation. A type system distinguishes events the kernel can actually *control* (deny or terminate before they take effect) from those it can only *observe*, and statically rejects policies that ask the kernel to enforce something it cannot. Each well-typed policy is compiled to a monitor that is provably correct with respect to its semantics, and then to eBPF programs that run inside the kernel. While related ideas have appeared separately in prior work, BPFence is the first framework to combine them under the eBPF verifier's constraints: the policy language is designed to trade expressiveness for verifier-compatibility, so that the resulting monitors not only pass the eBPF verifier checks but also run with an acceptable overhead, as our evaluation confirms.

Concretely, this paper makes the following contributions:

- *A policy language for in-kernel stateful security properties* (Section 5) with a denotational trace semantics and a type system that statically separates controllable from observable events, ruling out policies the kernel cannot enforce.
- *A monitor synthesis procedure* (Section 6) that turns every well-typed policy into a finite-state monitor. We prove that this monitor is correct with respect to the policy language semantics.
- *A compilation schema* (Section 7) that turns synthesized monitors into eBPF programs accepted by the in-kernel verifier and enforced synchronously on controllable events.
- *A PoC implementation* (Section 8) including the BPFence compiler and runtime, and an essential standard library of LSM hook schemas and policy templates. We evaluate the effectiveness of our implementation on seven case studies drawn from real-world attack patterns; compare the expressiveness of our policy language against some upstream Falco/Tetragon rules; and measure the monitoring overhead via micro- and macro-benchmarks.

Before presenting these contributions, Section 2 reviews the necessary background. Section 3 outlines BPFence through an example, and Section 4 states our threat model. After presenting the contributions, Section 9 situates BPFence with respect to the relevant literature, and Section 10 draws some conclusions and discusses future work. All the auxiliary notions required by our formal development, the proofs of our theorems, and further details on our implementation are in the extended version [30]. The artifact to reproduce our experiments is available online [29].

2 Background

Extended BPF (eBPF) [12] is a technology that allows user-defined programs to run inside the Linux kernel in response to system events, without modifying kernel source code or loading kernel modules. An eBPF program is compiled to a platform-independent bytecode, loaded into the kernel, JIT-compiled to native code, and attached to a *kernel hook*.

Kernel hooks intercept and modify the kernel execution flow at predefined points. They determine *when* an eBPF program executes and *what* it can observe or control. Three hook families are relevant to security monitoring: (i) *LSM hooks* belong to the Linux Security Module framework [47] and execute the eBPF program at security decision points *before* a sensitive operation completes: if the program returns an error code, the operation is blocked synchronously; (ii) *tracepoints* [33] are static probe points scattered throughout the kernel that trigger a program *after* an event has occurred, so they can only observe, not block, a syscall; (iii) *kprobes* [31] are dynamic probes that can be attached to any kernel routine to collect debugging and performance information non-disruptively. Only LSM hooks make an event *controllable*, so we can deny the action or kill the offending process; tracepoints and kprobes make it only *observable*, so we can only raise an alert. This distinction led us to design the type system of Section 5 that statically rejects policies attempting to deny an observable-only event.

At load time, every eBPF program is statically verified by the in-kernel *BPF verifier*, which enforces memory safety and guarantees termination: programs with uninitialized registers, unbounded loops, out-of-bounds accesses, or excessive instruction complexity are rejected. These constraints ensure that eBPF programs cannot crash or hang the kernel, but they restrict the class of expressible computations, a limitation that affects the design of the policy language of Section 5 and the compilation strategy of Section 7.

eBPF programs communicate with one another and with user space through *BPF maps*: key–value data structures that reside in kernel memory and persist across program invocations. There are several map types; the ones most relevant to us are hash maps, arrays, and ring buffers. Maps are shared, so two eBPF programs attached to different hooks can interact using the same map.

3 BPFence in a Nutshell

Consider an attacker who gains initial access to a remote host, e.g., via a web vulnerability, and attempts an SSH lateral movement: read an SSH private key (`/home/user/.ssh/id_rsa`), open a connection to port 22 on an internal host, and execute a command on the remote machine. Each individual action is legitimate: many services read SSH keys, connect on port 22, or spawn shells or subprocesses. Only the full three-step sequence reveals the attack [36].

To detect such an attack sequence and enforce a response to block the offending operations, we define the following BPFence policy:

```
import stdlib linux files     // read syscall
import stdlib linux network   // connect syscall
import stdlib linux process   // exec syscall

let key_read = happened(
  read("/root/.ssh/*") or read("/home/*/.ssh/*")
)

let ssh_connected = happened(
  connect(_, 22)
) when key_read
```

```
12
13 policy lateral_movement {
14   apply to pid action kill
15   forbid exec(_) when ssh_connected
16 }
```

The policy involves three events raised by invoking three syscalls: `read` (file open and read), `connect`, and `exec` (process execution). Each event is bound to an LSM hook via an event schema definition provided by the standard library (e.g., `read` maps to the LSM hook `security_file_open`, which intercepts both open and read operations). The actual parameters of each syscall are automatically extracted at runtime and made available as event fields in the policy (e.g., the file path for `read`, the destination port for `connect`).

The `let` declaration at line 5 introduces the *history predicate* `key_read`. This predicate becomes true when the monitored process reads the SSH private keys of the admin or a user from their `.ssh` directory. Once this happens, the predicate remains true in the future: the `happened` operator records whether a predicate has ever been satisfied. The predicate `ssh_connected` at line 9 is *conditioned* on the previous one: it becomes true only when a connection to port 22 is attempted and `key_read` is already true. An isolated SSH connection or an isolated key read does not affect it. Note that in the two definitions the syntax `*` matches zero or more characters within a single directory level, and `_` denotes "any address".

The policy `lateral_movement` is defined at line 13. Its header specifies that the *enforcement scope* is at the process level (`apply to pid`), namely one monitor instance per process. The body of the policy contains a single `forbid` clause guarded by `ssh_connected`: when the predicate holds, any subsequent `exec` call is blocked and the process is terminated (`action kill` in the policy header). For this to be possible in practice, the `exec` event in the `forbid` clause must be bound to an LSM hook that can deny the corresponding operation (controllable event). Our type system statically ensures the enforceability of the policy: it assigns the violation type C (controllable) to the event `exec` according to its hook and checks that the `kill` action is indeed compatible with the type. Had `exec` been bound to a tracepoint (observable only), the compiler would have rejected the policy due to a type mismatch.

For the lateral movement policy, the compiler generates three BPF programs, one per hook, that share their state through a common eBPF map. A user-space daemon loads the generated BPF bytecode, sets up the shared state, attaches the programs to the corresponding hooks, and starts dispatching violation events that the loaded programs generate. The next sections formalize each phase and present the implementation.

Finally, note that current eBPF-based monitoring tools cannot enforce policies such as the lateral movement above. `seccomp-bpf` [26] and AppArmor [7] are *stateless*: they can block *all* connections to port 22 or *none*, but cannot condition the decision on a previously observed file read. Falco [13] can only *alert* on individual events; multi-step detection requires external engines. Tetragon [8] can block operations, but its filtering conditions can only inspect fields of the *current* event: there is no mechanism to maintain state across events and condition a decision on what happened earlier. Additional comparisons are in Section 8.

4 Threat Model

We consider a Linux host running user-space processes, possibly inside containers, monitored by BPFence. We assume the BPFence loader installs the compiled monitors *before* process execution begins. The trusted computing base (TCB) comprises the Linux kernel (including the eBPF verifier and runtime), the BPFence compiler, its loader, and the hardware.

The attacker can execute arbitrary code in a monitored process and can issue any system call supported by the execution environment. Attack strategies include multi-step privilege escalation, lateral movement, and data exfiltration. We do not address kernel exploits, verifier bugs, or side-channel attacks. We assume that the attacker cannot compromise any component of the TCB, and that they do not actively try to evade known BPFence policies (e.g., by splitting a kill chain across distinct processes via `fork`) or attack the monitor itself.

Under this threat model, we show that the generated monitors enjoy *soundness* (no false positives) and *detection completeness* (no missed violations), assuming that events are attached to the relevant hooks. For controllable events, enforcement is *synchronous*: the offending system call is denied before the kernel operation completes. We also assume that updates to the monitor states are atomic and that policy reloads never expose a partially configured monitor, so guarantees hold throughout the lifetime of each monitored process; Sections 7.2 and 8.1 discuss how the implementation discharges these assumptions.

5 The Policy Language

BPFence policies are written in a declarative domain-specific language (DSL) whose formal semantics is defined over a data-aware temporal logic. The logic operates on traces of events that abstractly represent the system calls invoked by a running program, and combines three key features: temporal operators from timed linear temporal logic with past [6]; a three-valued semantics over the domain $\mathbb{B}_3 = \{\text{ok}, \text{pending}, \text{violated}\}$ to express monitor decisions; and data-aware event predicates that bind and constrain field values across events via correlation variables. In addition, a type system statically classifies each policy by how violations arise at runtime and ensures that enforcement actions are associated only with policies whose violations can be effectively intercepted.

We present the trace model (Section 5.1), the temporal logic (Section 5.2), the DSL (Section 5.3), and its type system (Section 5.4).

5.1 Event Model

We abstractly represent a program execution as a trace of events generated by system call invocations. According to the nature of events in eBPF, we distinguish between two types: events that can be intercepted via LSM hooks, enabling pre-event enforcement, and events that are only observable post hoc via tracepoints or kprobes.

Formally, let $\Sigma = \Sigma^C \uplus \Sigma^O$ be a finite set of event types, partitioned into *controllable* events Σ^C and *observable* events Σ^O. An *event* e is a tuple $\langle E, \mathit{fields}, t\rangle$ where $E \in \Sigma$, *fields* is a finite typed parameter map, and t is a monotone timestamp. An *execution trace* is a finite, time-ordered sequence of events $\sigma = e_1 e_2 \cdots e_n$ with $e_i.t < e_{i+1}.t$ where we use the dot notation to access the fields of e.

In the following, we denote by σ_k a trace of length k and by e_k its last event.

5.2 Data-aware Temporal Logic

Our data-aware temporal logic is defined at two levels: event predicates and temporal formulas, both of which use a three-valued semantics with an environment to handle variable binding.

Event predicates. Intuitively, an event predicate is a propositional formula evaluated on the events of a trace. Since events carry data values, we introduce the notions of *variables* and *guards* to refer to these values and express constraints over them.

Formally, let $\mathcal{V}$ be a set of variables (ranged over by $X, Y, \ldots$). An *event predicate* is defined as:

$$P ::= \alpha \mid \neg P \mid P \wedge P \mid P \vee P \qquad \alpha ::= (E, g)$$
$$g ::= \mathsf{true} \mid \mathsf{field} \sim \ell \mid \mathsf{field} = ?X \mid \mathsf{field} = X \mid g \wedge g$$

where an *event atom* $\alpha = (E, g)$ pairs an event type E with a guard g. A guard g is a conjunction of constraints over the fields of an event, where $\ell \in \mathsf{Val}$ is a literal value, $\sim \in \{=, \neq, <, \leq, >, \geq\}$ is a comparison operator, and field denotes the name of an event field. In addition, a guard introduces two variable constructs with distinct roles: $\mathsf{field} = ?X$ is a *binding*, which captures the value of a field into a fresh variable X; whereas $\mathsf{field} = X$ is a *constraint*, requiring the field to be equal to the previously bound variable X. Note that the correspondence between kernel hooks and the events referenced in our predicates P is assumed to be given by Σ.

We assume hereafter that a well-formedness condition holds: each variable must be bound before use, and no bindings may appear under $\neg$ or $\vee$ (see the extended version [30]). Given an event e and an *environment* $\rho : \mathcal{V} \to \mathsf{Val}$, we interpret an event predicate P on a three-valued boolean domain $K_3 = \{\mathit{true}, \mathit{false}, \mathit{na}\}$ using a strong Kleene semantics [27]. The idea is that *inapplicable* value *na* captures situations in which the event e is not among those P is talking about. Formally, the semantics of event predicate P is given by $[\![P]\!]^a(e, \rho) = (v, \rho')$ where $v \in K_3$ and ρ' is an updated environment with new bindings introduced by the guards of P. The complete definition of $[\![\cdot]\!]^a$ is in the extended version [30]. Hereafter, we write $[\![P]\!]^a(\sigma_k, \rho)$ to denote the application of $[\![\cdot]\!]^a$ to the last event e_k of a trace σ_k.

Example 5.1 (Event predicates). Consider the event predicate at line 6 of the policy of Section 3, written as

$$P_k = (\mathtt{read},\ \mathit{path} \sim \mathtt{/root/.ssh/*}) \vee (\mathtt{read},\ \mathit{path} \sim \mathtt{/home/*/.ssh/*})$$

where $\sim$ denotes glob matching, and the following three events:

$$e_1 = \langle \mathtt{read},\ \{\mathit{path} \mapsto \mathtt{/home/alice/.ssh/id_rsa}\},\ t_1 \rangle$$
$$e_2 = \langle \mathtt{connect},\ \{\mathit{dest} \mapsto \mathtt{10.0.0.7},\ \mathit{port} \mapsto 22\},\ t_2 \rangle$$
$$e_3 = \langle \mathtt{read},\ \{\mathit{path} \mapsto \mathtt{/etc/hosts}\},\ t_3 \rangle.$$

We have $[\![P_k]\!]^a(e_1, \emptyset) = (\mathit{true}, \emptyset)$ because the event arguments match those of the second disjunct. In contrast, $[\![P_k]\!]^a(e_2, \emptyset) = (\mathit{na}, \emptyset)$: the predicate P_k does not apply to e_2 because it simply does not concern connect events. Finally, $[\![P_k]\!]^a(e_3, \emptyset) = (\mathit{false}, \emptyset)$: the event type of e_3 matches, but the guard fails on both disjuncts. Telling *false* apart from *na* allows the monitor to evaluate predicates over unrelated event types without spurious violations.

Temporal formulas. The syntax of temporal formulas is:

$$\phi ::= P \mid \neg\phi \mid \phi \wedge \phi \mid {<}\phi \mid G\,\phi \mid P_1\, F^{\leq \Delta}\, P_2$$

where P is an event predicate, $<$ is the *once* operator, G is the *always* operator, and $F^{\leq \Delta}$ is a *bounded eventually* operator where $\Delta \in \mathbb{N}$ is a finite temporal bound: $P_1\, F^{\leq \Delta}\, P_2$ requires that every time P_1 holds, P_2 holds within Δ time units. We assume that a well-formedness condition holds to ensure each variable is bound before use and that no binding appears under $\neg$ or $<$ (see the extended version [30]).

The temporal formulas are evaluated over the semantic domain $\mathbb{B}_3 = \{\mathsf{ok}, \mathsf{pending}, \mathsf{violated}\}$. The value ok indicates that the monitor has not detected any violation so far, while violated indicates that a bad prefix has been detected. The value pending denotes that the truth of the formula cannot yet be decided due to open obligations: based on the information currently available, the monitor requires additional future events to reach a verdict.

Formally, given a trace σ_k and an environment ρ, the semantics of a formula ϕ is given by $[\![\phi]\!](\sigma_k, \rho) = (v, \rho')$ where $v \in \mathbb{B}_3$ and ρ' is an updated environment. The evaluation function $[\![\cdot]\!]$ is inductively defined over the syntax of formulas. Below, we illustrate the semantics of the once and always operators through an example and report the definition of $[\![\cdot]\!]$ for the case of bounded response; the other cases are in the extended version [30].

Example 5.2 (Once and always). Consider the predicate P_k of Example 5.1 and the trace $\sigma_3 = e_1 e_2 e_3$ built from the events taken from that example. Also, let P_e be the single-atom predicate (exec, true). We have that $[\![{<}P_k]\!](\sigma_3, \emptyset) = \mathsf{ok}$ since e_1 reads a key; in contrast, $[\![{<}P_e]\!](\sigma_3, \emptyset) = \mathsf{pending}$ because no exec has occurred yet and a future one could still make it ok. Finally, $[\![G(\neg P_k)]\!](\sigma_3, \emptyset) = \mathsf{violated}$, since P_k holds on the prefix e_1, making $\neg P_k$ violated there, and no extension can retract it. Both temporal operators look at all the prefixes of σ_3, but do so differently: the once operator asks whether some prefix succeeded, whereas the always operator asks whether some prefix failed.

For *bounded response* formulas $P_1 F^{\leq \Delta} P_2$, the pending value represents an obligation that is still unresolved: if the obligation is not fulfilled within the specified time bound, a violation occurs. We define the *active instance set* generated every time P_1 holds:

$$\mathcal{I}(\sigma_k, \rho) \stackrel{def}{=} \left\{ (i, \rho_i, d_i) \;\middle|\; i < k,\ [\![P_1]\!]^a(\sigma_i, \rho) = (\mathit{true}, \rho_i),\ d_i = e_i.t + \Delta \right\}$$

where every instance (i, ρ_i, d_i) records the index i of the event where P_1 was true, the produced environment ρ_i, and the absolute deadline d_i. An instance (i, ρ_i, d_i) is satisfied on σ_k if:

$$\mathsf{sat}(i, \rho_i, d_i, \sigma_k) \stackrel{def}{\iff} \exists\, j \in (i, k].\ e_j.t \leq d_i \ \wedge\ \pi_1\big([\![P_2]\!]^a(\sigma_j,\ \rho_i)\big) = \mathit{true}$$

Then, the semantics of the formula is:

$$[\![P_1\, F^{\leq \Delta}\, P_2]\!](\sigma_k,\ \rho) \ =\ (v,\ \rho)$$

where

$$v = \begin{cases} \text{violated} & \exists (i, \rho_i, d_i) \in \mathcal{I}(\sigma_k, \rho).\ e_k.t > d_i \wedge \neg \mathsf{sat}(i, \rho_i, d_i, \sigma_k) \\ \text{pending} & \exists (i, \rho_i, d_i) \in \mathcal{I}(\sigma_k, \rho).\ e_k.t \leq d_i \wedge \neg \mathsf{sat}(i, \rho_i, d_i, \sigma_k) \\ \text{ok} & \text{otherwise} \end{cases}$$

The first case captures the situation in which the deadline has expired, and the expected event has never occurred; from that point on, the violation is definitive and cannot be remedied by any future events. The second case captures the situation in which there is still time before the deadline, and thus the monitor can continue waiting. The third case captures the situation in which the formula becomes ok within the prescribed time bound. Note that the pending value arises only when an obligation is active, but its deadline has not yet expired, and that it is always transient since all deadlines are finite.

Example 5.3 (Bounded response and bindings). Consider the following formula ϕ, which requires every file to be closed within five seconds of being opened:

$$\phi \;=\; G\big((\mathsf{open},\ \mathit{ino} = ?X)\ F^{\leq 5\mathrm{s}}\ (\mathtt{close},\ \mathit{ino} = X)\big)$$

The open atom binds the i-node field to the variable X, and the close atom then constrains its own i-node to match X. In this way, the two events are correlated by the value of the i-node. Upon the event $e_4 = \langle \mathsf{open}, \{\mathit{ino} \mapsto 8123\}, t_4\rangle$ the open atom yields $(\mathit{true}, \{X \mapsto 8123\})$, so the instance $(4, \{X \mapsto 8123\}, t_4 + 5\mathrm{s})$ enters $\mathcal{I}$, and the close atom is evaluated under the environment it records. Only a close on i-node 8123 satisfies the instance, making sat hold and the verdict ok. Any event other than a close of i-node 8123 leaves the verdict pending; once the deadline expires, it turns violated. Note that the correlation between related open and close events would not be possible without X.

5.3 The BPFence DSL

The BPFence DSL is a user-oriented declarative language for expressing security policies. It provides structured constructs for common security patterns, including *safety*, *bounded response*, *history-based*, and *taint* policies. The DSL is intentionally designed to ensure online monitorability and runtime enforceability and to effectively target eBPF. The DSL has no independent semantics, but each policy denotes a formula of the temporal logic defined above: the DSL is thus a monitorable fragment of the logic. Note that since $[\![\Pi]\!]$ is a function of a single trace, BPFence can express and monitor only trace properties: hyperproperties such as non-interference are outside the scope of this paper.

Syntax. Given a finite set of event types Σ, the abstract syntax of a BPFence program is:

$$\Pi \;::=\; \langle d^*\ p, A\rangle$$

where d^* is a (possibly empty) sequence of historical-predicate definitions, p is the policy body, and $A \in \{\textsc{Alert}, \textsc{Deny}, \textsc{Kill}\}$ is the action to take when a violation occurs:

$$d \;::=\; \mathtt{let}\ H := \mathsf{happened}(P)[\mathtt{when}\ H']$$

$$p \;::=\; \mathtt{forbid}\ P\ [\mathtt{when}\ H]\ \mid\ \mathtt{when}\ P_1\ \mathtt{then\ within}\ \Delta\ P_2\ \mid\ p_1 \wedge p_2$$

where P, P_1, P_2 are event predicates introduced above, $H, H' \in \mathit{Id}$ are historical-predicate identifiers, and $\Delta \in \mathbb{N}$ is a finite time bound.

A historical-predicate definition d introduces a predicate that captures a property depending on the past history of events. A declaration introduces a new predicate named H and may include an optional when clause, conditioning the definition of H on another historical predicate H'. Historical predicates can be referenced in the policy body p. A forbid clause specifies an event predicate P that must never hold; it may optionally include a when clause, restricting the prohibition to situations in which the historical predicate H holds. A when ... then within clause expresses a bounded response property: whenever P_1 holds, P_2 must hold within the time bound Δ. Finally, policy clauses can be combined via conjunction.

Semantics. The semantics is given by first translating the policy into a temporal formula and then interpreting this formula on a trace σ_k. More precisely, we proceed as follows. The historical-predicate declarations d^* evaluate to a definition environment D mapping each identifier H to a logical formula. We have $D(H) = {<}P$ for $\mathtt{let}\ H := \mathsf{happened}(P)$, and $D(H) = {<}(D(H') \wedge P)$ for the when clause. Acyclicity of d^* (enforced statically by the type system) ensures D is well-defined.

Then, given a definition environment D and a policy body p, we define the *function* $[\![p]\!]^D_{\mathrm{DSL}}$ that maps policies to temporal formulas, inductively on the syntax of p:

$$[\![\mathtt{forbid}\ P]\!]^D_{\mathrm{DSL}} = G(\neg P)$$

$$[\![\mathtt{forbid}\ P\ \mathtt{when}\ H]\!]^D_{\mathrm{DSL}} = G\big(\neg(D(H) \wedge P)\big)$$

$$[\![\mathtt{when}\ P_1\ \mathtt{then\ within}\ \Delta\ P_2]\!]^D_{\mathrm{DSL}} = G\big(P_1\ F^{\leq \Delta}\ P_2\big)$$

$$[\![p_1 \wedge p_2]\!]^D_{\mathrm{DSL}} = [\![p_1]\!]^D_{\mathrm{DSL}} \wedge [\![p_2]\!]^D_{\mathrm{DSL}}$$

A forbid clause is translated into a G formula, while a when ... then within clause is translated into an always bounded-response formula. From the definition above, it follows directly that every policy is translated into a conjunction of G formulas.

Example 5.4 (From policy to formula). Consider the lateral_movement policy of Section 3, whose body p is the clause forbid P_e when ssh_connected, where P_e is as in Example 5.2. The two let declarations yield an environment D with the bindings

$$D(\mathtt{key_read}) = {<}P_k$$

$$D(\mathtt{ssh_connected}) = {<}({<}P_k \wedge P_c)$$

where $P_c = (\mathtt{connect},\ \mathit{port} = 22)$ and P_k is as in Example 5.1, making it evident that the second declaration is conditioned on the first. By the forbid–when case of $[\![\cdot]\!]^D_{\mathrm{DSL}}$ we obtain

$$\begin{aligned} [\![p]\!]^D_{\mathrm{DSL}} &= G\big(\neg(D(\mathtt{ssh_connected}) \wedge P_e)\big) \\ &= G\big(\neg({<}({<}P_k \wedge P_c) \wedge P_e)\big) \end{aligned}$$

as the denotation of our policy. Note that the nesting of $<$ ensures that the inner ${<}P_k$ already holds when the connection occurs, and that both hold when the exec is attempted.

Finally, given a BPFence program $\Pi = \langle d^*\ p, A\rangle$ and a trace σ_k, the semantics of Π is defined as:

$$[\![\Pi]\!](\sigma_k) \;\triangleq\; \pi_1\big([\![\phi]\!](\sigma_k,\ \emptyset)\big) \;\in\; \mathbb{B}_3 \qquad \text{where } \phi = [\![p]\!]^{D(d^*)}_{\mathrm{DSL}}.$$

We first translate Π into a temporal formula ϕ, evaluate ϕ on the trace σ_k and an empty environment $\emptyset$, and take the resulting value in $\mathbb{B}_3$: ok means no clause is violated and no obligation is pending;

pending means at least one $F^{\leq\Delta}$ obligation is active but its deadline has not expired; violated means a safety clause is violated *now*, or a deadline has elapsed without the required event. Crucially, violated is irrevocable: no future extension of the trace can undo it. When a violation occurs, the monitor performs action A.

Formal results. The following two results characterize the monitorability of our language. The first establishes that once a DSL program detects a violation, no future event can undo the decision:

Theorem 5.5 (Violation Persistence). *For every DSL program* Π *and finite traces* σ_k, σ_j *such that* σ_k *is a prefix of* σ_j, *if* $[\![\Pi]\!](\sigma_k) =$ violated *then* $[\![\Pi]\!](\sigma_j) =$ violated.

The second result ensures that each violating trace has a first violating prefix:

Proposition 5.6 (Existence of the first violating prefix). *Let* Π *be a DSL program and* σ_k *be a finite trace such that* $[\![\Pi]\!](\sigma_k) =$ violated. *There exists a unique prefix* σ_{i^*} *of* σ_k *such that*

$$i^* = \min\{\, i \leq k \mid [\![\Pi]\!](\sigma_i) = \text{violated} \,\}.$$

We call σ_{i^*} *the* first violating prefix *of* σ_k.

5.4 Type System

The semantics defines when an execution trace violates a policy but does not guarantee that a policy can be effectively enforced at runtime. In particular, a semantically valid policy may still be: *not enforceable*, because a violation is triggered by an observable event that can only be detected after its occurrence; or *not directly enforceable*, because the violation arises from a deadline expiration rather than from a specific controllable event. To address these issues, we introduce a type system that classifies DSL programs according to how violations may arise at runtime: due to a controllable event (type C), an observable event (type O), or by a deadline expiration (type T). The type system statically verifies that the action A associated with the program is compatible with its violation type, e.g., Deny is permitted only for policies of type C. Since types describe modes of violation and enforcement capabilities, we refer to them as *violation types*. In addition to the typing rules presented here, our implementation also verifies standard well-formedness: every identifier must be declared before use, historical-predicate definitions must be acyclic, and values used in event-predicate guards must match the types declared in the corresponding event schemas.

Types and type environment. The set of violation types consists of three elements $\tau \in \mathcal{T} = \{C, O, T\}$ where C means that the violation is triggered by a controllable event; O that the violation is caused by an observable event; and T that the violation arises from a deadline expiring without the required event. The types are ordered to form a finite lattice $C \sqsubseteq O \sqsubseteq T$: $C \sqsubseteq O$ because a controllable event is a special case of an observable one, and $O \sqsubseteq T$ because a violation anchored to a specific event is strictly more enforceable than one determined by a timeout. Hereafter, we denote by $\sqcup$ the join operator of the type lattice.

The classification of events as controllable or observable depends on the deployment environment. We capture this dependency through a typing environment Γ, which assigns to each event type $E \in \Sigma$ its corresponding violation type: $\Gamma(E) = C$ if the event E can be intercepted via an LSM hook, and $\Gamma(E) = O$ if E is obtained through a tracepoint or kprobe hook.

Since policy bodies may refer to historical predicates H, we introduce an environment $\Theta\colon Id \to \mathcal{T}$ that records the violation type of each identifier. The environment Θ is constructed inductively from the historical-predicate definitions d^* occurring in the program, as defined by the rules in the extended version [30].

Typing rules. The most relevant typing rules are in Figure 1 (the full set is in the extended version [30]). The rules for event predicates, characterized by the judgment $\Gamma \vdash P : \tau$, assign a violation type τ to an event predicate P under the environment Γ. The type for an event $\alpha = (E, g)$ is taken from the environment Γ (rule PEvent). Rule PAnd joins the types of the two sub-terms to ensure that a violation can occur in the worst possible way.

The rules for policies have typing judgments of the form $\Gamma; \Theta \vdash p : \tau$. Rule Comp takes the join of the types of the two sub-terms, reflecting that a violation may arise from either of the conjuncts. In rule Forbid, the resulting type is that of the event predicate P, reflecting the fact that `forbid` denotes a safety property whose violations are always anchored to a specific event. Rule ForbidWhen is similar, but the type is $\tau_H \sqcup \tau_P$, where τ_P is the type of P and τ_H is the type of the historical predicate H. In the previous two rules, the join means that a violation occurs when both $D(H)$ and P hold simultaneously. Since both event predicates and historical predicates are typed in $\{C, O\}$, in either case the resulting type is at most O, never T: no timeout can cause a violation. In rule Response, the premises $\Gamma \vdash P_1 : \tau_1$ and $\Gamma \vdash P_2 : \tau_2$ ensure that P_1 and P_2 are well-typed event predicates, but their types do not affect the resulting type T, which is determined solely by the temporal structure $F^{\leq\Delta}$: any policy with a bounded-response obligation may produce a timeout violation, regardless of whether the involved events are controllable or observable.

Rule Prog derives the environment Θ from the declarations d^*, obtains the violation type τ of the policy body p, and checks that the action A is compatible with τ via the predicate Allowed:

$$\text{Allowed}(\tau, A) \triangleq \begin{cases} A \in \{\text{Alert}, \text{Deny}, \text{Kill}\} & \text{if } \tau = C, \\ A = \text{Alert} & \text{if } \tau \in \{O, T\}. \end{cases}$$

The actions Deny and Kill are permitted only when $\tau = C$, while the action Alert is always permitted, since the monitor can always raise a warning. Note that the type τ of a policy depends on the hooks available in the deployment environment: the same event may have type C where an LSM hook is available and type O otherwise, making the policy Deny-able in the first case and only Alert-able in the second.

Example 5.7 (Typing a policy). Consider again the `lateral_movement` policy of Section 3, on a deployment where `read`, `connect`, and `exec` are all bound to LSM hooks, so that Γ assigns C to all of them. The let declarations yield an environment Θ such that $\Theta(\texttt{key_read}) = \Theta(\texttt{ssh_connected}) = C$. To type the policy, we apply rule ForbidWhen, which joins the type of `ssh_connected` with that of `exec`, resulting in $\Gamma; \Theta \vdash p : C$ for its body p. Since Allowed(C, Kill) holds, Prog accepts the program with the `kill` action. Consider extending the policy with a new clause denoting the formula of Example 5.3, which requires every file to be closed within five seconds of being opened. Even when Γ assigns C to both

$$\textsc{PEvent}\ \frac{}{\Gamma \vdash (E, g) : \Gamma(E)} \qquad \textsc{PAnd}\ \frac{\Gamma \vdash P_1 : \tau_1 \quad \Gamma \vdash P_2 : \tau_2}{\Gamma \vdash P_1 \wedge P_2 : \tau_1 \sqcup \tau_2} \qquad \textsc{Comp}\ \frac{\Gamma;\Theta \vdash p_1 : \tau_1 \quad \Gamma;\Theta \vdash p_2 : \tau_2}{\Gamma;\Theta \vdash p_1 \wedge p_2 : \tau_1 \sqcup \tau_2} \qquad \textsc{Forbid}\ \frac{\Gamma \vdash P : \tau}{\Gamma;\Theta \vdash \texttt{forbid}\ P : \tau}$$

$$\textsc{ForbidWhen}\ \frac{\Gamma \vdash P : \tau \quad \tau_H = \Theta(H)}{\Gamma;\Theta \vdash \texttt{forbid}\ P\ \texttt{when}\ H : \tau \sqcup \tau_H} \qquad \textsc{Response}\ \frac{\Gamma \vdash P_1 : \tau_1 \quad \Gamma \vdash P_2 : \tau_2}{\Gamma;\Theta \vdash \texttt{when}\ P_1\ \texttt{then within}\ \Delta\ P_2 : T} \qquad \textsc{Prog}\ \frac{\Gamma;\emptyset \vdash d^* : \Theta \quad \Gamma;\Theta \vdash p : \tau \quad \mathsf{Allowed}(\tau, A)}{\Gamma \vdash \langle d^*\, p,\ A\rangle : \checkmark}$$

Figure 1: Relevant typing rules for BPFence DSL.

open and `close`, rule Response assigns the type T to the new clause because the violation may come from the expiry of the deadline. By rule Comp, the type propagates to the whole program, so, for the Allowed predicate to hold, the action can only be Alert.

Type soundness. The following theorem establishes the soundness of the type system with respect to the semantics: the violation type τ assigned to a policy is a sound upper bound on the controllability class of the event that triggers the violation at runtime, when such an event exists:

Theorem 5.8 (Type Soundness). *Let Γ be a type environment, $\Pi = \langle d^*\, p, A\rangle$ be a DSL program such that $\Gamma;\emptyset \vdash d^* : \Theta$ and $\Gamma;\Theta \vdash p : \tau$. For each finite trace σ_k such that $[\![\Pi]\!](\sigma_k) = \mathsf{violated}$, let σ_{i^*} be the first violating prefix of σ_k. Then:*

- *if $\tau \in \{C, O\}$, the violation is triggered by an offending event e_{i^*} such that $\Gamma(e_{i^*}.type) \sqsubseteq \tau$;*
- *if $\tau = T$, either the violation is triggered by an offending event e_{i^*} such that $\Gamma(e_{i^*}.type) \sqsubseteq O$, or it is due to a deadline expiration: there exists a bounded-response sub-formula $P_1 F^{\leq\Delta} P_2$ of $[\![p]\!]^D_{\mathsf{DSL}}$ and an instance $(i, \rho_i, d_i) \in \mathcal{I}(\sigma_{i^*}, \emptyset)$ of its active instance set such that $e_{i^*}.t > d_i$ and $\neg\,\mathsf{sat}(i, \rho_i, d_i, \sigma_{i^*})$.*

6 Monitor Synthesis

A DSL program is transformed into a *monitor automaton* that performs the monitoring and the enforcement. We first introduce a *normal form* to capture a structural property of the denotations of DSL programs (Section 6.1). We then define the synthesis function GenMon, which translates such normal form into a monitor, and establish that the resulting monitor is sound and complete with respect to the semantics of BPFence DSL (Section 6.2).

6.1 Normal Form

Every formula produced by the denotation function $[\![\cdot]\!]_{\mathsf{DSL}}$ of Section 5 either has the G operator at the outermost level or is a conjunction of two formulas. Since the operator G distributes over $\wedge$ (see the extended version [30]), we can always collect all conjuncts under a single G, obtaining the following normal form:

$$G\Big(\bigwedge_{i\in I} \neg\psi_i \ \wedge\ \bigwedge_{j\in J} \big(P_1^j\, F^{\leq\Delta_j}\, P_2^j\big)\Big)$$

where each ψ_i is either an event predicate P or a formula of the form $<\!\phi_H \wedge P$, whereas P_1^j, P_2^j are event predicates, $\Delta_j \in \mathbb{N}$ is a finite bound, and I, J are finite index sets. The formula $<\!\phi_H$ is the one associated by D with the historical predicate H. The rationale is that the normal form highlights two kinds of constraints a policy can express. The conjuncts $\neg\psi_i$ express safety conditions: a violation occurs when ψ_i holds on the current event, either because the event predicate P is satisfied directly (`forbid` P), or because both the past condition $<\!\phi_H$ and the predicate P hold simultaneously (`forbid` P `when` H). The conjuncts $P_1^j F^{\leq\Delta_j} P_2^j$ express bounded-response obligations that are violated by a deadline expiry.

Given a DSL program $\Pi = \langle d^* p, A\rangle$, we define the normalization function $\mathsf{NF}(\Pi)$ inductively on the structure of p (see the extended version [30]). Intuitively, for a `forbid` clause, it produces a safety conjunct $\neg\psi_i$; for a `then within` clause, it produces a bounded-response conjunct $P_1^j F^{\leq\Delta} P_2^j$. For a conjunction $p_1 \wedge p_2$, it applies NF recursively to p_1 and p_2, and merges the resulting formulas into a single G. The normalization is correct with respect to the semantics of formulas, processes each clause exactly once, and terminates in a number of steps equal to the number of clauses in p (see the extended version [30]).

Example 6.1 (Normal form of a two-clause policy). Consider the `lateral_movement` policy of Section 3 extended, as in Example 5.7, with a bounded-response clause that requires every file to be closed within five seconds of being opened. Denote by $P_o = (\mathsf{open},\ ino = ?X)$ and by $P_{cl} = (\mathtt{close},\ ino = X)$ the two predicates of Example 5.3. The body of the extended policy is the conjunction $p = p_1 \wedge p_2$ of the two clauses, so its denotation is $[\![p]\!]^D_{\mathsf{DSL}} = [\![p_1]\!]^D_{\mathsf{DSL}} \wedge [\![p_2]\!]^D_{\mathsf{DSL}}$:

$$[\![p_1]\!]^D_{\mathsf{DSL}} = G\big(\neg(<(<P_k \wedge P_c) \wedge P_e)\big)$$
$$[\![p_2]\!]^D_{\mathsf{DSL}} = G\big(P_o\ F^{\leq 5\mathrm{s}}\ P_{cl}\big)$$

where the denotation of p_1 is the one computed in Example 5.4. Each clause of p thus carries its own G. The normalization function NF collects the two into a single one:

$$\mathsf{NF}(\Pi)\ =\ G\big(\neg(<(<P_k \wedge P_c) \wedge P_e)\ \wedge\ P_o\ F^{\leq 5\mathrm{s}}\ P_{cl}\big)$$

6.2 Building Monitor Automata

The normal form plays a key role in monitor synthesis: by the semantics of G, every violated outcome of $\mathsf{NF}(\Pi)$ on a prefix σ_k is due to a violation of at least one conjunct, either a safety conjunct $\neg\psi_i$ or a bounded-response conjunct $P_1^j F^{\leq\Delta_j} P_2^j$. This separation allows us to construct the monitor compositionally: we synthesize a local automaton for each conjunct and then combine them into a single monitor by parallel composition.

Specifically, a *monitor automaton* is a tuple

$$\mathcal{A} = \langle Q,\ q_0,\ \mathit{Mem},\ \rightarrow,\ \mathit{bad}\rangle$$

where Q is a finite control-state set, q_0 is the initial state, Mem is the monitor memory (defined below), $\rightarrow$ is the transition relation, and $\mathit{bad} \subseteq Q \times \mathit{Mem}$ is the set of bad configurations. Below, we

first define the monitor memory and then the synthesis function GenMon to translate each conjunct into a local monitor.

Monitor memory. The memory consists of two components. The first records whether each past formula $<\phi_H$ occurring in the safety conjuncts of $\mathsf{NF}(\Pi)$ has been satisfied at some point in the past (*historical memory*). The second records the set of active obligations and a violation flag for each bounded-response conjunct.

Formally, let $\Phi_<$ denote the set of past formulas $<\phi_H$ occurring in $\mathsf{NF}(\Pi)$. The *historical memory* is a function $\mathcal{H}\colon \Phi_< \to \{\mathsf{ok}, \mathsf{pending}\}$ where $\mathcal{H}(<\phi_H) = \mathsf{ok}$ means that $<\phi_H$ has been satisfied in the past, whereas $\mathcal{H}(<\phi_H) = \mathsf{pending}$ means that it is has not. The historical memory is monotone: once an entry becomes ok, it remains ok on all subsequent steps of the monitor.

For each bounded-response conjunct $P_1^j F^{\leq \Delta_j} P_2^j$, the monitor maintains a *clause state* (I_j, f_j). The set I_j collects the currently *active instances*: each instance (d, ρ) records the absolute deadline d and the environment ρ captured when P_1^j matched. The flag $f_j \in \{\mathsf{ok}, \mathsf{violated}\}$ is set to violated when at least one instance expires without the required response, and remains ok otherwise.

The *monitor memory* is then a pair $m = (\mathcal{H}, \mathcal{J})$, where $\mathcal{H}$ is the historical memory and $\mathcal{J}$ maps each bounded-response conjunct $j \in J$ to a clause state (I_j, f_j). We denote by $m_{\mathcal{H}}$ and $m_{\mathcal{J}}$ the first and the second element of m, respectively.

Safety automaton. For each safety conjunct $\neg\psi_i$ in $\mathsf{NF}(\Pi)$, the synthesis function GenMon returns a safety monitor automaton $A_i^s = \langle Q^s, q_0^s, \mathit{Mem}, \rightarrow_i^s, \mathit{bad}^s \rangle$ where:

$$Q^s = \{\mathsf{ok}, \mathsf{violated}\}, \quad q_0^s = \mathsf{ok}, \quad \mathit{bad}^s = \{(\mathsf{violated}, m) \mid m \in \mathit{Mem}\}.$$

Note that the violation condition depends solely on the control state: A_i^s is in violation if and only if its state is violated, regardless of the memory contents. The transition relation $\rightarrow_i^s$ is defined by the rules S-* at the top of Figure 2. In the rules, we use an *evaluation function* $\mathcal{E}[\![\psi_i]\!]^{\mathcal{H}}(e) \in K_3$ that evaluates a safety conjunct ψ_i on a single event e using the current historical memory $\mathcal{H}$ to resolve past sub-formulas. For $\psi_i = P$, evaluation reduces to the evaluation of the predicate P on e. For $\psi_i = <\phi_H \wedge P$, we first resolve the past condition by looking it up in $\mathcal{H}$: if $\mathcal{H}(<\phi_H) = \mathsf{pending}$, the evaluation results in *false* regardless of P; otherwise the evaluation results in the evaluation of P on e. Rule S-Viol fires when ψ_i evaluates to *true* on the event e under the current historical memory $m_{\mathcal{H}}$: in this case, the automaton transitions to the violated state. Rule S-Ok covers the cases *false* and *na*, both of which leave the automaton in the ok state: *na* means the current event is irrelevant to ψ_i and does not constitute a violation. Rule S-Absorb makes violated an absorbing state: once a safety violation is detected, it is irrevocable, mirroring the semantics of $G(\neg\psi_i)$.

Bounded-response automaton. For each bounded-response conjunct $P_1^j F^{\leq \Delta_j} P_2^j$, the synthesis function GenMon constructs a bounded-response automaton $A_j^r = \langle Q^r, q_0^r, \mathit{Mem}, \rightarrow_j^r, \mathit{bad}^r \rangle$ where:

$$Q^r = \{\mathsf{ok}, \mathsf{violated}\}, \quad q_0^r = \mathsf{ok}, \quad \mathit{bad}^r = \{(\mathsf{violated}, m) \mid m \in \mathit{Mem}\}.$$

The automaton tracks the clause state (I_j, f_j) in the shared memory. For each event e, the automaton: (*i*) checks whether any active instance has expired without satisfaction; (*ii*) removes from I_j all instances that are expired or satisfied by e; and (*iii*) adds a new instance if P_1^j is triggered by e, recording the deadline $e.t + \Delta_j$ and the environment ρ' captured at activation. These steps are performed by the update function $U((I_j, f_j), e)$.

The transition relation $\rightarrow_j^r$ is defined by the rules R-* in Figure 2, where m' is the memory m updated with the new clause state. Rule R-Ok fires when the memory update produces the violation flag ok: *no* active instance has expired without satisfaction, so the automaton remains in ok and the clause state is updated to (I', ok), reflecting the removal of satisfied instances and the possible addition of a new one. Rule R-Viol fires when the update produces the violation flag violated: *some* active instance has expired without satisfaction, so the automaton transitions to the violated state and the clause state is updated to $(I', \mathsf{violated})$. Rule R-Absorb makes the violated state absorbing: once a deadline violation is detected, the memory is left unchanged and the violation is irrevocable.

Monitor automaton. Given a normal form formula $\mathsf{NF}(\Pi)$, the synthesis function GenMon constructs the global monitor as the parallel composition of one local automaton for each conjunct:

$$\mathcal{A} \;=\; \left(\|_{i \in I} A_i^s\right) \,\|\, \left(\|_{j \in J} A_j^r\right)$$

where the set of control states is $Q = \prod_{i \in I} Q^s \times \prod_{j \in J} Q^r$, the initial state q_0 is given componentwise by $(q_0)_i = q_0^s$ for every $i \in I$ and $(q_0)_j = q_0^r$ for every $j \in J$, the initial monitor memory is $m = (\mathcal{H}, \mathcal{J})$ such that $\mathcal{H}$ maps every entry to pending and $\mathcal{J}$ maps every j to the clause state $(\emptyset, \mathsf{ok})$. A global configuration (q, m) is *bad* if and only if at least one local automaton is in violated:

$$\mathit{bad} = \{(q, m) \mid \exists\, i \in I.\ q_i = \mathsf{violated} \;\vee\; \exists\, j \in J.\ q_j = \mathsf{violated}\}.$$

On each event e, the monitor transition performs two steps. First, the historical memory $\mathcal{H}$ is updated to $\mathcal{H}'$: for each past formula $<\phi_H$, the corresponding entry is set to ok if ϕ_H holds on e, and left unchanged otherwise. This update is performed before any local automaton fires, so that all automata read the same updated $\mathcal{H}'$. Second, all local automata fire independently: for every $i \in I$ the automaton A_i^s performs a $\rightarrow_i^s$ transition, and for every $j \in J$ the automaton A_j^r performs a $\rightarrow_j^r$ transition, on the same updated memory. Since safety automata do not modify any clause state and the clause states of distinct bounded-response automata are disjoint, all updates are independent. The formal definition of the global monitor behavior is in the extended version [30].

After each event e, the monitor applies the enforcement function Enf to the updated configuration (q', m'):

$$\mathsf{Enf}(q', m') = \begin{cases} \mathit{allow} & \text{if } (q', m') \notin \mathit{bad} \\ A & \text{otherwise} \end{cases}$$

where $A \in \{\textsc{Alert}, \textsc{Deny}, \textsc{Kill}\}$ is the declared action in the DSL program. Recall that for a well-typed program, the type system guarantees statically that A respects the nature of the event e.

Note that the resulting monitor is deterministic and has size linear in the size of the formula and historical-predicate declarations; both properties are proved in the extended version [30]. These two properties are critical to achieve an effective eBPF implementation.

Correctness. The following theorems ensure the correctness of the monitor synthesis function GenMon. *Soundness* guarantees

$$\textsc{S-Viol}\ \frac{\mathcal{E}[\![\psi_i]\!]^{m_{\mathcal{H}}}(e) = \mathit{true}}{(\mathsf{ok}, m) \xrightarrow{e}{}^{s}_{i} (\mathsf{violated}, m)} \qquad \textsc{S-Ok}\ \frac{\mathcal{E}[\![\psi_i]\!]^{m_{\mathcal{H}}}(e) \neq \mathit{true}}{(\mathsf{ok}, m) \xrightarrow{e}{}^{s}_{i} (\mathsf{ok}, m)} \qquad \textsc{S-Absorb}\ \frac{}{(\mathsf{violated}, m) \xrightarrow{e}{}^{s}_{i} (\mathsf{violated}, m)}$$

$$\textsc{R-Ok}\ \frac{U(m_{\mathcal{J}}(j), e) = (I', \mathsf{ok}) \quad m'_{\mathcal{J}} = m_{\mathcal{J}}[j \mapsto (I', \mathsf{ok})]}{(\mathsf{ok}, m) \xrightarrow{e}{}^{r}_{j} (\mathsf{ok}, m')} \qquad \textsc{R-Viol}\ \frac{U(m_{\mathcal{J}}(j), e) = (I', \mathsf{violated}) \quad m'_{\mathcal{J}} = m_{\mathcal{J}}[j \mapsto (I', \mathsf{violated})]}{(\mathsf{ok}, m) \xrightarrow{e}{}^{r}_{j} (\mathsf{violated}, m')}$$

$$\textsc{R-Absorb}\ \frac{}{(\mathsf{violated}, m) \xrightarrow{e}{}^{r}_{j} (\mathsf{violated}, m)}$$

Figure 2: Rules defining the transition relations $\rightarrow^s_i$ and $\rightarrow^r_j$ of the local automata A^s_i and A^r_j, for every $i \in I$ and $j \in J$.

that the monitor never raises false alarms, whereas *detection completeness* guarantees that every semantic violation is eventually detected. Together, they ensure that the monitor is faithful to the formal semantics of $\mathsf{NF}(\Pi)$:

Theorem 6.2 (Soundness). *For every DSL program* Π *and finite trace* σ_k*, if* $\mathcal{A} = \mathsf{GenMon}(\mathsf{NF}(\Pi))$ *reaches a bad configuration after reading* σ_k*, then* $[\![\Pi]\!](\sigma_k) = \mathsf{violated}$.

Theorem 6.3 (Completeness). *For every DSL program* Π *and finite trace* σ_k*, if* $[\![\Pi]\!](\sigma_k) = \mathsf{violated}$*, then* $\mathcal{A} = \mathsf{GenMon}(\mathsf{NF}(\Pi))$ *reaches a bad configuration on some prefix* $\sigma_{k'}$ *with* $k' \leq k$*, provided that* (i) *every event relevant to the policy is delivered to the monitor and* (ii) *at least one event is delivered after every deadline expiration.*

Note that detection completeness requires two additional assumptions: first, that every event relevant to the policy is delivered to the monitor without loss; second, that for bounded-response clauses, at least one event is delivered after every deadline expiration, so that the monitor has the opportunity to detect the timeout. The first assumption holds when the eBPF hook is attached before any relevant event can occur; the second is satisfied in practice by the continuous stream of kernel events.

7 Compilation to eBPF

This section describes how a monitor $\mathcal{A}$ produced in Section 6 is compiled into concrete eBPF programs that enforce the policy inside the Linux kernel. We give an overview of the compilation scheme (Section 7.1), show how the monitor state is realized via BPF maps (Section 7.2), present the per-event processing logic (Section 7.3), and then describe the runtime support that completes the enforcement loop (Section 7.4).

7.1 Compilation Overview

Each eBPF program is attached to exactly one kernel hook and can only intercept events raised at that hook. A policy, however, can refer to events from multiple hooks: a bounded-response clause $P_1^j \, F^{\leq \Delta_j} \, P_2^j$ may trigger on one hook and expect a response on another, and a safety conjunct $\neg(<\!\phi_H \wedge P)$ may record a historical flag on one hook and check it on another. The BPFence compiler thus translates a policy Π into one BPF program per hook mentioned in the policy. Since these programs collectively implement the single monitor $\mathcal{A}$, they must share the monitor memory $(\mathcal{H}, \mathcal{J})$ so that the state written by one program is visible to the others.

Concretely, the compilation of a program Π is driven by its normal form $\mathsf{NF}(\Pi)$ (Section 6.1), which splits the policy into safety conjuncts $\neg\psi_i$ and bounded-response conjuncts $P_1^j \, F^{\leq \Delta_j} \, P_2^j$. Each conjunct maps directly to a block of generated code: safety conjuncts become `if–goto` checks implementing the rules S-Viol/S-Ok of Figure 2, while bounded-response conjuncts become lookups and updates on the monitor state implementing the U function and the rules R-Viol/R-Ok. Since the number of clauses is fixed, all conjuncts are *unrolled at compile time*: the generated code is a straight-line sequence with no loops, which guarantees that every program passes the eBPF verifier's termination check.

Additionally, the generated programs run in the kernel and capture events from all processes simultaneously, making the monitor inherently system-wide. However, policies track the behavior of individual entities independently, e.g., a taint flag set by one process must not propagate to another. A policy defines the *unit of monitoring* by declaring a *scope* (`pid`, `tgid`, `cgroup`, or `namespace`): two events belong to the same monitored entity if they share the same scope value. To achieve this grouping, on each intercepted event, the generated code extracts an *execution key* κ using an appropriate BPF helper (e.g., `bpf_get_current_pid_tgid()` for per-process scope, `bpf_get_current_cgroup_id()` for per-container scope). The execution key κ partitions the monitor memory: each monitored entity maintains its own independent copy $(\mathcal{H}_\kappa, \mathcal{J}_\kappa)$. However, kernel identifiers can be recycled, for example, once a process exits, its `pid` can be reassigned to an unrelated process, which would then inherit the state accumulated under the same key. For `pid`-scoped policies, the generated code therefore invalidates the historical memory of a key when the process it identifies exits, so that the monitor memory follows the lifetime of that process and not the reuse of its identifier. Below, we detail how this per-entity state is concretely stored and shared across programs.

7.2 Representing the Monitor State

eBPF programs have no heap and no writable global variables, so the only mechanism for keeping persistent state is the *BPF map*. The generated BPFence programs use statically sized BPF maps to maintain the per-entity monitor memory $(\mathcal{H}_\kappa, \mathcal{J}_\kappa)$.

The historical memory $\mathcal{H}_\kappa$ is implemented by a hash table keyed by a pair (κ, h), where h is a compile-time numeric identifier assigned to each historical predicate $<\!\phi_H$. When the generated program evaluates ϕ_H to true, it sets the corresponding entry to 1; the

entry is never reset while the monitored entity exists, preserving the monotonicity of $\mathcal{H}$.

The active instance set I_j of each bounded-response clause is stored in a map keyed by a pair (κ, j), where each entry holds a deadline (a timestamp in nanoseconds) and the correlation environment ρ' captured at activation of the instance, encoded as a fixed-size byte array. When the map is full, an LRU policy discards the least-recently-used entry; as long as all the active instances fit in it, the generated program faithfully realizes the formal semantics of I_j of Section 6.2, which assumes an unbounded I_j.

The memory space required by a monitored entity can be determined by analyzing the policy: since the keys have the form (κ, h) and (κ, j), an entity occupies at most $|\Phi_<|$ entries of the first map and $|J|$ of the second.

Finally, the monitor is system-wide and its programs run on every core, so the state of one entity may be updated concurrently. The generated code relies on the atomic operations the kernel provides for eBPF maps, and makes atomic those updates that read a value and write it back.

7.3 Per-event Processing

Figure 3 shows the pseudocode of a generated BPF program. For each intercepted event e, the program first extracts the execution key κ from the event context and, if the policy restricts enforcement to a target set, short-circuits to `allow` for entities outside it (lines 3–5). The code then reads the relevant event fields from the hook arguments into a local instance of a suitable structure (lines 7–9).

The core of the program implements the monitor behavior as described in Section 6.2. The program first evaluates all historical predicates: for each $<\phi_H$, if ϕ_H holds on the current event, the corresponding entry in $\mathcal{H}_\kappa$ is set to 1 (lines 11–14). All historical updates are performed *before* any safety or response check, so that subsequent evaluations read an updated $\mathcal{H}_\kappa$. Next, for each safety conjunct $\neg\psi_i$, the program evaluates $[\![\psi_i]\!]^{\mathcal{H}_\kappa}$ on the current event. If the result is *true*, the event violates $\neg\psi_i$, and the program jumps immediately to the enforcement handler (lines 16–19). Finally, for each bounded-response clause $P_1^j \, F^{\leq \Delta_j} \, P_2^j$, the program updates the active instance set I_j following a three-phase logic: it detects expired instances, removes satisfied ones, and activates new ones when P_1^j matches (lines 21–30).

When a violation is detected, the BPF program applies the action declared in the policy: Deny returns `-EPERM` from the LSM hook, blocking the operation before it completes; Kill invokes `bpf_send_signal(SIGKILL)`; Alert emits the violation event to user space without blocking. A bounded-response violation occurs when a deadline expires without the required response. Since the BPF program only executes when an event arrives, the runtime daemon provides a dedicated *timeout scanner* that periodically checks for expired deadlines.

7.4 Runtime Support

A user-space daemon loads the generated BPF programs and attaches them to their corresponding kernel hooks. When a policy spans multiple hooks, the programs must share the monitor memory $(\mathcal{H}_\kappa, \mathcal{J}_\kappa)$ introduced in Section 7.2. The daemon achieves this sharing by loading programs sequentially: after the first program is loaded, the daemon binds each subsequent program's map slots to the maps already created by the first, so that all programs in the policy read and write the same BPF maps. This mechanism enables cross-hook correlation of variables and values.

```
1  SEC("lsm/hook_name")
2  int BPF_PROG(bpfence_hook, ...) {
3    // 1. Key extraction and target filter
4    u64 kappa = bpf_get_current_pid_tgid();
5    if (!in_target_set(kappa)) return 0;
6
7    // 2. Event extraction
8    struct bpfence_ev ev;
9    extract_fields(&ev, ctx);
10
11   // 3. Historical-flag update UpdH
12   for each <φ_H ∈ H:    // unrolled during compilation
13     if (φ_H holds on ev)
14       history_map[kappa, h] = 1;
15
16   // 4. Safety-clause checks ¬ψ_i
17   for each ψ_i ∈ S:          // unrolled during compilation
18     if ([[ψ_i]]^{H_κ}(ev))
19       goto violation;
20
21   // 5. Bounded-response update P_1^j F^{≤Δ_j} P_2^j
22   for each clause j in R:     // unrolled during compilation
23     rv = response_map[kappa, j];
24     if (rv && expired(rv))    // (i) detect
25       goto violation;
26     if (rv && P_2^j matches ev) // (ii) satisfy
27       delete response_map[kappa, j];
28     if (P_1^j matches ev)       // (iii) activate
29       response_map[kappa, j] =
30         { deadline: now + Δ_j, env: ρ' };
31
32   return 0; // allow
33 violation:
34   emit_alert(kappa, policy, clause, ...);
35   // Deny: return -EPERM;
36   // Kill: bpf_send_signal(SIGKILL);
37 }
```

Figure 3: Pseudocode of a generated BPF program.

Once all programs are attached, the daemon performs two tasks. First, it polls a shared ring buffer where the BPF programs write violation events (each carrying the policy identifier, execution key κ, violated clause, and contextual information) and dispatches them to a configurable handler. Second, it runs a *timeout scanner* that periodically checks the response map for expired deadlines, detecting bounded-response violations.

8 Implementation and Evaluation

BPFence has been implemented as two open-source tools available online [29]: a compiler that translates policies into loadable eBPF programs, and a daemon that loads the eBPF programs, attaches them to their corresponding kernel hooks, and manages the violation notification. Section 8.1 briefly presents our implementation, whereas Section 8.2 presents the experimental evaluation.

8.1 Implementation

The implementation consists of an OCaml compiler (~7K LoC) and a C runtime daemon (~2K LoC). The compiler translates a `.bpfence` source file into C files, which are then compiled to `.bpf.o` objects

via `clang` and loaded by the daemon. In addition to the tasks described in Section 7.4, the daemon dispatches violation events and manages policy updates. Violation events are processed by a configurable handler chain that ships with three built-in handlers: `log` (structured logging to file or stdout), `alert` (forwarding to syslog), and `exec` (execution of a shell command to handle the event). Policy updates can be applied at runtime, without interrupting enforcement, upon receipt of a suitable Linux signal.

A subtle but critical invariant of the daemon concerns the order in which the compiled programs are started. eBPF programs attached to LSM hooks start intercepting system calls the moment they are linked to their hook, so if the daemon attached a `deny` or `kill` policy before configuring its set of target processes, the policy would momentarily apply to every process on the host. To avoid this race, the daemon enforces a three-phase initialization: it first *loads* each BPF object into the kernel, creating the maps, then *populates the target map* with the PIDs or cgroup identifiers, and only at the end *attaches* the programs to their hooks. The same invariant is preserved across policy reloads: new programs are loaded, and their target map is transferred from the old instance before the old programs are detached, so there is no time window in which an unconfigured monitor is live.

The user can control how much space is allocated to the monitor memory, either directly or through one of four available profiles. The daemon applies this choice before the loading phase and refuses to start when the requested capacity is below the requirements recorded by the compiler in the generated object.

The compiler and the daemon also implement two countermeasures against techniques an attacker may use to evade a policy. These countermeasures are a first step towards strengthening the threat model of Section 4, which assumes an attacker that does not actively evade the deployed policies. The first targets evasion through `fork`: a policy scoped to `pid` correlates events from a single process, so an attacker who splits a kill chain across a `fork` would present the monitor with two unrelated processes. The compiler, therefore, emits additional programs that propagate a process's history to its children and reclaim it when the process exits. The second targets evasion through file aliasing: a path is one of the names of a file rather than the file itself, so symbolic and hard links, bind mounts, and renames give an attacker distinct names for the same file. When a policy mentions literal paths, the compiler keys the match on the file's identity and keeps it up to date as files are created, aliased, or removed.

Beyond the core constructs of Section 5, the implementation supports further language features to simplify policy creation. It provides four additional temporal operators: *bounded sequences* (`sequence` P_1 `then within` Δ_1 P_2 `...`) for multi-step actions with per-step deadlines, *bounded until* (`while` P_1 `then within` Δ P_2) for asserting that a condition P_2 must occur within Δ once P_1 holds, *bounded count* (`count` $E \geq N$ `within` Δ) for detecting N or more occurrences of an event in a sliding window, and *bounded safety* (`forbid P when G within` Δ). Bounded sequences desugar into independent bounded-response clauses of the base temporal logic. Bounded until, count, and bounded safety are compiled via dedicated schemes. To promote reuse, the DSL supports *parameterized templates* and *policy inheritance*. A policy can declare parameters that are bound at instantiation time via the `use` construct, allowing a single template to be reused across different policies. A policy can also *extend* another, inheriting its clauses and optionally adding or specializing them. Both mechanisms are resolved at desugaring time and do not affect the formal semantics of the language. In addition, BPFence ships a standard library of event schemas covering common Linux subsystems (file operations, process lifecycle, network connections, raw kernel events) and reusable policy templates, e.g., Chinese Wall, sandboxing, data-loss prevention, and rate limiting. More details on all additional constructs, their compilation, and the standard library are in the extended version [30].

8.2 Experimental Evaluation

Our experimental evaluation addresses three research questions:

RQ1. What overhead does a BPFence monitor impose on hooked system calls, and how does it scale with policy complexity?

RQ2. Can BPFence monitors block real attack patterns while avoiding false positives on benign workloads?

RQ3. What policies can BPFence easily express and enforce that existing tools cannot?

All experiments ran on a laptop with an Intel Core i7-1195G7 (4 cores, 8 threads, 2.90 GHz base clock), 32 GB RAM, Ubuntu 24.04, Linux kernel 6.8.0. All runs were performed with the CPU frequency pinned to the base clock (governor `performance`, turbo disabled).

8.2.1 RQ1: Runtime Overhead. We evaluated BPFence on five micro-benchmarks (B-1 to B-5) and one macro-benchmark (B-6).

Micro-benchmarks. The micro-benchmarks use a synthetic workload consisting of a C program that invokes `open`/`close` syscalls N=100,000 times. The monitor execution time is measured via the kernel's built-in profiling infrastructure, which maintains, for each loaded BPF program, a cumulative invocation counter (`run_cnt`) and a cumulative execution-time counter (`run_time_ns`). By reading both counters before and after the N syscalls, we compute the per-invocation BPF cost as Δ`run_time_ns`/Δ`run_cnt`, where Δ denotes the difference between the two readings, obtaining a direct kernel-level measurement without user-space timing noise. We repeat this process for R=10 trials and report mean ± standard deviation. The overhead is computed as BPF cost/baseline cost × 100, where the baseline cost (1318 ns/op) is measured via `clock_gettime` without any BPF program loaded. The baseline was measured again after each run: it drifted by less than 1%, and the CPU never throttled. We generate synthetic policies attached to the `security_file_open` LSM hook. Each policy defines an event `open` with a string field (the file path), extracted via the kernel helper `bpf_d_path`.

Table 1 reports the results of these benchmarks. The first two measure how the cost scales with the size of the policy. In benchmark B-1, we vary the number of `forbid` clauses (C) and show that the cost is *nearly constant*: from C=1 to C=20, the monitor execution time grows from 240 to 295 ns (~3 ns per additional clause). Benchmark B-2 varies the number of historical predicates (H) in a policy and shows a *linear* cost: each predicate requires one BPF hash-map lookup, contributing ~19 ns (from 260 ns at H=1 to 432 ns at H=10). The other three benchmarks allow us to understand where the fixed cost comes from. Benchmark B-4 repeats the measurement of B-1 at C=1 with an event taking an integer parameter, eliminating

Table 1: Monitor execution time per `open`/`close` syscall (baseline without monitor: 1318 ns/op).

	Configuration	BPF (ns/op)	Overhead
B-1	C=1 clause	240 ± 2	18.2%
B-1	C=5 clauses	249 ± 2	18.9%
B-1	C=20 clauses	295 ± 2	22.4%
B-2	H=1 predicate	260 ± 1	19.7%
B-2	H=5 predicates	341 ± 2	25.9%
B-2	H=10 predicates	432 ± 2	32.8%
B-3	K=1−100 targets	61 ± 1	4.6%
B-4	No path extraction	71 ± 1	5.4%
B-5	Non-targeted process	61 ± 1	4.6%

Table 2: Case studies: attack surface reduction. *Outcome*: how the attack trigger is handled. In all cases the benign trigger completes without interference.

	Scenario	Motivation	Outcome
CS-1	Post-exploit. sandbox	Log4Shell [41]	Killed
CS-2	Resource exhaustion	Fork-bomb [38]	Alerted
CS-3	Container isol. (BLP)	CVE-2024-21626 [43]	Denied
CS-4	Data exfiltration	Equifax 2017 [46]	Denied
CS-5	3-step SSH kill chain	MITRE T1021 [35]	Killed
CS-6	Persistence + FD leak	Persistence [37]	Killed
CS-7	Clark-Wilson (E2)	Dirty Pipe [42]	Denied

the call to `bpf_d_path` and the checks that accompany it: the cost drops to 71 ns, showing that resolving the file path accounts for ∼169 ns, about 70% of the per-invocation cost. Benchmarks B-3 and B-5 measure a process that the policy does not target: the generated code checks the target map and returns *before* extracting the event, so neither reaches `bpf_d_path`. Benchmark B-3 varies the number of scoped targets (K=1 to 100) on the same path-based schema as B-1 at C=1. It confirms that the target lookup is constant in K: all configurations yield 61 ± 1 ns. Benchmark B-5 repeats the measurement on the integer schema of B-4, paying only 61 ns. In all benchmarks except B-3 and B-5, we use global scope, which is a worst case because every process pays the hook cost.

Macro-benchmark. To assess the impact on an application workload, we deploy a non-trivial policy on nginx serving a 1 KB static file. We generate sustained HTTP load using `wrk` with 2 threads and 100 connections, running each trial for 30 s and repeating for R=10 trials. The load generator and the server are pinned to disjoint physical cores to prevent the measurement from being dominated by contention between them. The considered policy combines the post-exploitation sandbox (CS-1) with SSH lateral-movement detection (CS-5), both described in the next subsection. During normal HTTP traffic, nginx never spawns a shell or reads SSH keys, so all historical predicates remain false throughout the test; the monitor still executes on every hooked syscall, but every guard evaluates to false and no violation is triggered. This represents the common case in production, where the monitor is active, but no attack is in progress. Without any policy, nginx sustains 155,798 ± 666 req/s; with the policy loaded, throughput drops to 147,223 ± 436 req/s, a reduction of 5.5%. Using the same kernel counters as above, the per-invocation BPF cost is 149 ns, consistent with the micro-benchmark results for policies of comparable complexity.

Note that the per-invocation cost is not specific to BPFence, as every BPF-LSM monitor pays this cost whenever a hooked operation occurs. In fact, the overhead of BPFence is similar to that of other monitors that rely on LSM hooks. For example, BPFContain [16] reports 12.69–14.31% on Apache and up to 16.76% on micro-benchmarks, and BPFBox [17] 1.0–15.0% across the Phoronix suite. For the tools of Section 8.2.3, an independent study [21] reports 9.79% for Tetragon and 9.96% for Falco on a per-execution micro-benchmark, while their vendors report lower overheads. Our results, 5.5% on nginx and 4.6–5.4% per hooked syscall when the hook does not resolve a file path (B-3–B-5), sit at the low end of this range. Our comparison is indicative only, as the overheads above were measured across different workloads and machines.

8.2.2 RQ2: Attack Surface Reduction. We evaluated the effectiveness of BPFence on the seven case studies (CS-1 to CS-7) of Table 2. Each policy targets a post-exploitation syscall pattern associated with a documented attack or CVE (third column); the original exploit is outside the scope of the policy. For each case study, we execute two trigger programs: an *attack trigger* that reproduces the syscall pattern, and a *benign trigger* that performs similar but legitimate operations.

In all cases, the attack trigger is correctly blocked or alerted, whereas the benign trigger completes without interference (zero false negatives and zero false positives). In CS-1 (*post-exploitation sandbox*), the policy detects when a server process spawns a shell and then prevents that shell from executing further binaries, opening network connections, or writing configuration files. This behavioral pattern is common in post-exploitation steps. The policy relies on a historical predicate to condition the enforcement on a previous `exec`. In CS-2 (*resource exhaustion*), the policy raises an alert when a process clones itself more than 100 times within a 1 s time window, mitigating fork-bomb attacks, or opens more than 50 new connections within a 10 s sliding time window, mitigating connection-flooding patterns. CS-3 (*container isolation*) enforces a Bell–LaPadula no-read-up / no-write-down policy across container cgroups, preventing information flows from a high-sensitivity container to a low one via the host file system. In CS-4 (*data exfiltration*), the policy detects when a process reads a sensitive file and *subsequently* opens a network connection; it correlates two events on different LSM hooks via cross-hook binding. Case study CS-5 (*SSH lateral movement*) was already presented in Section 3: the policy detects a three-step kill chain in which the attacker reads an SSH private key, opens an outbound connection on port 22, and finally executes a command. The process is killed on the final `exec`. In CS-6 (*persistence containment + FD-leak watchdog*), the policy combines two clauses with different enforcement semantics: an inline safety check kills the process if it attempts post-exploitation persistence operations, e.g., `exec(/usr/bin/crontab)`, or write to `/etc/passwd`, after a shell has been spawned, while a liveness obligation alerts when a file descriptor is opened but not closed within

30 s, mitigating resource-leak patterns. Case study CS-7 (*Clark–Wilson integrity*) enforces the E2 rule of the Clark–Wilson model: protected files (e.g., `/etc/passwd`, setuid binaries) can be modified only by authorized processes. Any write from an unauthorized process is denied, blocking attacks such as Dirty Pipe, in which an unprivileged process bypasses standard permissions to overwrite a protected file. The policy uses a negated history guard of the form "forbid write unless an authorized program was executed".

Additionally, the artifact includes six further case studies (CS-8 to CS-13), in which the attack trigger is a public proof-of-concept exploit for a documented (patched upstream) kernel vulnerability, and the policy forbids the system call patterns that may lead to the attack, e.g., the use of exotic crypto sockets. Since a policy describes the attack pattern rather than a specific bug, it may intercept multiple vulnerabilities. For example, the policy of CS-8 covers *copy_fail* (CVE-2026-31431) and *Fragnesia* (CVE-2026-46300), which affect different kernel subsystems but share the same behavioral pattern: the setup of an exotic crypto socket followed by the execution of a setuid binary. These case studies show that BPFence may also serve as a containment layer for this class of bugs when the patch is not yet deployed.

8.2.3 RQ3: Comparison with Existing Tools. To compare BPFence with production tools, we took four real rules from Falco [13] and Tetragon [8], two well-known open-source projects, and re-expressed each of them as a BPFence policy. The two Falco rules are *Read sensitive file untrusted* [14] and *Write below binary directories* [15]: both belong to the default ruleset shipped with every Falco installation and are among the most cited examples in the project's documentation and tutorials, targeting credential exfiltration and binary tampering, respectively. The two Tetragon policies are `monitor-kernel-modules` [9] and `sys_mount` [10], both taken from the examples in the upstream repository, and cover two attack surfaces that the documentation prominently advertises as use cases: kernel-rootkit installation and mount-based container escape. For each rule, we provide a *direct port policy* that reproduces the upstream behavior, and one or more *companion policies* that capture behavior the upstream rule cannot express. The policies and their attack and benign triggers are in the online repository.

Falco's *Read sensitive file untrusted* (∼ 35 LoC) matches reads of credential, SSH-key, TLS-key, and Kubernetes-secret files performed outside an allowlist of trusted readers, and raises an alert. Our direct port policy (∼ 16 LoC) keeps the same set of paths, but instead of a tracepoint uses the LSM hook `security_file_open`, so the same event can be synchronously denied. A companion policy denies the same reads only *after* the offending process has spawned a shell, capturing a "shell exec → credential read" post-exploitation kill chain [39]. Falco cannot express this stateful correlation, since its rule language is single-event and stateless. For *Write below binary directories* (∼ 19 LoC), we follow the same approach: a direct port (∼ 16 LoC) denies writes to `/bin`, `/sbin`, `/usr/bin`, `/usr/lib`, and similar locations, while the companion policy kills the offending process when it attempts these writes after a shell spawn.

Tetragon's `monitor-kernel-modules` policy (∼ 50 LoC) relies on the `security_kernel_module_request` and `do_init_module` hooks: it logs every attempt to load a kernel module but does not block it. Our direct port policy (∼ 7 LoC) relies on the same hooks but upgrades the action to a synchronous LSM deny, so the module is rejected before it is linked into the kernel. We provide two companion policies that perform cross-hook correlations that a single Tetragon policy cannot express. The first synchronously blocks the `bpf()` syscall when invoked with the argument `BPF_PROG_LOAD` for loading a BPF program or with the argument `BPF_MAP_CREATE` for creating a BPF map; the second policy blocks any `bpf()` syscall *after* the same process has loaded a kernel module, capturing in a single policy a kill chain in which an attacker first installs a kernel-mode rootkit [34] and then leverages BPF for further in-kernel persistence. Tetragon's `sys_mount` policy (∼ 12 LoC) traces every `mount` syscall and reports its source and target without filtering or blocking. Our direct port policy (∼ 11 LoC) attaches to the LSM hook `security_sb_mount` and synchronously denies mounts of the filesystem types most commonly abused for container escape, such as `proc` and `sysfs`. The two companion policies extend it, considering multiple hooks: the first denies any `bpf()` syscall after a suspicious mount has been observed, capturing a kill chain in which an attacker first escapes the container by mounting host filesystems [40] and then uses BPF to consolidate the breakout; the second denies every outbound `connect()` after the process has spawned a shell, capturing post-escape lateral movement [35].

We tested all four ports by providing an attack and a benign trigger for each policy, following the same approach as in RQ2: in every case the attack trigger is blocked, while the benign trigger completes without interference. The following observations emerge from the comparison. BPFence policies are typically shorter. Falco rules are intrinsically alert-only and operate on a single hook, because they observe syscalls via tracepoints *after* the kernel has executed them; re-expressing them in BPFence turns detection into synchronous prevention with no change to the matching logic, simply by attaching the same predicate to the corresponding LSM hook. Tetragon supports synchronous enforcement, but its policies are single-hook by construction, so cross-event correlations such as "module load *then* BPF program load" or "shell exec *then* connect" cannot be expressed in a single policy, whereas BPFence's historical predicates and `when` construct express these correlations directly inside one policy. The comparison also revealed two matching primitives that Tetragon offers and BPFence does not: testing whether a specific bit is set in a syscall flag argument, and filtering on the privileges of the calling process. We leave their implementation in BPFence for future work.

9 Related Work

We discuss the two lines of research to which our work is closely related: in-kernel security monitoring and runtime verification.

In-kernel security monitoring. Falco [13], Tetragon [8], and KubeArmor [28] are the industry references for eBPF-backed runtime security: they express policies as YAML rules whose semantics is defined only by the implementation, and they do not statically distinguish the hooks on which blocking is possible from those on which it is not. We compared BPFence against Falco and Tetragon in Section 8 (RQ3). The tool bpftrace [3] is complementary: a scripting language for kernel tracing focused on observability, with no LSM hook support and with write access restricted to error-injection

points. BPFBox [17] and BPFContain [16] compile process confinement policies to eBPF LSM programs, demonstrating the feasibility of the hook point but without temporal or stateful operators. Jia et al. [24] extend Seccomp-BPF with temporal specialization and counter-based checks at the system-call boundary; both are subsumed by BPFence's policy language. More recently, eBPF-PATROL [18] applies eBPF to runtime threat recognition in containerized environments, again without formal guarantees or a type discipline. The closest system in spirit is the Linux kernel's own runtime verification subsystem [11, 32], upstream since v6.0, which loads hand-drawn deterministic automata as kernel modules attached to tracepoints and reacts to violations with `printk` or `panic`. BPFence shares the same goal of being an online monitor inside the kernel, but offers a DSL for the specification of policies with a formal semantics and a type system that statically separates controllable from merely observable hooks, delivers monitors as eBPF programs rather than kernel modules, attaches to LSM hooks as well, and scopes enforcement per process, namespace, or cgroup.

Overall, none of these systems combines a policy language with formal semantics, a type discipline separating controllable from observable events, and provably correct compilation to in-kernel monitors, which is what BPFence contributes.

Runtime verification. BPFence brings runtime verification inside the kernel. The construction of finite-state monitors from temporal-logic specifications dates back to the work of Bauer et al. [6] for LTL and TLTL, and to the automata-based techniques of Havelund and Roşu [20]. On the theoretical side, Schneider [44] and Basin et al. [5] characterized the classes of policies enforceable by execution monitors; Aceto et al. [2] extended enforcement to first-order branching-time properties, and Hublet et al. [22, 23] developed proactive enforcement for metric first-order temporal logic together with scalable implementations. At the tool level, MonPoly [4] monitors metric first-order temporal properties over logs, JavaMOP [25] compiles parametric specifications into aspect-oriented instrumentation, and DejaVu [19] targets first-order past-time logic; decentralized variants such as RIARC [1] address the instrumentation of reactive components. All of these monitors rely on very expressive logics and run in user space, in the same address space as the monitored program, or offline over recorded traces; BPFence aims at in-kernel policy enforcement and realizes monitor synthesis under the constraints of the eBPF verifier: bounded memory, no unbounded loops, a fixed instruction budget. In-kernel enforcement requires trading off the expressiveness of the policy language with implementation constraints, as done in Section 7.

10 Conclusion

We have presented BPFence, a runtime-verification framework to enforce trace-based security policies inside the Linux kernel. BPFence offers a policy language with a formal semantics over execution traces and a type system that statically separates controllable from observable events. It then synthesizes, from every well-typed policy, a correct finite-state monitor, and compiles it to eBPF programs that run synchronously inside the kernel. We implemented a PoC that includes a compiler and a runtime daemon to manage the policy life-cycle. We have evaluated our implementation on seven case studies drawn from real-world attack patterns and on micro- and macro-benchmarks to show that stateful security policies can be enforced inside the kernel with an acceptable overhead. Finally, we have qualitatively evaluated the expressiveness of our policy language against upstream Falco and Tetragon rules, showing that our language can prevent more complex attack patterns than these widely adopted tools.

BPFence inherits two structural limitations from its target environment. First, synchronous enforcement is only possible on events mediated by an LSM hook: policies whose violating events occur only through tracepoints or kprobes can be *detected* but not *prevented*. Second, the eBPF execution model forbids dynamic allocation; as a consequence, the monitor state is stored in fixed-size maps, whose capacity is set when a policy is loaded and checked against the requirements computed at compile time.

There are several directions in which to extend the present work. On the *language* side, we plan to introduce richer predicates and inline aggregations, and lift correlation from kernel identities (`pid`, `tgid`, `cgroup`) to causal ones (process trees and data-flow taint): the propagation of history across `fork` (Section 8.1) is a first step, but attacks that spread through `execve` or across cooperating processes require the unit of monitoring itself to be defined by causality rather than by a declared scope. On the *formal* side, we are interested in mechanizing our soundness proof and in monitoring hyperproperties such as non-interference, which lie beyond plain trace properties. On the *deployment* side, we plan to explore distributed enforcement across a cluster of containers, automatic policy synthesis from high-level intents or from observed benign behavior, and integration with container orchestrators to tie the life-cycle of policies to that of the workloads they protect.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions, which helped improve the paper. This work has been partially supported by RDS PTR 25-27 CYBER 2.1 "Progetto Cybersecurity" WP3 LA 3.21 – CUP: D63C24001060001.

References

[1] Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfsdóttir. 2024. Runtime Instrumentation for Reactive Components. In *ECOOP (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:33. doi:10.4230/LIPIcs.ECOOP.2024.2

[2] Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. 2023. On First-Order Runtime Enforcement of Branching-Time Properties. *Acta Informatica* 60, 4 (2023), 385–451. doi:10.1007/s00236-023-00441-9

[3] Alastair Robertson. 2024. bpftrace: High-Level Tracing Language for Linux. https://bpftrace.org/. Accessed: April 20, 2026.

[4] David A. Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. 2011. MONPOLY: Monitoring Usage-Control Policies. In *RV (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 360–364. doi:10.1007/978-3-642-29860-8_27

[5] David A. Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zalinescu. 2013. Enforceable Security Policies Revisited. *ACM Trans. Inf. Syst. Secur.* 16, 1 (2013), 3. doi:10.1145/2487222.2487225

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4, Article 14 (2011), 64 pages. doi:10.1145/2000799.2000800

[7] Canonical Ltd. 2024. AppArmor: Application Armor. https://apparmor.net.

[8] Cilium Project. 2024. Tetragon: eBPF-based Security Observability and Runtime Enforcement. https://tetragon.io.

[9] Cilium Project. 2024. Tetragon `TracingPolicy`: `monitor-kernel-modules.yaml`. https://github.com/cilium/tetragon/blob/main/examples/tracingpolicy/host-changes/monitor-kernel-modules.yaml.

[10] Cilium Project. 2024. Tetragon `TracingPolicy`: `sys_mount.yaml`. https://github.com/cilium/tetragon/blob/main/examples/tracingpolicy/sys_mount.yaml.

[11] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. 2019. Efficient Formal Verification for the Linux Kernel. In *Software Engineering and Formal Methods: 17th International Conference, SEFM 2019, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 315–332. doi:10.1007/978-3-030-30446-1_17
[12] eBPF Foundation. 2023. eBPF: Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io.
[13] Falco Project. 2023. Falco: Cloud-Native Runtime Security. https://falco.org/docs/.
[14] Falco Project. 2024. Falco default rule: *Read sensitive file untrusted*. https://falco.org/docs/reference/rules/default-rules/#Read_sensitive_file_untrusted.
[15] Falco Project. 2024. Falco default rule: *Write below binary directories*. https://falco.org/docs/reference/rules/default-rules/#Write_below_binary_dir.
[16] William Findlay, David Barrera, and Anil Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. arXiv preprint arXiv:2102.06972.
[17] William Findlay, Anil Somayaji, and David Barrera. 2020. bpfbox: Simple Precise Process Confinement with eBPF. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*. Association for Computing Machinery, New York, NY, USA, 91–103. doi:10.1145/3411495.3421358
[18] Sangam Ghimire, Nirjal Bhurtel, Roshan Sahani, and Sudan Jha. 2025. eBPF-PATROL: Protective Agent for Threat Recognition and Overreach Limitation using eBPF in Containerized and Virtualized Environments. arXiv preprint arXiv:2511.18155.
[19] Klaus Havelund, Doron Peled, and Dogan Ulus. 2018. DejaVu: A Monitoring Tool for First-Order Temporal Logic. In *MT@CPSWeek*. IEEE, 12–13. doi:10.1109/MT-CPS.2018.00013
[20] Klaus Havelund and Grigore Rosu. 2004. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods Syst. Des.* 24, 2 (2004), 189–215.
[21] Jin Her, Jongseop Kim, Jinwoo Kim, and Seungsoo Lee. 2025. An In-Depth Analysis of eBPF-Based System Security Tools in Cloud-Native Environments. *IEEE Access* 13 (2025), 155588–155604. doi:10.1109/ACCESS.2025.3605432
[22] François Hublet, Leonardo Lima, David Basin, Srđan Krstić, and Dmitriy Traytel. 2024. Proactive Real-Time First-Order Enforcement. In *Computer Aided Verification (CAV)*. Springer Nature Switzerland, Cham, 156–181. doi:10.1007/978-3-031-65630-9_8
[23] François Hublet, Leonardo Lima, David Basin, Srđan Krstić, and Dmitriy Traytel. 2025. Scaling Up Proactive Enforcement. In *Computer Aided Verification (CAV)*. Springer Nature Switzerland, Cham, 370–392. doi:10.1007/978-3-031-98682-6_19
[24] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable System Call Security with eBPF. arXiv:2302.10366
[25] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE*. IEEE Computer Society, 1427–1430. doi:10.1109/ICSE.2012.6227231
[26] The kernel development community. 2012. Seccomp-BPF: A Sandboxing Mechanism for Linux. https://www.kernel.org/doc/html/v4.13/userspace-api/seccomp_filter.html.
[27] Stephen Cole Kleene. 1952. *Introduction to Metamathematics*. Ishi Press International.
[28] KubeArmor Project. 2024. KubeArmor: Cloud-Native Runtime Security Enforcement System. https://kubearmor.io.
[29] Letterio Galletta. 2026. Enforcement of In-Kernel Stateful Security Policies via eBPF (Artifact). https://doi.org/10.5281/zenodo.22332429. Original repository: https://github.com/lillo/bpfence.
[30] Letterio Galletta. 2026. Enforcement of In-Kernel Stateful Security Policies via eBPF (Extended Version). doi:10.5281/zenodo.22744507
[31] Linux kernel developers. 2024. Kernel Probes (Kprobes). Linux kernel documentation, https://www.kernel.org/doc/html/latest/trace/kprobes.html.
[32] Linux kernel developers. 2024. Runtime Verification. Linux kernel documentation, https://docs.kernel.org/trace/rv/runtime-verification.html.
[33] Linux kernel developers. 2024. Using the Linux Kernel Tracepoints. Linux kernel documentation, https://www.kernel.org/doc/html/latest/trace/tracepoints.html.
[34] MITRE ATT&CK. 2024. T1014: Rootkit. https://attack.mitre.org/techniques/T1014/. Accessed: April 20, 2026.
[35] MITRE ATT&CK. 2024. T1021: Remote Services. https://attack.mitre.org/techniques/T1021/. Accessed: April 20, 2026.
[36] MITRE ATT&CK. 2024. T1021.004: Remote Services: SSH, Sub-technique. https://attack.mitre.org/techniques/T1021/004/. Accessed: April 20, 2026.
[37] MITRE ATT&CK. 2024. T1053: Scheduled Task/Job. https://attack.mitre.org/techniques/T1053/. Accessed: April 20, 2026.
[38] MITRE ATT&CK. 2024. T1499: Endpoint Denial of Service. https://attack.mitre.org/techniques/T1499/. Accessed: April 20, 2026.
[39] MITRE ATT&CK. 2024. T1552: Unsecured Credentials. https://attack.mitre.org/techniques/T1552/. Accessed: April 20, 2026.
[40] MITRE ATT&CK. 2024. T1611: Escape to Host. https://attack.mitre.org/techniques/T1611/. Accessed: April 20, 2026.
[41] National Institute of Standards and Technology. 2021. CVE-2021-44228 Detail – National Vulnerability Database. https://nvd.nist.gov/vuln/detail/cve-2021-44228. Accessed: April 20, 2026.
[42] National Institute of Standards and Technology. 2022. CVE-2022-0847 Detail – National Vulnerability Database. https://nvd.nist.gov/vuln/detail/cve-2022-0847. Accessed: April 20, 2026.
[43] National Institute of Standards and Technology. 2024. CVE-2024-21626 Detail – National Vulnerability Database. https://nvd.nist.gov/vuln/detail/cve-2024-21626. Accessed: April 20, 2026.
[44] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. doi:10.1145/353323.353382
[45] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. *Implementing SELinux as a Linux Security Module*. Technical Report. NAI Labs.
[46] U.S. Government Accountability Office. 2018. *Data Protection: Actions Taken by Equifax and Federal Agencies in Response to the 2017 Breach*. Technical Report GAO-18-559. GAO. https://www.gao.gov/products/gao-18-559
[47] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *11th USENIX Security Symposium (USENIX Security 02)*. USENIX Association, San Francisco, CA, 17–31.

A Generative AI Usage

During the preparation of this work, the author used Anthropic's Claude Code to scaffold the structure of the paper, revise its text, and develop the prototype implementation and the scripts needed for the experimental evaluations in Section 8 and for packaging the artifact. After using this tool, the author reviewed and edited the content of the paper as needed, and manually inspected, validated and tested the generated code. The author takes full responsibility for the final content of the publication.

B Open Science

All artifacts supporting the claims of this paper are publicly available at https://github.com/lillo/bpfence and archived on Zenodo with the persistent DOI 10.5281/zenodo.22332429. The repository includes: the BPFence compiler and runtime daemon, the standard library, the case-study policies and their attack and benign triggers, and the micro- and macro-benchmarks of Section 8. A Vagrant-based reproducibility package is available to reproduce the experiments: an automated script rebuilds the toolchain, executes the case studies and the benchmarks, and produces the data of the tables of Section 8; a separate script runs the six exploit-based case studies, which fetch their exploits from upstream.

C Ethical Considerations

This work raises no ethical concerns: it discloses no new vulnerability, so no responsible-disclosure process applies, and the evaluation involves no human subjects, user data, or third-party systems, and runs entirely in a local VM. Most of the artifact's trigger programs reproduce the observable syscall patterns of documented CVEs and MITRE ATT&CK techniques without weaponizing the underlying vulnerabilities. Six case studies instead run public proof-of-concept exploits for upstream-patched kernel vulnerabilities. These exploits are fetched from the original repositories, are not modified in any way that extends their reach, and are executed only inside the VM.