Comprehensive Data Center Network Monitoring and Management with FARM

Abstract

Modern data centers have to manage increasing workloads. This puts accrued pressure on network monitoring solutions necessary for ensuring correct and efficient operation. Advances in network programmability have meanwhile led to yet more monitoring data being straightforwardly collected from switches, exacerbating bottlenecks in corresponding collection-centric approaches. This limits scalability and responsiveness, especially when several monitoring tasks are deployed side-by-side as is common for network management.

We present a novel comprehensive selection-centric solution for network monitoring and management (M&M), called FARM *(framework for network M&M), that significantly simplifies the development and deployment of network M&M tasks while being effective and scalable.* FARM*'s integrated design exploits switch hardware for a novel strongly decentralized software architecture aligned with a specifically designed programming model and integrated performance optimization framework. In short,* FARM *performs monitoring actions and reactions locally on switches to the extent possible, using task-specific centralized components only if and when needed, and globally optimizes placement across the network and monitoring tasks, considering intrinsically expressed placement constraints and commonalities among tasks.*

Deployed in a production data center, FARM *shows significant gains in responsiveness (up to 3427× faster over recent generic approaches and 4× faster over highly specialized solutions), and savings in network bandwidth (10000×) and computational effort. Placement optimization shows excellent scalability up to 10200 seeds across 1040 switches.*

1. Introduction

To manage networks, administrators need to continuously monitor them to detect exceptional behavior. Existing monitoring systems however exhibit many limitations affecting their semantics, scalability, and responsiveness (cf. *resource efficiency and full accuracy dilemma* [21]).

Information overload. Based on early constrained switch designs, monitoring approaches are traditionally *collection-centric*: collecting all information possible (raw samples, simple statistics) through simple agents executing on switches, forwarding it all to a logically centralized collector that computes a global picture of the network state by filtering and analyzing data sent by all agents (e.g., sFlow [38], IPFIX [14]).

More recent monitoring approaches exploit the increasing programmability of network devices to obtain yet more information, in particular from packet payload. However, sending raw packets and statistics from hundreds or thousands of switches to a collector can quickly congest network links and overwhelm the collector, even if implemented in a streaming fashion, such as for instance in Marple [35], Sonata [18], or Newton [50]. While some of these approaches aim to process raw statistics locally to switches in order to alleviate the bottleneck introduced by a logically centralized collector, they can only capture simple tasks since their statefulness is confined to aggregates (e.g., minimum, maximum, average, count).

Read-only semantics. Moreover such recent approaches lack abstractions to dynamically adapt their behaviour once a query is satisfied. Monitoring tasks can thus only pull information from switches but can not perform *local (re)actions* [25] in response, e.g., adding a ternary content-addressable memory (TCAM) rule or P4 [11] table entries to quench distributed denial of service (DDoS) attacks [32]. Triggering mitigating reactions thus requires additional out-of-band mechanisms, incurring an increased latency that can be prohibitive in many scenarios requiring fast reaction (e.g., DDoS attacks).

Narrow scope. In addition, to ensure correct behavior of the network, many monitoring tasks need to run simultaneously, e.g., to detect different types of anomalies such as heavy hitters (HHs), DoS attacks, super-spreaders, quality of service violations. Naïvely running several tasks independently side-by-side can lead to transmitting and processing the same data multiple times, exacerbating bottlenecks and multiplying operational costs. Existing solutions provide no opportunities for globally optimizing resource usage across the network and concurrently running monitoring tasks.

Last but not least, many solutions restrict their deployment scope to specific highly specialized HW or SW platforms [39, 24, 34, 46, 18, 50] to mitigate performance hurdles induced by design limitations mentioned before.

FARM. We present a novel monitoring and management (M&M) system called FARM (framework for network M&M) for accurate, efficient, scalable, and semantically rich network monitoring as well as management. FARM's design integrates HW and SW architecture, programming model, and optimization framework for prime performance. In short, FARM is fully *selection*-centric as opposed to collection-centric by supporting expressive, strongly decentralized, reasoning through so-called *M&M seeds* — or just *seeds* for short — deployable directly on a large range of network switch platforms. Seeds accurately poll traffic statistics, probe packets, and perform (re)actions *locally* to these network devices; they execute in a lightweight manner and interact among each other and with their *harvester* (i.e., a global analyzer) *only in specific, well-defined states, if at all needed*. FARM's programming model

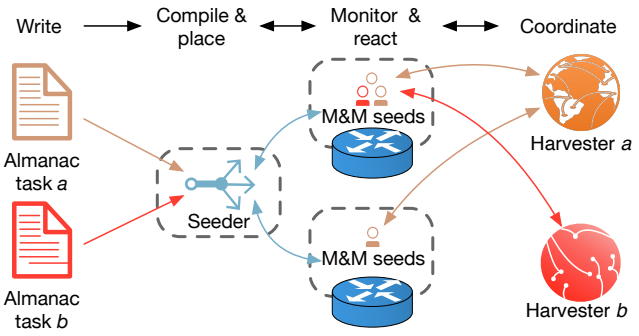

Fig. 1: FARM workflow overview. M&M tasks described in the Almanac domain-specific language, possibly by different users, are sent to the seeder. The seeder translates these into executable seeds and deploys them on switches in a network-wide optimized manner. At runtime, seeds (re)act locally and may provide information to their respective harvester if (and when) global coordination is needed.

exposes just enough information for joint and dynamic seed placement optimization. Fig. 1 shows the workflow of an M&M task in FARM from its description to its execution.

Contributions and roadmap. After pinpointing limitations of existing work in § 2, we present our novel solution:
Decentralized architecture **[DEC]** (§ 3): A key idea in the FARM approach is to run tasks and *perform actions where they belong*. FARM is designed for network management beyond simple monitoring. It exploits semantic knowledge from the programming model to efficiently use resources available on network devices. Seeds are executed on the switch level to get *select* information which is as timely accurate as possible and to immediately *perform required reactions directly*. A seed can nonetheless communicate with a harvester to take global decisions if needed, but does so much more efficiently since information is fully prefiltered locally.
Expressive model **[EXPR]** (§ 4): FARM uses a domain specific language (DSL) called automata language for network management and monitoring code (Almanac)[1] to describe M&M tasks by leveraging the intuitive abstraction of state machines to be executed as seeds. State machines are an expressive and well-known vehicle for capturing network policies concisely and precisely in a way cognizant of dynamics and amenable to verification [27]. Almanac makes it easy to succinctly describe M&M tasks as executable entities without knowledge of network topology or resources. It is specialized to define communication patterns, resource constraints and utility, placement policies, and local (re)actions. Almanac captures a wide spectrum of use-cases where seeds can analyze switch *statistics*, *packet payloads*, but also TCAM rules.
Globally optimized deployment **[OPTIM]** (§ 5): Our DSL abstractions allow FARM's runtime system to *dynamically deploy and relocate* seeds across devices without disruptions which facilitates holistic resource optimization — continuous in time and space — of seed placement for co-existing M&M tasks. To that end, FARM uses a novel, specialized optimization algorithm that considers network device resources, various overheads (e.g., seed migration), and *beneficial aggregation factors* from (re)using collected data for multiple M&M tasks deployed side-by-side.
Platform-independent implementation **[INDEP]** (§ 6): The implementation of FARM allows M&M tasks to be implemented on a wide variety of platforms. It is built on Stratum [7], an open-source framework that supports *HW and SW platforms of major vendors*.

Our empirical evaluation (§ 7) of FARM in a production data center (DC) of a major software vendor and cloud service provider[2] using different switch OSs and varying numbers of switches shows: (1) FARM experiences significant gains in responsiveness (up to 3427× faster over recent generic approaches and 4× faster over highly specialized solutions), precision, and savings in network bandwidth consumption (up to 10000×) and computational effort over the state of the art; (2) commodity switches can execute dozens of (even CPU-intensive) seeds with FARM; (3) FARM's global optimizer is scalable and efficient, capable of optimizing up to 10200 seeds across 1040 switches. We conclude in § 8. To the matter of presentation and comparability to existing work we use in the core paper the HH detection task — identification of flows beyond a threshold size — a very well-known task [44, 30, 16, 39]. We give a wide variety of other M&M tasks using Almanac in an anonymized online document [1].

2. Related Work

This section discusses closest related work on *generic* monitoring systems. Tab. 1 summarizes their shortcomings with respect to features and requirements introduced in § 1. Note that we view dynamic deployment (migration) as a prerequisite to global optimization and thus report it as subcategory there, although no prior work allowing such deployment attempts optimization across concurrent monitoring tasks as FARM does. From the many *specialized* solutions introduced for specific monitoring scenarios (e.g., HH detection [44, 30, 16, 39], DoS detection [43], link utilization [8]), we refer to a few later for comparison or for having influenced FARM's design.

sFlow [38] is a standard technology for monitoring network traffic encompassing: (1) *sFlow agents* implementing traffic sampling mechanisms; (2) a centralized *sFlow collector* analyzing samples or statistics. sFlow uses minimal switch-local processing or triage, performing all analysis on (2). This hampers latency as all statistical data has to be transferred there, and limits scalability. sFlow is not an IETF standard (cf. RFC 3176) but a golden standard widely deployed on many switch types of many vendors; thus we use it in our evaluation (§ 7).

[1] An almanac is a calendar with climate data and seasonal advice for agriculturers.

[2] Anonymized for double-blind reviewing.

Table 1: Features of *generic* network M&M solutions.

System	Local action [DEC]	Local reaction [DEC]	Statistics poll. [EXPR]	Payload poll. [EXPR]	Dyn. deploy. [OPTIM]	Poll. aggreg. [OPTIM]	HW [INDEP]	SW [INDEP]
sFlow [38]	○	○	○	○	●	○	●	○
Sonata [18]	●	○	●	●	●	○	○	○
Newton [50]	●	○	●	●	●	○	○	○
OmniMon [21]	●	◑	●	●	●	○	○	○
BeauCoup [12]	●	○	●	○	◑	○	●	○
Marple [35]	●	○	●	○	◑	○	○	○
FARM (this work)	●	●	●	●	●	●	●	●

Sonata [18] emphasizes "stream processing-like" network telemetry [5]. Sonata is implemented in parts in the data plane via P4 [11], offloading complex query parts to Spark Streaming [48]. The state of monitoring tasks is however limited to that of simple aggregation operations. Moreover, Sonata does not support merging of streams from several switches, and hence can not be used in many standard scenarios like global HH detection.[3] Sonata optimizes switch data plane resources for queries via a mixed-integer linear program (MILP) using the Gurobi solver out of the box [19], limiting scalability.

Newton [50] inherits Sonata's P4-based streaming approach and use of Spark Streaming. Newton however allows to deploy monitoring tasks dynamically and update queries without rebooting switches. Newton can also merge streams from several switches, yet despite some ideas to reduce streaming overhead, processing remains logically centralized, leading to scalability and responsiveness similar to Sonata.

OmniMon [21] tackles the collector bottleneck by separating tasks on end hosts and switches directly. Nevertheless, a centralized controller has to synchronize all hosts and switches, and share a global state. OmniMon does not optimize resource utilization across monitoring tasks, and provides no generic abstraction — all evaluated tasks are individually designed.

BeauCoup [12] abstracts hardware design similarly to FARM. The authors implement a memory-efficient and performant "coupon" system upon the TCAM over queries similar to Sonata and Newton. BeauCoup focuses on monitoring tasks that are solvable with a probabilistic distinct-counting, limiting the approach in terms of generality.

Marple [35] pioneered the stream-based monitoring approach, but, unlike other systems, supports data aggregation directly on switches by using local state. However, this support comes with a very limited set of aggregation primitives, which suffice only for basic statistics (e.g., counting out-of-order packets or aggregated packet latency) but not for advanced scenarios like HHs. In addition, Marple relies on a specific key-value store design implemented in HW on switches.

[3] Several of the authors propose a separate system for HH [20] where they propose to adapt their work in the future "*to detect network-wide heavy hitters [...] for inclusion in [...] a general network telemetry system.*"

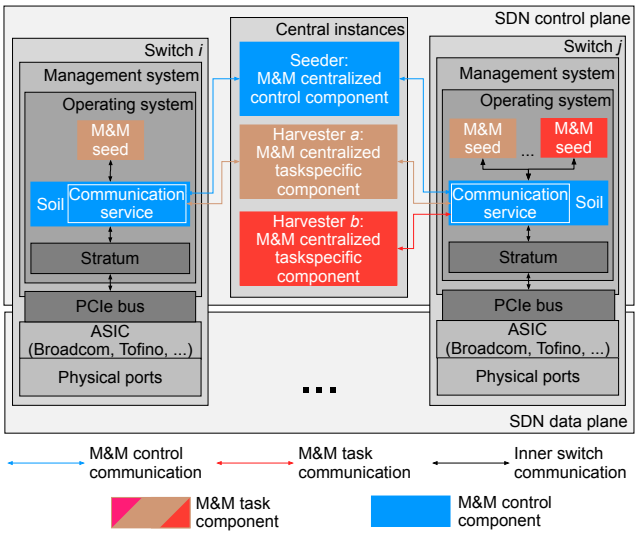

Fig. 2: FARM's architecture overview. Seeds interact via their soil with their harvester and other seeds.

3. M&M Architecture

We first present the complete high-level architecture of our framework for network M&M (FARM) and its components.

3.1. Synopsis

FARM builds on the idea of using switch-local support for execution of monitoring tasks (cf. Marple [35]) and extends this idea further to switch-local *management* tasks including *re*actions. Unlike pure monitoring, management decisions often cannot be made without any centralized coordination. One of the key features of the FARM design is that both monitoring and management functionalities can be decomposed into switch-local (distributed) components and centralized components. The former can take advantage of their proximity to the data source to monitor and actuate, while the latter (if needed) use a global view of system state. Communication among the two types of components follows a well-defined pattern expressed through a novel DSL called Almanac.

Most reactions involve the software-defined networking (SDN) control plane, hence, local reactions entail implementing distributed FARM components inside the switch-local *control plane*. FARM components are designed for generality and seamless deployment across a variety of underlying HW platforms upon a well-defined abstraction, Stratum [7]. For superior performance, switch-local components take advantage of HW resources of the switch as available, providing the best accuracy of monitoring information, with lowest possible delay, as it is crucial for a variety of time-critical measurement tasks (e.g. HH, hierarchical heavy hitter (HHH), DoS, DDoS, super-spreaders, quality of service violations). Placement of FARM's decentralized components is then globally optimized

through our purpose-built heuristic (see § 5).

FARM's design distinguishes between two types of components, as shown in Fig. 2: (1) M&M task components executing the logic of M&M applications, and (2) M&M control components managing deployment and execution of (2).

3.2. Switch-local Components

State-of-the-art DC switching devices have two main processing domains: a management system with a common CPU, and an ASIC for packet processing. While the ASIC is optimized for fast packet forwarding, the management system is responsible for communication with control devices, e.g., an SDN controller, and, following either local or global management decisions, updating the forwarding rules of the ASIC (i.e., reacting). FARM components run at the management system, but continuously poll packet processing statistics, including from P4 programs, or sample packets from the ASIC. To optimize usage of shared switch resources, e.g. polling bandwidth, by multiple M&M tasks, we introduce two types of switch-local components – an execution unit and a "hypervisor":

Seed: The M&M seeds — seeds for short — of an M&M task collect monitoring data (e.g., sampled packets, statistics), filter it, and analyze it with the goal of performing local management (re)actions (e.g., change its state, update TCAM rules, deploy new P4 table entries) immediately in response without requiring remote intervention. Seeds are stateful and run as lightweight instances (processes or threads) in the switch control plane. They may interact with other seeds and their harvester (a centralized M&M task-specific coordinator, cf. Fig. 1 and § 3.3) if and when needed for the M&M task. A seed definition, written in Almanac as detailed shortly in § 4, forms the core of an M&M algorithm. It includes an abstract description of where the corresponding seed instance(s) will execute in the network. As the seed behavior itself may need to change as part of local reaction, we introduce explicit states to its definition, where every state may listen to events and perform actions of its own choosing. Because of the high dynamics of FARM, a seed can also change its polling rate dynamically to reduce the switch resources required, which is important to optimize M&M tasks globally (see § 5).

Soil: The M&M seed foundation layer (soil) manages the execution of the seeds, tracks their switch resource usage, and optimizes/aggregates communication with the ASIC over a PCIe bus thus serving as abstraction layer between seeds and the switch. Resources tracked include three ASIC-specific types – *bus bandwidth* for packet probing, statistics *polling capacity*, and *TCAM space* for tracking specific flows and/or implementing various forms of local reaction. It employs its own communication service to establish and optimize its communication with remote components (i.e., centralized components or seeds or soil on other switches). The soil can aggregate polling when multiple seeds that execute different M&M tasks poll the *same data* from the switch. In such case, it is usually possible to poll the data only once for all these seeds to minimize communication to the ASIC and avoid contention. Such opportunities are statically analyzed and leveraged for aggregation benefits (see § 5) by the M&M centralized control instance (seeder). Note that the soil carefully divides the ASICs' TCAM between monitoring and packet forwarding such that the routing/switching behavior is not affected when rearranging the TCAM memory due to FARM operation. This approach draws inspiration from iSTAMP's [30] TCAM division, used for fine-grained monitoring, and extends it with an accurate polling mechanism between TCAM and seeds.

We detail the seed programming abstraction in § 4, but we would like to highlight here that this abstraction is more generic than query operations used in several prior approaches (e.g., Sonata [18], Marple [35]). This allows FARM to support altering local behavior, e.g., *re*acting to some stimuli, quickly, without the need to involve a centralized entity for decision making and/or seed redeployment (e.g., Newton [50]).

3.3. Centralized Components

Switch-local components may still require centralized components to partake in M&M tasks by taking centralized decisions based on data received from distributed seeds, and coordinate the placement and maximize joint utility of deployed seeds. FARM thus uses, respectively:

Harvester: Each M&M task can use its own specific centralized component, called harvester, that collects (or *harvests*) events sent by seeds of the task and takes global management actions for it when seed-local decision-making is insufficient.

Seeder: The M&M centralized control instance, called seeder, optimizes the resource utilization of all M&M tasks co-deployed over the network. It dynamically (un)installs and (re)positions the seeds following a global placement optimization algorithm (see § 5). The seeder also establishes the interface for seeds to communicate with each other either via the seeder or directly by requesting a seed's network location.

4. M&M Seed Programming Model

This section introduces our automata language for network management and monitoring code (Almanac) and illustrates it through the example of HH detection.

4.1. Language Overview

Almanac is centered around the concept of seeds, which are patterned after the well-known state machine abstraction. Almanac draws inspiration from a variety of more generic languages and models (e.g., Esterel [10], IO-Automata [17]) based on state machines due to programmers' familiarity with that abstraction in the space of networking (cf. [27]), adding features and actions specific to M&M (e.g., packet filter expressions). Fig. 3 presents a subset of the C-like syntax used to express state machines in Almanac. In the following $\overline{z}$ represents several instances of z, and $[z]$ means that z is optional. Blue highlighted keywords represent common state machine

almanac	*alm*	::= $\overline{strdec}$ $\overline{fundec}$ $\overline{ma}$
machine	*ma*	::= **machine** *mname* [**extends** *mname*] *mdef*
m. def.	*mdef*	::= {$\overline{pl}$**;** $\overline{xdec}$**;** $\overline{st}$}
var. dec.	*xdec*	::= [**external**] *typ x* [= *ex*] | *ttyp y* [= *ex*]
state	*st*	::= **state** *sname* {$\overline{xdec}$**;** [*ut*] $\overline{ev}$}
type	*typ*	::= **bool** | **int** | **long** | **string** | **list** | ... | **packet** | **action** | **filter** | ...
trig t.	*ttyp*	::= **time** | **probe** | **poll** | ...
utility	*ut*	::= **util** (*x*) {$\overline{ac}$}
place	*pl*	::= **place** (**all** | **any**) [$\overline{ex}$ | *ra*]
range	*ra*	::= [**sender** | **receiver**] [**midpoint**] [*ex*] **range** *op ex*
filter	*fil*	::= **dstIP** *ex* | **srcIP** *ex* | **port** *ex* | ...
event	*ev*	::= **when** (*trg*) **do** {$\overline{ac}$**;**}
trigger	*trg*	::= *rec* | *y* [**as** *x*] | **enter** | **exit** | **realloc**
receipt	*rec*	::= **recv** *pat* **from** (*mname* [@*dst*] | **harvester**)
operator	*op*	::= **and** | **or** | + | - | * | / | <= | >= | == | <>
expr.	*ex*	::= *v* | *y* | *x* | *fil* | **not** *ex* | *ex op ex* | ...
action	*ac*	::= *y* = *ex*; | *x* = *ex*; | **transit** *ex*; | **if** (*ex*) **then** {$\overline{ac}$} [**else** {$\overline{ac}$}] | **while** (*ex*) {$\overline{ac}$} | **return** *ex*; | **send** *ex* **to** (*mname* [@*dst*] | **harvester**);

Fig. 3: Core Almanac syntax. $\overline{z}$ represents several instances of *z*, [*z*] means that *z* is optional. Blue highlighted keywords represent common state machine constructs; orange highlights seed-specific primitives. Background coloring marks constructs with a common purpose: red is placement, green — resource allocation, and purple — event handling.

constructs; orange highlights seed-specific primitives.

Machines. A state machine **machine** has a name *mname*. This name describes the type of a seed and not a single seed instance. A state machine further includes a set of variable declarations *xdec*, a set of placement constraints *pl*, and a set of declarations of explicit states *st* for the machine. Variables marked with **external** are supplied at the deployment time providing the means to customize seed behavior. A machine optionally **extends** another machine. Almanac currently implements a simple form of single inheritance, where states *st* can be overridden in child machines; variables can not be overridden or shadowed, though. (More advanced mechanisms, e.g. [13], are under investigation.) Note that *trigger variables y* are special kinds of variables used for triggering events. They are assigned one of the following types *ttyp* — **time**, **poll**, and **probe**. Variables of type **time** or **poll** both denote events strictly periodic in time, but the latter represent polling data from the ASIC and contain filter information *fil* needed for seed placement optimization (see § 5). Type **probe** variables carry similar information to **poll** but are used to set up packet probing (sampling). The period provided for **probe** is only a lower bound and actual rate depends on the traffic. Semantics of placement constraints are explained in § 4.2.

States. Each discrete state **state** of a machine has a name *sname* and a definition including local variables *xdec* (**external** is disallowed), a set of events *ev* that can affect the machine in the given state, and a callback function *ut* estimating the seed's utility in a given state when supplied with resource allocation bound to variable *x*.

Note that as syntactic sugar (not shown in the abstract syntax for brevity), events can also be described at the level of a machine, which means that they apply to every single state. (Such global definitions are also subject to overriding.)

Events. Each **event** — asynchronous events are used to affect machine state — is defined by a trigger *trg* for executing it and a set of *ac*s performed in response. The trigger can be entering (**enter**) or exiting (**exit**) the state, the reception (**recv**) of a message from another machine (instance) or the harvester, a trigger variable reaching its triggering condition possibly assigning event data to a variable with **as** *x* (e.g., polled statistics, packet sample), or a resource reallocation (**realloc**) event due to placement (re)optimization (see § 5).

Receptions include pattern matching on messages and can constrain the source of a message to a given machine *mname* at a given *dst*, which can be a seed, a group of seeds, other switches, or a harvester. We omit the details of pattern matching for brevity. A simple and common pattern is a formal argument; if the received message has the same type as the argument, the corresponding value will be assigned implicitly.

Actions. The body of an event handler includes a sequence of actions — assignments of expressions to trigger variables (e.g., to modify polling rates) or regular variables, common control structures (**if**, **while**), sending (**send**) of messages to another machine *mname* at a given host *dst* or broadcast to all hosts (no *dst*), and explicit transitions (**transit**) to states.

Logic *without* state machine-related operations can be modularized into common auxiliary functions (*fundec*, omitted in the syntax for brevity), e.g. to operate on lists, filter TCAM rules, match regular expressions. These can **return** values.

Runtime library. The soil also provides a runtime support library, which, in particular, contains definitions for types `Probe` and `Poll` expected by respective trigger variables. The library also allows for querying resource information, e.g., `res()` returns a `Resources` structure containing amounts of allocated resources of each type. In addition, the forwarding rules in TCAM can be modified through the API using `addTCAMRule()`, `deleteTCAMRule()`, and `getTCAMRule()`. Finally, `exec()` runs external code (cf. ML example in § 7).

Utility callback function. Every state comes with a special *partial* callback function denoted by **util** which takes `Resources` argument with resource amounts and returns a float. To keep placement efficient, we impose a range of syntactic restrictions on **util**'s body:

- the allowed *ac*s are **if-then-else** and **return**;
- the allowed *op*s are **and**, **or**, ==, <=, >=, +, -, *, and /;
- function calls are forbidden except for `min` and `max`.

```
// Triggers
struct Probe { int ival; filter what; }
struct Poll { int ival; filter what; }
// Resource monitoring
struct Resources { float vCPU; int TCAM; ...}
Resources res() { ... }
// Dataplane
struct Rule { filter pattern; action act; }
void addTCAMRule(Rule rule) { ... }
void removeTCAMRule(filter pattern) { ... }
Rule getTCAMRule(filter pattern) { ... }
// Running external code
void exec(string command) { ... }
```

List. 1: Excerpt of runtime library's API.

Use case	Seed	Harv.	Use case	Seed	Harv.
Heavy hitter (HH)	29	12	Link failure [51]	31	8
Hier. HH [49]	21	26	Traffic change [41]	7	5
(inherited)			Flow size distr. [15]	30	15
Hier. HH [49]	38	26	Superspreader [46]	58	21
DDoS [32]	71	30	SSH brute force [23]	34	9
New TCP conn. [47]	19	5	Port scan [22]	44	23
TCP SYN flood [47]	63	18	DNS reflection [28]	83	22
Partial TCP flow [47]	73	18	Entropy estim. [33]	67	15
Slowloris [42]	44	29	FloodDefender [43]	126	35

Table 2: 16 well-known network monitoring and attack examples implemented in FARM with numbers of lines of code. The numbers include all code, e.g., also abstracted functions.

4.2. From Almanac to Tasks and Seeds

A network operator creates a task t by supplying the seeder with a set M^t of machines and, for each $m \in M^t$, the assignment of values to m's `external` variables. The seeder's *first step* for that task is to use the SDN controller to resolve `place` directives for each machines m, producing: the set of seeds S^m, and for each seed $s \in S^m$ the non-empty set of switches N^s, at exactly one of which s must be placed. The seeder's *second step* is to analyze `util` to determine each s's resource constraints $C^s(\overline{r_i})$ and utility function $u^s(\overline{r_i})$, where $\overline{r_i}$ is a sequence of variables representing allocated resource amounts. In essence, $C^s(\overline{r_i})$ reflects `util`'s domain, and $u^s(\overline{r_i})$ `util`'s return value; both are represented as explicit polynomials making them suitable for placement optimization (see §5). Finally, to infer aggregation opportunities, as the *third step*, for every seed, the seeder derives the set of `poll` variables Y^s, and for each $y \in Y^s$: the polling subject y`.what`, and the polling interval function y`.ival`$(\overline{r_i})$, which is allowed to depend on allocated resources.

Placement constraints. To find S^m and then N^s for each $s \in S^m$, the seeder considers a sequence of *pl* directives $\Pi_1, \ldots, \Pi_k$ from m's Almanac description with each *ex* inside Π_i fully evaluated to constants. So $\Pi_i =$ `place` q_i pc_i, where q_i is either `all` or `any` quantifier, and pc_i is an optional placement constraint. Then, S^m is simply a union of seed sets $\pi[\![q_i\ pc_i]\!]$ corresponding to individual Π_i, where $\pi[\![\cdot]\!]$ is the placement interpretation function described next.

```
machine HH {
  place all; S^m = {s_1,...,s_|N|}; ∀i. N^{s_i} = {n_i}
  poll pollStats = Poll {
    .ival = 10 / res().PCIe, .what = port ANY
  }; y.ival(r_1,r_2,r_3) = 10/r_3; y.what = {eth0,eth1,...}
  long threshold;
  action hitterAction;
  list hitters;
  state observe {
    util (res) {
      if (res.vCPU >= 1 and res.RAM >= 100) then {
        return min(res.vCPU, res.PCIe);
      }
    }
    when (pollStats as stats) do {
      hitters = getHH(stats, threshold);
      if (not is_list_empty(hitters)) then {
        transit HHdetected;
      }
    }
  } u^s(r_1,r_2,r_3) = min(r_1,r_3); C^s(r_1,r_2,r_3) = {r_1 - 1, r_2 - 100}
  state HHdetected {
    util (res) { return 100; }
    when (enter) do {
      send hitters to harvester;
      setHitterRules(hitters, hitterAction);
      transit observe;
    }
  } u^s(r_1,r_2,r_3) = 100; C^s(r_1,r_2,r_3) = ∅
  when (recv long newTh from harvester)
  do { threshold = newTh; }
  when (recv action hitAct from harvester)
  do { hitterAction = hitAct; }
}
```

List. 2: Heavy hitter (HH) seed example.

Consider a *pl* directive `place` q pc. Constraint pc is either a set of switch IDs, or pair of an optional filter *fil* identifying a set of paths (a missing filter denotes all paths) and range *ra* picking switches from each of those paths; a missing pc is interpreted as all switch IDs, i.e., $\pi[\![q]\!] = \pi[\![q\ n_1\ \ldots\ n_{|N|}]\!]$, where $\{n_1, \ldots, n_{|N|}\}$ is the set of all switches. For switch IDs, $q =$ `all` and $q =$ `any` are interpreted as "*every* switch gets a seed" and "*any* switch gets a seed", respectively:

$$\pi[\![\texttt{all}\ n_1\ n_2\ \ldots\ n_k]\!] = (\{n_1\}, \{n_2\}, \ldots, \{n_k\})$$
$$\pi[\![\texttt{any}\ n_1\ n_2\ \ldots\ n_k]\!] = (\{n_1, n_2, \ldots, n_k\})$$

A filter-based directive `place` q h m ex `range` $\oplus$ v, where q, h, m, ex, $\oplus$, and v correspond to syntactic constituents of *ra*, is interpreted in three stages: set of paths P^{ex} traversed by packets matching ex is obtained from the SDN controller; for every path $p \in P^{ex}$, the range constraint produces a subset of p's switches $\pi^p[\![h\ m\ \oplus\ v]\!]$; $q =$ `all` and $q =$ `all` either place a seed on *every* path or a single seed on *any* path:

$$\pi[\![\texttt{all}\ h\ m\ ex\ \texttt{range}\ \oplus\ v]\!] = (\pi^p[\![h\ m\ \oplus\ v]\!] : p \in P^f)$$
$$\pi[\![\texttt{any}\ h\ m\ ex\ \texttt{range}\ \oplus\ v]\!] = (\cup_{p \in P^{ex}} \pi^p[\![h\ m\ \oplus\ v]\!])$$

When h is unspecified, the union of both options is taken, i.e., $\pi^p[\![m \oplus v]\!] = \pi^p[\![\texttt{sender}\ m \oplus v]\!] \cup \pi^p[\![\texttt{receiver}\ m \oplus v]\!]$. Otherwise, $\pi^p[\![h\ m\ \oplus\ v]\!]$ for $p = (n_0, \ldots, n_l)$ simply filters

according to $d(i) \oplus v$, where $d(i)$ is a signed distance from a specific node on p: $\pi^p[\![h\ m\ \oplus\ v]\!] = \{n_i : d(i) \oplus l\}$. The distance d can be with respect to senders (`sender`) $d(i) = i$, receivers (`receiver`) $d(i) = k - i$, from midpoint to sender (`sender midpoint`) $d(i) = k/2 - i$, or from midpoint to receiver (`receiver midpoint`) $d(i) = i - k/2$.

Resource constraints and utility. For simplicity we start with a utility callback consisting of a single `if-then` statement whose condition does not use `or`. Expressions are converted to rational polynomials over $\overline{r_i}$ by expression interpretation function $\varepsilon^s[\![\cdot]\!]$. Specifically, if x_{res} is `util`'s formal parameter:

$$\varepsilon^s[\![x]\!] = s\texttt{.get("x")} \quad \varepsilon^s[\![v]\!] = \texttt{v} \quad \varepsilon^s[\![x_{res}.f]\!] = r_f$$

$$\varepsilon^s[\![ex_1\ op\ ex_2]\!] = \varepsilon^s[\![ex_1]\!]\ op\ \varepsilon^s[\![ex_2]\!] \text{ for } op \in \{+, -, *, /\},$$

where s`.get("x")` retrieves the current value of s's variable, and literal v is interpreted as is. The `if` condition produces a set of fractional polynomials, each *constrained to be non-negative*, using constraint interpretation function $\kappa^s[\![\cdot]\!]$:

$$\kappa^s[\![ex_1 \texttt{ >= } ex_2]\!] = \kappa^s[\![ex_2 \texttt{ <= } ex_1]\!] = \{\varepsilon^s[\![ex_1]\!] - \varepsilon^s[\![ex_2]\!]\}$$

$$\kappa^s[\![ex_1 \texttt{ == } ex_2]\!] = \kappa^s[\![ex_1 \texttt{ >= } ex_2 \texttt{ and } ex_1 \texttt{ <= } ex_2]\!]$$

$$\kappa^s[\![ex_1 \texttt{ and } ex_2]\!] = \kappa^s[\![ex_1]\!] \cup \kappa^s[\![ex_2]\!]$$

Thus, with `util(x){if (`ex_1`) then {return `ex_2`;}}` for the current state of s, $C^s(\overline{r_i}) = \kappa^s[\![ex_1]\!]$ and $u^s(\overline{r_i}) = \varepsilon^s[\![ex_2]\!]$.

Supporting `or` operators or several `if` requires instead of a single set C^s and a single function u^s sets $\{C^s_i\}_{i=1}^k$ and $\{u^s_i\}_{i=1}^k$ of those, meaning that utility is $u^s_i(\overline{r_i})$ once $C^s_i(\overline{r_i}) \geq 0$. For placement optimization (see §5) that would amount to splitting the seed into several copies, at most one to be placed.

Polling aggregation. For polling aggregation the seeder must know to what extent the two filters x_1`.what` and x_2`.what` for `poll` variables x_1 and x_2 share polling subjects, e.g., the same TCAM entries or the same logical interfaces. To determine that, filter interpretation function $\varphi^s[\![\cdot]\!]$ evaluates the expression into a boolean formula over *fil* with constants:

$$\varphi^s[\![\texttt{dstIP } ex]\!] = \texttt{dstIP } \varepsilon^s[\![ex]\!] \quad \varphi^s[\![\texttt{not } ex]\!] = \texttt{not } \varphi^s[\![ex]\!]$$

$$\varphi^s[\![ex_1\ op\ ex_2]\!] = \varphi^s[\![ex_1]\!]\ op\ \varphi^s[\![ex_2]\!] \text{ for } op \in \{\texttt{and}, \texttt{or}\}$$

A single filter may require polling multiple statistics from the ASIC (e.g., from several TCAM rules) so the precise sharing depends on the actual encoding of the filter. We assume that there exists filter encoding function φ^{enc} that returns a set of polling subjects given an output of $\varphi^s[\![\cdot]\!]$.

Finally, since the seed programmer may choose y`.ival` to depend on actual resource amount (encouraging allocation with `util`), we need to capture that dependency as well. Hence, for a seed s with poll variables Y^s and for each $y \in \mathrm{Y}^s$ defined as `poll` y`=Poll{.ival=`ex_1`,.what=`ex_2`}`, we derive y`.what` $= \varphi^{\text{enc}}(\varphi^s[\![ex_1]\!])$ and y`.ival`$(\overline{r_i}) = \varepsilon^s[\![ex_2]\!]$.

4.3. Illustration

Tab. 2 gives an overview of scenarios implemented with Almanac. We detail the example of heavy hitter (HH) detection (see List. 2) for illustration. HHs are flows with sizes larger than a given threshold. The example has two states `observe` and `HHdetected`. In the `observe` state, none of the observed ports are identified as an HH; as soon as the number of transmitted bytes of a port reach the defined `threshold`, a state transition to `HHdetected` occurs. In the `HHdetected` state, the current port list will be sent to the HH `harvester`. With this information, the HH harvester can react to the HHs in the network. In addition, local reaction will be performed that installs TCAM rules through auxiliary functions (abstracted inside the *setHitterRules* procedure) for the detected flows altering QoS policy for respective packets.

The two events for receiving messages are defined outside of the states; as mentioned, this syntactic sugar denotes that they apply to all states. In this example, the harvester sets up the threshold for an HH and can dynamically change it based on the overall traffic load in the network. If the network policy changes, the harvester can also modify the action that seeds apply locally to detected HHs. Auxiliary function *getHH* uses common programming constructs for determining which flows are HHs and is abstracted (thus *italicized*) for brevity. More advanced detection of hierarchical heavy hitter (HHH) (with and without inheritance) along with an outline of a corresponding harvester implementation, and seed code for all other examples of Tab. 2 are given in Appx. C.

5. M&M Seed Placement

This section formulates FARM's seed placement problem and discusses its hardness and our algorithmic solution.

5.1. Rationale and Overview

As introduced in § 3 and § 4, FARM enables the description of switch-local tasks and their deployment in a distributed manner as seeds on switches. Seeds $S^t = \bigcup_{m \in M^t} S^m$ of a task t may be positioned on different switches, seed s on exactly one of N^s. A seed may also be migrated; either due to placement constraint for the seed not being satisfied anymore (e.g., after a routing change) or due to a new seed with a higher utility needing to use the same switch. Migration induces extra resource usage due to seed's own inner state being synchronized between different nodes. The polling periods of a seed can depend on the actual resource allocation. Also, certain seeds can benefit from aggregation as they consider the same data. Considering the best performance for tasks and the many parameters and options available, in § 5.4 we propose a heuristic algorithm to optimize seed placement in FARM.

Tab. 3 summarizes notation of the involved entities. Lowercase letters denote elements of respective kinds, while the set of all elements of a kind is denoted by the corresponding uppercase letter. E.g., n denotes an individual switch while the set of all switches is represented by N. Also, $S = \bigcup_{t \in T} S^t$.

Table 3: Elements and notation of optimization model.

Description	Element	Set	Description	Element	Set
`poll` variables	y	Y	Seeds	s	S
Poll subjects	p	P	Resource types	r	R
M&M tasks	t	T	Switches	n	N

Table 4: Functions and variables of optimization model.

Optimization *input* description	Notation	Source
Set of seeds that belong to t	S^t	`place`
Set of switches where s can be placed	N^s	`place`
Polling interval function for variable y	$y.\texttt{ival}$	`poll`
Polling subject for variable y	$y.\texttt{what}$	`poll`
Utility function for seed s	u^s	`util`
Set of resource constraints for seed s	C^s	`util`
Available resources of type r on n	ares(n, r)	soil

Optimization *variable* description	Function
Returns 1 if all t's seeds are placed, else 0	tplc(t)
Returns 1 if s is placed on $n \in N^s$, else 0	plc(s, n)
Returns 1 if s is being migrated	migr(s)
Amount of type r res. assigned to s at $n \in N^s$	res(s, n, r)
Amount of type r_{poll} res. assigned to p at n	pollres(n, p)

5.2. Monitoring Utility

The optimization model for seed placement is captured below. The goal of optimization is to *maximize* the *monitoring utility*, which reflects the quality of monitoring for the entire system based on the resources assigned to each seed. The monitoring utility is defined as the sum over all seeds $s \in S$ and all switches $n \in N$, values returned by s's `util` callback with every resource $r \in R$ set to the assigned resources $\text{res}(s,n,r)$:

$$\text{maximize} \quad \sum_{s \in S} \sum_{n \in N^s} \text{plc}(s,n) \cdot u^s(\overline{\text{res}(s,n,r_i)}) \qquad \text{(MU)}$$

Tab. 4 summarizes the helper functions and variables used.

Migration overhead. While seed migration can generally optimize the seed layout, it incurs costs that must be considered. Migrating a seed consists of installing its description on the target switch and transferring its state over from the source switch. As the state is being transferred, and before it is deleted on the source switch, the seed resource utilization is temporarily doubled. Below, migr(s, n) checks if the seed s is migrating. We use plc'(s, n) to denote the *known value* of plc(s, n) from the previous placement run, i.e., current placement.

$$\text{migr}(s) = \sum_{n \in N^s} \text{plc}(s,n) \cdot (1 - \text{plc}'(s,n)) \quad \forall\, s \in S$$

Aggregation benefits. Aggregating seeds at the same switch can reduce data polling cost. Let $r_{\text{poll}} \in R$ be the resource type reflecting polling capacity. Our key assumption is that the demand for r_{poll} arising from trigger variable y is inversely proportional to the polling interval $y.\texttt{ival}(\overline{\text{res}(s,n,r_i)})$ with the coefficient $\alpha_{\text{poll}}(n)$, which may depend on switch n's architecture. As the demand is shared among poll variables s having the same polling subject, we introduce a single variable $\text{pollres}(n,p)$ for the amount r_{poll} consumed by a given polling subject p at switch n:

$$\begin{aligned}\text{pollres}(n,p) \geq\ & \alpha_{\text{poll}}(n) \cdot \text{plc}(s,n) / y.\texttt{ival}(\overline{\text{res}(s,n,r_i)}) \\ & + \alpha_{\text{poll}}(n) \cdot \text{migr}(s) / y.\texttt{ival}(\overline{res'(s,n,r_i)}) \\ & \forall\, n\ s\ p\ y, \text{ s.t. } y.\texttt{what} = p\end{aligned}$$

5.3. Constraints

Several constraints have to be taken into account when optimizing monitoring seed placement. E.g., we must guarantee that every seed of every task is placed on *some* switch, or, if there are not enough resources, that none of this task's seeds are counted toward utility. In the following, we describe all four constraints (C1)–(C4) required for valid seed placement.

Every seed is placed at one switch at most. This constraint guarantees that for any task $t \in T$ if one of t's seeds is placed, then every seed $s \in S^t$ is placed at exactly one switch $n \in N^s$:

$$\sum_{n \in N^s} \text{plc}(s,n) = \text{tplc}(t) \quad \forall\, t \in T,\ s \in S^t \qquad \text{(C1)}$$

Resources assigned for placed seeds. The amount of type r resources assigned to s on n, $\text{res}(s,n,r)$, shall keep s's `util` callback function within its domain:

$$\text{plc}(s,n) \cdot c(\overline{\text{res}(s,n,r_i)}) \geq 0 \quad \forall\, s \in S,\ n \in N^s,\ c \in C^s \qquad \text{(C2)}$$

Only placed seeds consume resources. Any seed s on some switch n can get at most the total amount of resources of type r available at switch n, but *only if it is placed there*:

$$\text{res}(s,n,r) \leq \text{ares}(n,r) \cdot \text{plc}(s,n)\ \forall\, s \in S,\ n \in N^s,\ r \in R \qquad \text{(C3)}$$

Switch resource limit for all seeds. The total of type r resources assigned to seeds at n cannot exceed the available amount at n with a special account for aggregated r_{poll}:

$$\begin{aligned}&\sum_{s,\ N^s \ni n} \text{res}(s,n,r) + \text{migr}(s) \cdot \text{res}'(s,n,r) \\ &\quad \leq \text{ares}(n,r)\ \forall\, n \in N,\ r \in R \setminus \{r_{\text{poll}}\} \\ &\sum_{p \in P} \text{pollres}(n,p) \leq \text{ares}(n, r_{\text{poll}})\ \forall\, n \in N\end{aligned} \qquad \text{(C4)}$$

5.4. Placement Optimization Algorithm

We first show that the placement optimization problem is expressible as a MILP under some linearity restrictions. Next we see that even with these restrictions, the problem remains hard in the worst case, so we propose a heuristic.

Conversion to MILP. If we make sure that u^s and C^s are linear, and $y.\texttt{ival}$ is inverse linear, then we would almost have the MILP formulation. The issue is that in equation (MU), inequality for pollres(n, p), and resource constraints (C2), there is an occurrence of $\text{plc}(s,n) \cdot f(\overline{\text{res}(s,n,r_i)})$ term where f is linear, but the expression as a whole violates linearity. Fortunately, thanks to (C3) we know that $\text{plc}(s,n) = 0$, implies $\text{res}(s,n,r) = 0$ for any r, hence we can rewrite the term in a linear form: $f(\overline{\text{res}(s,n,r_i)}) - (1 - \text{plc}(s,n)) \cdot f(\overline{0})$.

Algorithm 1 FARM seed placement optimization heuristic.

1. Sort tasks T in by decreasing minimum utility as $t_1, t_2, \ldots, t_k$.
2. For every $t = t_1, t_2, \ldots, t_k$
   (a) Repeat while possible: among $s \in S^t$ choose and *place* such s that adds the most to the utility *without unnecessary migration* (for existing seeds with valid placement).
   (b) If there remains $s \notin S^t$, remove S^t from the placement.
3. Redistribute resources using linear programming formulation.
4. For every seed s and every switch $n \in N^s$ compute the *migration benefit* as the increase in utility when s is migrated to n (expressible as a linear program).
5. Migrate the seeds in the order of decreasing *migration benefit*.

Hardness. The hardest part of placement optimization is assigning seeds to switches. After assignment, local optimization becomes a linear program and is efficiently solvable.

Getting the assignment right is an intractable optimization problem. If we assume one seed per task, ignore aggregation, and even consider only one switch, the remaining constraints (C2)–(C4) would still be at least as expressive as the multi-dimensional knapsack problem (MdKP). Thus, the placement optimization problem is *NP-hard in the strong sense* (*NP-hard* even if integer inputs are polynomial in problem size), just like MdKP [26], even for two resource types. Note that the hardness introduced by the multiplicity of resources is unavoidable; as we show in Appx. Bwitches have several bottlenecks.

Heuristic. To address possible scalability issues in larger deployments, we propose a simple heuristic in Alg. 1. The idea is to *place* tasks and seeds greedily with minimum utility and no migration, then *redistribute resources* with linear programming, and only then greedily *migrate* placed seeds.

6. Implementation

We elaborate on aspects of the implementation of FARM's components and Almanac, and of the seed placement.

6.1. FARM Components

Switch integration. FARM implements two drivers responsible for the communication between the CPU and the ASIC via the PCIe bus: one driver for Stratum [7] and one for Arista's EOSSDK [2]. Stratum is an OS module, available for Open Network Linux (ONL) among others, that abstracts the HW layer of ASICs from major vendors (e.g., Barefoot, Broadcom, Mellanox) to provide common interfaces (e.g., P4Runtime, OpenConfig). As such, FARM is deployable on all ASICs supported by Stratum and Arista EOS switches. We ensured that communication over the PCIe bus between the (i) CPU running the soil and seeds and (ii) ASIC can be scheduled to fully exploit the bus' capabilities.

Switch-local components. The seeds and the soil are optimized to execute directly on switching devices. Seeds can run as isolated processes or as threads of the soil process.

Communication between seeds and soil is done over a generic interface supporting two communication schemes — 1. gRPC [3] and 2. a tailor-fitted shared memory buffer usable when seeds are implemented as threads of the soil. gRPC's poor performance motivated the development of 2. We evaluate both seed execution models and communication schemes in Appx. B.

Centralized components. FARM's centralized components — the seeder and harvesters — are implemented in Python and contain a communication service to interact with the communication service of the soils to exchange data with both soils and the seeds they support. Communication between seeder/harvesters and soils (e.g. Tab. 6) is performed via RabbitMQ [6].

Almanac. State machines for seeds are described in Almanac, compiled by the seeder into XML, and transformed from XML to one or more seeds by each switch's soil. XML is used for interoperability and portability across OSs. The *ttype*s in Almanac (cf. § 4) are used by the soil to optimize communication with the ASIC over all running seeds.

6.2. Placement

The M&M placement function is in charge of optimizing the resource utility of the network following heuristics defined in § 5. This function takes as inputs (1) the set of switches using FARM, (2) their topology, (3) the switches' local resource consumption, (4) the current seed placement, and finally (5) the resource consumption of each seed. The function outputs the new placement of the seeds and the allocated resources, i.e., period for probing packets and polling statistics. We implemented the function in Rust [31] using a MILP library [4] supporting multiple solvers. The performance of FARM's placement heuristic is compared to Gurobi [19] in § 7.4.

The seeder calls the placement optimization algorithm every time an input parameter of the M&M placement function changes, e.g., when a switch's soil notifies the seeder that its resources are depleting. The seeder takes the actions necessary to realize the optimizer's output, e.g., by migrating seeds. When migrating a seed, the seeder first deploys the seed's description to its new location, then transfers its state there; seed execution resumes once the state is migrated.

7. Evaluation

We evaluate FARM in a production DC of a major software vendor and cloud service provider wrt three research questions:
(§ 7.2) How does FARM perform compared to state-of-the-art solutions in terms of responsiveness (i.e., reaction time, mitigation time), network load, and switch CPU load?
(§ 7.3) How does FARM's monitoring accuracy (which affects responsiveness) scale with a large number of — possibly CPU-intensive — seeds executed concurrently?
(§ 7.4) How does FARM's placement algorithm scale in terms of monitoring utility (cf. Eq. MU) and runtime?

Tab. 5: HH detection times with FARM, sFlow, Sonata, and specialized link utilization monitoring systems Planck and Helios.

System	Type	Time
FARM	Generic	1 ms
Planck 10 Gbps [39]	Specific	4 ms
Helios [16]	Specific	77 ms
sFlow [38]	Generic	100 ms
Sonata [18]	Generic	3427 ms

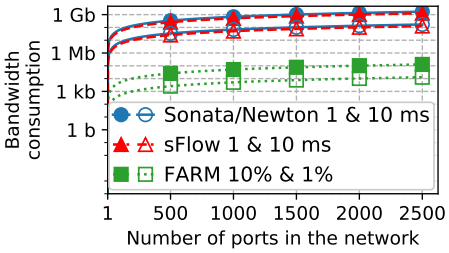

Fig. 4: Network load of FARM with 1 and 10% HH ratios, the sFlow collector with 1 and 10 ms accuracies, and similarly Sonata/Newton.

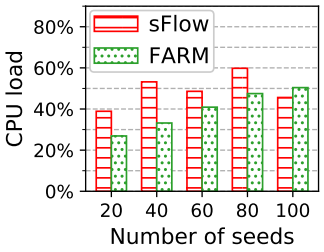

Fig. 5: Switch CPU load of FARM and sFlow for HH detection, 10 ms accuracy.

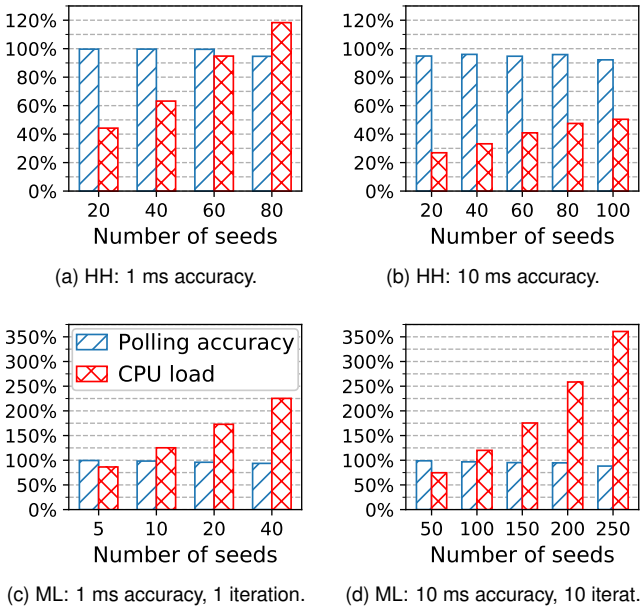

Fig. 6: CPU load of FARM for HH and ML tasks.

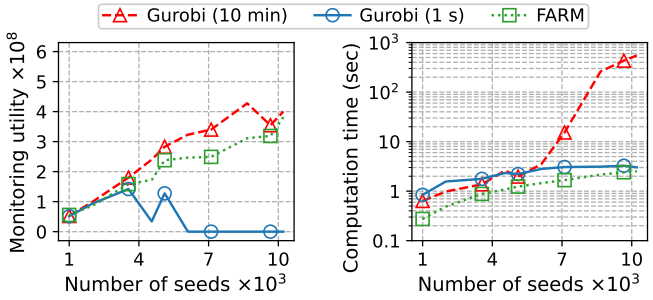

Fig. 7: FARM's global seed placement optimization algorithm (cf. Eq. MU) is close in utility to Gurobi with 10 min timeout and as fast as Gurobi with 1 s timeout.

We also show via a series of microbenchmarks the need for the optimizations implemented in FARM at the appendix (cf. § 6.1) due to space constraints. In particular, we show that FARM performs best with seeds executing as threads within the soil process and using a shared buffer for soil-seeds communication. We used this implementation for the rest of FARM's evaluation.

7.1. Setup

Platforms. We used APS BF2556X-1T, Accton AS5712, Accton AS7712, and Arista 7280QRA-C36S switches. APS BF2556X-1T run ONL with a 2.0 Tbps Intel Tofino ASIC and an Intel Xeon 8-core 2.6 GHz x86 processor with 32 GB SO-DIMM DDR4 RAM with ECC. Accton AS5712 run ONL and have an Intel Atom C2538 quad-core 2.4 GHz x86 processor with 8 GB SO-DIMM DDR3 RAM with ECC. Accton AS7712 have the same OS and CPU but twice the RAM. Arista 7280QRA-C36S run EOS and have an AMD GX-424CC SOC quad-core 2.4 GHz with 8 GB DRAM.

Topology. We deployed FARM on a cluster with a spine-leaf topology in a production DC of a major software vendor and cloud service provider. As FARM is undergoing a long-term evaluation period before being globally rolled out we report performance results on 20 switches.

HH task. We investigate the HH detection task presented in § 4.3 by deploying one of such seed per port on all switches.

ML task. Leveraging machine learning (ML) for prediction is a budding field in networks, with neural networks being used in various settings [37, 9, 45]. Additional support for prediction is an often stated need for monitoring [40]. We thus investigate FARM with a CPU-intensive task using ML to directly react to events on switches. This task relies on support vector regression [29] using matrix-matrix multiplications with 1000×1000 matrices. The Python implementation is executed via `exec()`, parameterized by the polled statistics.

7.2. Scalability

Identifying HHs is useful for many purposes (e.g., flow-size aware routing, DoS detection, traffic engineering). While HH detection does not show FARM's full potential, its widespread use in literature allows the best comparison (responsiveness, network and CPU load) against existing systems, including HH detection specific ones. Appx. C presents a wide variety of M&M tasks in FARM.

We evaluate in detail against sFlow [38] and Sonata [18] (and Newton [50] which extends Sonata with dynamic load-

ing), two generic solutions representatives of collector- and stream-based approaches. Other recent approaches show promising results, but have the same conceptual limitations as sFlow or Sonata, and are not publicly available.

Responsiveness. Tab. 5 compares the time needed to recognize an HH with FARM also to the more specialized Planck [39] (leveraging specialized HW) and Helios [16] systems. FARM shows great speedups while being at least as generic (sFlow) or more (Planck, Helios, Sonata). Transitively, FARM also greatly reduces mitigation time. Note that Sonata only computes a switch-local HH (cf. § 2).

FARM achieves such speedups by analyzing traffic directly on switches while the other solutions send simpler statistics and data to centralized instances. Another advantage of FARM is the ability to *react* on a switch when recognizing an HH. E.g., to install a rate limit for HHs, an `action` can be described with Almanac in the seed's `HHdetected` state. Both HH recognition *and* mitigation happen within 1 ms.

Network load. Fig. 4 depicts network load for HH detection and highlights FARM's benefits over sFlow and Sonata. We chose HH parameters based on observations in our production DC — HHs usually affect 1% of the network ports, 10% at worst, and the HH ratio changes up to once a minute.

sFlow periodically sends packets to probe every port in the network. We thus run sFlow with a 1 ms probing period to achieve a similar detection time as that of FARM, as well as with a 10 ms probing period to reduce load since the load of collector-based solutions increases linearly with the network size. Assuming Sonata could aggregate over several switches to compute HHs, the raw statistics issued by the switches to the Spark system deployed by Sonata would still create further network load. We run Sonata assuming an aggregation factor of 75%, which is the best that Sonata could achieve with an HH ratio changing up to once a minute. Further decentralizing aggregation by using more aggregation levels would only generate yet more network traffic. In comparison, FARM's bandwidth consumption increases by only 1 packet per minute if the network increases by 100 ports.

This yields also a linear gradient as shown in Fig. 4 (note the log y axis). The total amount and the slope is much lower for FARM than for the sFlow collector. Besides using less bandwidth, the computational effort of the centralized collector is much higher than FARM's ($> 1000\times$). Moreover, if the HH ratio changes more often, it is important to recognize the change immediately, which FARM enables.

CPU load. Fig. 5 depicts FARM's and sFlow's CPU loads as they poll statistics from multiple flow rules with equal monitoring accuracy. We do not compare against Sonata because it mirrors the traffic and thus its bottleneck is the sampling rate of the PCIe bus (cf. Appx. B). Its number of individual instances is not meaningful due to the lack of samples.

sFlow's CPU load is always higher than FARM's except with 100 flows. sFlow's CPU load is stable since it is a (locally) lightweight approach that samples packets and forwards them to its collector without filtering. On the other hand, FARM analyzes the data and manages its own state, thus CPU load increases with the number of monitored ports. Yet, as long as not all ports are affected, the SDN control plane is not congested with FARM unlike with sFlow (cf. Fig. 4).

7.3. Accuracy vs CPU Load

We evaluate the effect of running many collocated seeds on the same switch, specifically from the angle of monitoring accuracy (i.e., polling period) and its impact on CPU load.

HH task. Fig. 6a and Fig. 6b show CPU load with various numbers of seeds with every seed polling statistics from multiple flow rules every 1 and 10 ms respectively. The HH task incurs only light CPU load and easily scales to more than a hundred of seeds per switch with a 10 ms accuracy.

ML task. Because of their complexity we run ML seeds with a 1 ms accuracy in parallel (cf. Fig. 6c), and a 10 ms accuracy with statistics polling once but executing 10 iterations of the algorithm (cf. Fig. 6d) thus dividing by 10 the number of seeds executed in parallel. For comparison we use the same statistics polling periods as for the HH task.

Fig. 6c shows the CPU load is $\approx 150\%$ higher for the ML task with a 1 ms accuracy than for the HH task. This leads to the situation where the CPU is unable to handle all seeds in parallel due to the many context switches. By dividing the seed into partitions (cf. Fig. 6d), the CPU load decreases and the system scales well up to 250 seeds of this ML task.

7.4. Global Seed Placement Optimization

For evaluating FARM's global placement optimization algorithm, we compare it against a commodity MILP solver using Gurobi [19] (used by Sonata). Two timeouts are used for Gurobi: 1 s to get runtime similar to FARM (at the expense of utility), and 10 min as an absolute practical upper bound (to get similar utility). We test all approaches with up to 10 different tasks (cf. Tab. 2) comprising up to 10200 seeds and deploy them on 1040 switches. For every number of seeds to deploy, we execute 10 runs with different resource and placement requirements for tasks.

Fig. 7a shows the average monitoring utility (cf. Eq. MU) and Fig. 7b the average time needed to find a corresponding solution. FARM's placement optimization algorithm achieves similar utility compared to Gurobi but much faster, which is crucial with a large number of tasks and seeds to deploy.

8. Conclusions

We introduced FARM, a novel comprehensive solution for large-scale DC network monitoring and management (M&M). FARM relies on the Almanac programming framework for describing autonomous seeds that execute M&M tasks directly on switches; it uses a custom-designed scalable heuristic to

globally optimize their placement, considering heterogeneous HW resources, aggregation benefits, and migration overhead.

We evaluated the accuracy and scalability of FARM against existing generic systems (e.g., sFlow, Sonata) and specialized link utilization/HH detectors (e.g., Planck) showing reduced bandwidth requirements of centralized instances compared to collector-based approaches, and benefits of reacting directly to anomalies on the switch with predefined actions.

We are currently investigating several avenues for future work including fault tolerance and extensions to Almanac (e.g., advanced inheritance, combined implementation of seeds and harvester à-la tierless programming [36]).

References

[1] Anonymized online Appendix via Dropbox. https://raw.githubusercontent.com/asplos-farm-2023/submission/main/FARM.pdf.

[2] EOSSDK. https://github.com/aristanetworks/EosSdk.

[3] gRPC. https://grpc.io/.

[4] lp-modeler. https://github.com/jcavat/rust-lp-modeler.

[5] Network telemetry. https://tools.ietf.org/id/draft-song-opsawg-ntf-02.html.

[6] RabbitMQ. https://www.rabbitmq.com/.

[7] Stratum. https://www.opennetworking.org/stratum/.

[8] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. pages 89–92, 2010.

[9] C. Narendra Babu and B. Eswara Reddy. Performance comparison of four new ARIMA-ANN prediction models on internet traffic data. *Journal of Telecommunications and Information Technology*, pages 67–75, 2015.

[10] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *SIGCOMM*, pages 87–95, 2014.

[12] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. SIGCOMM '20.

[13] Brian Chin and Todd D. Millstein. An extensible state machine pattern for interactive applications. In *22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 566–591, 2008.

[14] Benoit Claise, Stewart Bryant, Simon Leinen, Thomas Dietz, and Brian H. Trammell. Specification of the ip flow information export (IPFIX) protocol for the exchange of ip traffic flow information (rfc 5101). *IETF RFC5101*, 2008.

[15] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 325–336, 2003.

[16] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. SIGCOMM '11, pages 339–350, 2011.

[17] Stephen J. Garland and Nancy Lynch. *Using I/O Automata for Developing Distributed Systems*, pages 285–312. Cambridge University Press, 2000.

[18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371, 2018.

[19] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[20] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, 2018.

[21] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. SIGCOMM '20.

[22] Jaeyeon Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy, 2004. Proceedings.*, pages 211–225, 2004.

[23] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. CCS '13, pages 85–96, 2013.

[24] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. pages 3–14, 2014.

[25] Lavanya Jose, Minlan Yu, and Jennifer Rexford. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Hot-ICE*, pages 13–13, 2011.

[26] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Multidimensional Knapsack Problems*, pages 235–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[27] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 59–72, 2015.

[28] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from hell? reducing the impact of amplification ddos attacks. In *23rd USENIX Security Symposium*, USENIX Security '14, pages 111–125, 2014.

[29] Y. Liang and L. Qiu. Network traffic prediction based on SVR improved by chaos theory and ant colony optimization. *International Journal of Future Generation Communication and Networking*, 8(1):69–78, 2015.

[30] Mehdi Malboubi, Liyuan Wang, Chen-Nee Chuah, and Puneet Sharma. Intelligent SDN based traffic (de)aggregation and measurement paradigm (iSTAMP). *INFOCOM*, pages 934–942, 2014.

[31] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 2014.

[32] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, April 2004.

[33] Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 746–755, 2008.

[34] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: Dynamic resource allocation for software-defined measurement. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 419–430, 2014.

[35] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, 2017.

[36] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-defined Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 519–531, 2014.

[37] D. C. Park and D. M. Woo. Prediction of network traffic using dynamic bilinear recurrent neural network. In *2009 Fifth International Conference on Natural Computation*, volume 2, pages 419–423, 2009.

[38] Peter Phaal, Sonia Panchen, and Neil McKee. Inmon corporation's sFlow: A method for monitoring traffic in a switched and routed networks. *IETF RFC 3176*, 2001.

[39] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. SIGCOMM '14, pages 407–418, 2014.

[40] Ganguli Sanjit and Corbett Ted. Magic quadrant for network performance monitoring and diagnostics. *Gartner Research ID G*, 354365, 2019.

[41] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 207–212, 2004.

[42] David Senecal. Slow DoS on the rise. https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html.

[43] Gao Shang, Peng Zhe, Xiao Bin, Hu Aiqun, and Ren Kui. Flooddefender: Protecting data and control plane resources under sdn-aimed dos attacks. In *IEEE Conference on Computer Communications*, INFOCOM '17, pages 1–9, 2017.

[44] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. SOSR '17.

[45] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, 2019.

[46] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 29–42, 2013.

[47] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 99–112, 2017.

[48] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '17, pages 423–438, 2013.

[49] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, SIGCOMM '14, pages 101–114, 2004.

[50] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, pages 295–308, 2020.

[51] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, 2015.

Appendix for FARM Paper #XXX

A. Seeder-Soil Communication

Since the centralized seeder (un)installs and (re)positions the seeds, it interacts with the communication service of the soil, deployed on all switching devices (see Fig. 2). Tab. 6 presents the main messages used in the process.

New seeds are added to a switch with `ADD_SEED`. The seeder generates and manages unique IDs for every seed instance. The unique IDs are important for establishing communication among FARM's components (e.g., between different seeds of the same M&M task). An existing seed can be changed with `MODIFY_SEED`. `MON_START` and `MON_STOP` messages are used to start and stop seeds respectively. The current state of a seed is set or queried by `MON_STATUS`. The state is stored at the seed, allowing FARM to migrate seeds between different switches.

For placement decisions the seeder needs resource usage information for subordinate switches. It may derive CPU and memory data by leveraging global view on the services deployed in the control plane. In contrast, availability of TCAM space depends on the dynamic behavior of network protocols; hence, such information must come directly from network devices. The seeder receives the current status of the TCAM with `GET_TCAM_STATUS` and sets thresholds on it with `SET_TCAM_THRES` to prevent FARM from occupying TCAM space needed for packet forwarding rules. The thresholds are observed by the soil running on the switch. The period for checking TCAM thresholds is defined with `CNFG_TCAM_TIMER`. The soil then checks the available resources of the switch. If the management system has no resources left to execute the seeds or the TCAM threshold is exceeded, the seeder (1) receives a `TCAM_THRES_EXCEED` message from the soil, and (2) starts a new optimization run to achieve better global M&M resource utilization (see § 5).

B. Implementation Microbenchmarks

We show via a series of microbenchmarks the need for the optimizations implemented in FARM (cf. § 6.1). In particular, we show that FARM performs best with seeds executing as threads within the soil process and using a shared buffer for soil-seeds communication. We used this implementation for the rest of FARM's evaluation. We deploy the ML task to benchmark switch HW utilization and identify HW bottlenecks. We use Accton AS5712 and AS7712 switches for the benchmarks and plot the averaged (similar) results.

PCIe bus capacity. Fig. 8 shows that the primal bottleneck for most M&M tasks is the PCIe bus. It rapidly congests as seeds poll the ASIC's TCAM. The PCIe bus capacity for polling traffic statistics is limited to 8 Mbps on both tested switches while their ASICs support 100 Gbps, showing a 1:12500 ratio between the two capacities. To circumvent the PCIe bus bottleneck, FARM enables, in addition to data sample polling, the soil to aggregate the seeds' requests before sending them over the PCIe bus.

Table 6: Control messages from seeder to soil.

Message	Description
`CNFG_TCAM_TIMER`	Set TCAM utilization check period
`SET_TCAM_THRES`	Set threshold on TCAM memory
`TCAM_THRES_EXCEED`	TCAM threshold set exceeded
`GET_TCAM_STATUS`	Fetch TCAM status
`<ADD|MODIFY>_SEED`	Add new seed/modify seed definition
`MON_PARAM_CHNG`	Change seed polling period
`MON_<START|STOP>`	Start seed/stop seed
`MON_STATUS`	Get or set status and state of a seed

Aggregation cost. Aggregating seeds' requests requires computation by the soil, thus trading PCIe bandwidth for CPU consumption. Fig. 9 shows CPU load for aggregation is only noticeable when seeds run as processes, and that thread-based seeds (within the soil) perform equally well regardless of aggregation, even with more than 100 seeds.

Latency overhead. Fig. 10 shows that gRPC scales linearly with the number of deployed seeds (i.e., connections) and thus becomes the latency bottleneck. As a fix we implemented a lightweight soil-seed communication scheme based on a shared buffer where seeds run as threads within the soil. Fig. 10 shows a negligible latency overhead of the shared buffer scheme even with 150 seeds.

C. Complementary Seed Examples

We illustrate M&M seed programming in Almanac with further well-known monitoring examples listed in Tab. 2.

Domain specific code is not discussed in the following examples but marked as *`italic`* (as mentioned it is obviously included in the number of lines of code reported in Tab. 2). The communication schema with the corresponding harvester is part of the examples. The harvesters are running on an SDN controller implementation (e.g., Ryu) and are not part of the following examples (cf. § 6). Many of the following examples are in the area of attack detection and prevention; for simplicity we omit "attack" from their names.

C.1. Hierarchical Heavy Hitter (HHH)

HHHs offer finer-grained HH source detection in a hierarchical topology — the traffic of a leaf switch (which was detected as a HH) is not taken into account when calculating the traffic of the spine switch, thus avoiding wrongly tagging a spine switch as an HH. Hierarchical heavy hitters (HHHs) are natural candidates for demonstrating inheritance (from HH). The `HHH` machine (List. 3) shows an example of HHH seed implementation that overrides the `observe` state from the `HH` machine (List. 2). Inheritance, in this case, reduces the number of LoC from 38 to 27. In short, the HHH harvester

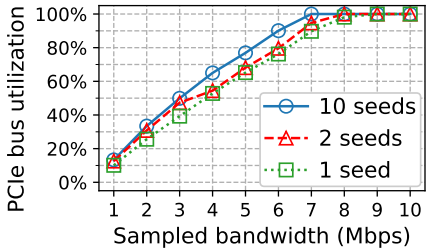

Fig. 8: The PCIe bus easily congests compared to the ASIC bus, calling for polling aggregation.

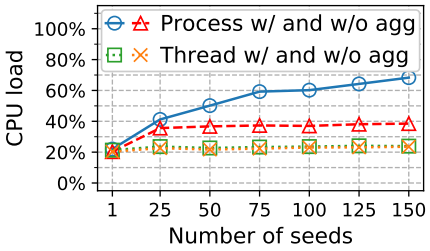

Fig. 9: Soil's CPU load showing the cost of seed requests' aggregation when seeds are threads vs processes.

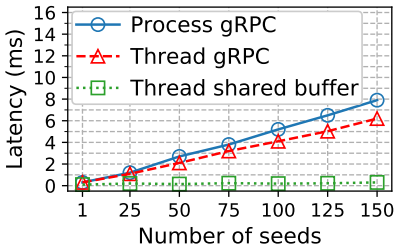

Fig. 10: Shared buffer vs gRPC communication latency between seeds being threads or processes and soil.

determines global HHs and periodically sends these to seeds, that incorporate those (`hitters`) when locally determining HHs.

List. 5 shows a possible HHH detection implementation without inheritance (compared to List. 3). It has two states `observe` and `HHHdetected`. In the former state no HHH is detected among the observed ports; as soon as a HHH is detected a state transition to the latter state occurs. In that latter state, first, the current port list is sent to the centralized HHH monitoring instance (`Harvester`). With this information the centralized HHH monitoring instance tells the affected seeds to update their lists of `subHHHs`. The list of detected HHH is updated in `HHHdetected`; if it changes a new message is sent the to centralized HHH instance. If there is no HHH detected anymore, the centralized HHH instance is informed and the state transits to `observe`. Here, `place all` specifies that the definition has to be installed on all switches. In this example the harvester (List. 4) is responsible for setting up the threshold for a HHH and distributing the list of detected HHHs; the list is important because HHs of child switches will not be considered by their parent switches for their HHH calculation.

```
machine HHH extends HH {
  list newHitters;
  state observe {
    when (pollStats) do {
      newHitters = getHHH(hitters, threshold);
      if (newHitters <> hitters) then {
        hitters = newHitters;
        transit HHdetected;
      }
    }
  }
  when (recv list newHs from Harvester) do
    { hitters = newHs; }
}
```

List. 3: Hierarchical HH (HHH) seed inherited from the HH seed implementation in List. 2.

C.2. Distributed Denial of Service (DDoS)

In a DDoS scenario [32] an attack is not initiated by one attacker, as in a DoS attack; many compromised nodes start

```
import ...
import HarvesterLib;

...
def receiveMsg(sender, earg):
  HHHList[sender.ID] = earg.newHHHList
  harvester.send("HHHList",
    getParents(sender.ID),
    HHHList)

def calcThreshold():
  threshold = getNetworkThreshold()
  harvester.send("threshold", threshold)

HHHList = [[], []]
harvester = HarvesterLib.Harvester("Hit")
harvester.receive += receiveMsg
threading.Timer(getThresholdPeriod(),
  calcThreshold)
```

List. 4: Excerpt of the Hitter harvester in Python handling communication with HHH seeds.

contributing to an attack to a target. Coordination is thus crucial to prevent a DDoS attack; often it can just be detected by the number of connection requests from different nodes. In such a (worst) case the network elements closer to the target are able to identify the attack and have to immediately forward this information to others to enact countermeasures.

A DDoS detection system, as presented in the seed example in List. 6, usually has four states: (1) In the `normal` state no DDoS attack (anomaly) is suspected. (2) In the second `anomaly` state a switch has identified an anomaly but cannot be sure whether it is just a short term peak. If such an anomaly is observed for a given time duration, a transition to `suspect` occurs. (3) In that `suspect` state all participants in the DDoS attack prevention have to be informed. (4) If a defined number of local instances corroborate the observation then the harvester informs all M&M seeds to change their state to `attack`. In this example the statistics polling accuracy is increased at the `anomaly` state. In the `suspect` state, flow rules against a DDoS attack with limits on individual flows are installed. The `attack` state can only be reached if a message from the DDoS harvester is received. If the attack ends only the DDoS harvester can transit states back to `normal` by broadcasting an `UndoAttack` message. In this example a certain IP

```
struct Wakeup { Host parent; }

machine HiHit {
  place all;
  time pollingStats = 10;
  long threshold;
  list subHHH;
  list HHHList;
  list newHHHList;
  Host parent;

  state dormant {
    when (recv Wakeup msg from Harvester) do {
      parent = msg.parent;
      transit observe;
    }
  }
  state observe {
    when (pollingStats) do {
      newHHHList = getHHH(subHHH, threshold);
      if (newHHHList <> HHHList) then {
        HHHList = newHHHList;
        transit HHHdetected;
      }
    }
  }
  state HHHdetected {
    when (enter) do {
      send HHHList to Harvester;
      if (parent <> "Harvester") then {
        send HHHList to HiHit@parent;
      }
    }
  }
  when (recv long newTh
    from Harvester) do {
    threshold = newTh;
    transit observe;
  }
  when (recv list nuHitrs
    from Harvester) do
    { subHHH = nuHitrs; }
}
```

List. 5: HHH seed example, expanded vs List. 2 + List. 3.

range `IPRangeX` is observed. At any time the placement policy and the TCAM rules which will be installed in a certain state (`attack`, `suspect`) can be changed during the execution of the monitoring algorithm.

C.3. New TCP Connections

The code in List. 7 checks and saves the number of new TCP connections (using the SYN flag [47]) created since the last `resetConn` message received from the responsible harvester. If the harvester sends a request, the seed replies back with the number of stored connections.

C.4. TCP SYN Flood

Compared to new TCP connection detection (cf. List. 7), a TCP SYN flood attack can be detected by counting the number of incomplete TCP handshakes in a time interval [47]. The corresponding seed implementation in List. 8 contains

```
machine NewTCPConnections {
  place all midpoint range == 1;
  time pollingStats = 10;
  long conns;
  state newConn {
    when (pollingStats)
    do { conns = conns + getNewConn(); }
  }
  when (recv "request"
    from Harvester) do
    { send conns to Harvester; }
  when (recv "resetConn"
    from Harvester) do
    { conns = 0; }
}
```

List. 7: New TCP connections seed example.

```
machine TCPSYNFlood {
  place all;
  time pollingStats = 10;
  time pollAnomaly = 10;
  list conns;
  int switchThresh;
  int indivThresh;
  state normal {
    when (pollingStats) do {
      conns = getIncompleteHandshakes();
      if (checkIndivHandshakes(conns,
        indivThresh))
      then { transit suspect; }
      if (checkGlobalConn(conns, switchThresh))
      then { setIndividualTCAMRules(conns); }
    }
  }
  state suspect {
    when (enter) do
      { setSuspectedTCAMRule(); }
    when (pollingStats) do {
      if (isNormal()) then { transit normal; }
      send "suspect" to Harvester;
      transit TCPSYNFlood;
    }
  }
  state TCPSYNFlood {
    when (enter) do {
      setAttackTCAMRule();
      send "attack" to Harvester;
    }
  }
  when (recv int newSwitchThresh
      from Harvester) do
    { switchThresh = newSwitchThresh; }
  when (recv int newIndivThresh
      from Harvester) do
    { indivThresh = newIndivThresh; }
  when (recv list undoAttack
      from Harvester) do {
    setTCAMRule(undoAttack);
    transit normal;
  }
}
```

List. 8: TCP SYN flood seed example.

```
machine DDoS {
  place all midpoint dstIP IPRangeX range == 1;
  external prefixPatt IPRangeX;
  time pollNormal = 10;
  time pollAnomaly = 1;
  int obsTime;
  int obsTimer;
  int oldState;
  Rule ruleAttack;
  Rule ruleSuspect;
  string placementPolicy;
  bool suspectSent = false;
  state normal {
    when (pollNormal) do {
      if (not isAnomaly()) then {
        send "pollNormal"
          to Harvester;
        obsTime = 0;
        transit anomaly;
      }
    }
  }
  state anomaly {
    when (pollAnomaly) do {
      if (isNormal()) then
        { transit normal; }
      if (obsTimer >= obsTime) then
        { transit suspect; }
      obsTime = obsTime + 1;
    }
  }
  state suspect {
    when (enter) do {
      suspectSent = false;
      setSuspectedTCAMRule(ruleSuspect);
    }
    when (pollAnomaly) do {
      if (isNormal()) then
        { transit normal; }
      if (not suspectSent) then {
        suspectSent = true;
        send "suspect" to Harvester;
      }
    }
  }
  state attack {
    when (enter) do {
      setAttackTCAMRule(ruleAttack);
      send "attack" to Harvester;
    }
  }
  when (recv list newTCAMRuleAttack
      from Harvester) do
    { ruleAttack = newTCAMRuleAttack; }
  when (recv list newTCAMRuleSuspect
      from Harvester) do
    { ruleSuspect = newTCAMRuleSuspect; }
  when (recv list newPlacement
      from Harvester) do
    { setPlacementPolicy(newPlacement); }
  when (recv int newObsTimer
      from Harvester) do
    { obsTimer = newObsTimer; }
  when (recv "attackMessage"
      from Harvester) do {
    oldState = currentState;
    transit attack;
  }
  when (recv "undoAttack"
      from Harvester) do
    { transit oldState; }
}
```

List. 6: DDoS seed example.

three states. The `normal` state considers an incomplete TCP handshake to be a packet trace consisting of a SYN packet and a SYNACK packet, with corresponding sequence number and acknowledge number, but without a subsequent ACK packet to complete the handshake. At the `normal` state first global flows are checked, and if a certain threshold is reached, the global flow are split into individual flows and checked. If the individual flows are sending a certain amount of SYN packets a transition to the `suspect` state is executed. In the `suspect` state the bandwidth of the suspected connections gets limited while in the `TCPSYNFlood` state the connections is interrupted.

C.5. TCP Incomplete Flood

A TCP incomplete flood attack [47] (cf. List. 9) is quite similar to a TCP SYN flood attack (cf. List. 8), but instead of suffering from an incomplete TCP handshake, the TCP flow as a whole is never completed. As with the TCP SYN flood attack, the TCP incomplete flood attack relies on finer details compared to the new TCP connections (cf. List. 7) example.

```
machine TCPIncompleteFlood {
  place all;
  time pollingStats = 10;
  time pollAnomaly = 10;
  list conns;
  int switchThresh;
  int indivThresh;
  state normal {
    when (enter) do { setTCAMRule(); }
    when (pollingStats) do {
      conns = getIncompleteFlows();
      if (checkIndivFlows(conns, indivThresh))
      then { transit suspect; }
      if (checkGlobalFlows(conns, switchThresh))
      then { setIndividualTCAMRules(conns); }
    }
  }
  state suspect {
    when (enter) do { setSuspectedTCAMRule(); }
    when (pollingStats) do {
      if (isNormal()) then { transit normal; }
      send "suspect" to Harvester;
      transit TCPIncomplete;
    }
  }
  state TCPIncomplete {
    when (enter) do {
      setAttackTCAMRule();
      send "attack" to Harvester;
    }
  }
  when (recv int newSwitchThresh
    from Harvester) do
    { switchThresh = switchThresh; }
  when (recv int newIndivThresh
    from Harvester) do
    { indivThresh = newIndivThresh; }
  when (recv list undoAttack
    from Harvester) do {
    setTCAMRule(undoAttack);
    transit normal;
  }
}
```

List. 9: TCP incomplete flood seed example.

```
machine Slowloris {
  place all;
  time pollingStats = 10;
  time pollAnomaly = 10;
  list conns;
  int threshold;
  state normal {
    when (enter) do { setTCAMRule();}
    when (pollingStats) do {
      conns = getIncompleteFlows();
      if (checkSlowloris(conns, threshold))
      then { setIndividualTCAMRules(conns); }
    }
  }
  state suspect {
    when (enter) do { setSuspectedTCAMRule(); }
    when (pollingStats) do {
      if (isNormal()) then { transit normal; }
      send "suspect" to Harvester;
      transit SlowlorisAttack;
    }
  }
  state SlowlorisAttack {
    when (enter) do {
      setAttackTCAMRule();
      send "attack" to Harvester;
    }
  }
  when (recv int newThreshold
      from Harvester) do
    { threshold = newThreshold; }
  when (recv list undoAttack
      from Harvester) do {
    setTCAMRule(undoAttack);
    transit normal;
  }
}
```

List. 10: Slowloris seed example.

C.6. Slowloris

A Slowloris attack [42], as implemented in List. 10, is related to the TCP incomplete flow attack. But in contrast to the TCP incomplete flow attack, a Slowloris attacker sends partial headers at a very slow rate (less than the idle connection timeout value on the server), yet never completes the request. The headers are periodically sent to keep sockets from closing, thereby keeping the server resources occupied. Due to the similarity to the TCP incomplete flow attack the description of the M&M seed looks similar and has the same states.

C.7. Link Failure Detection

To detect link failures, the ASIC offers a link status. The seed implementation in List. 11 takes advantage of the existing status messages and polls them. Once a change is detected (link recovers or link fails), an internal list gets updated, and the seed sends the updated list to the harvester and transits back to the observation state.

```
machine LinksStatus {
  place all;
  time pollingStats = 1;
  list newRecos;
  list newFails;
  state observe {
    when (pollingStats) do {
      newRecos = getRecoveries();
      newFails = getFailures();
      if (not is_list_empty(newRecos) or
        not is_list_empty(newFails)) then {
        transit changeDetected;
      }
    }
  }
  state changeDetected {
    when (enter) do {
      if (not is_list_empty(newRecos)) then
      { send newRecos to Harvester; }
      if (not is_list_empty(newFails)) then
      { send newFails to Harvester; }
      transit observe;
    }
  }
}
```

List. 11: Link status monitor seed example.

```
machine TrafficChangeDetector {
  list flowRules;
  list TCAMRule = detectorRules();
  time pollingStats = 10;
  state normal {
    when (pollingStats) do {
      flowRules = TrafficChangeDetection();
      setTCAMRule(flowRules);
    }
  }
  when (recv list newFlowRules
    from Harvester) do
    { flowRules = newFlowRules; }
}
```

List. 12: Traffic change detection seed example.

C.8. Traffic Change Detection

The traffic change detector is an external algorithm discussed in [41]. It detects changes in network traffic patterns (e.g., volume, number of connections). Depending on the results, resources have to be adjusted. The seed implementation in List. 12 simply executes at a given period the external traffic change detection function which returns the result that has to be sit.

C.9. Flow Size Distribution

Duffield et al. [15] set out to understand detailed flow statistics of Internet traffic on the basis of flow statistics compiled from sampled packet streams. Increasingly, only sampled flow statistics are available: inference is required to determine the flow characteristics of the original unsampled traffic and propose two inference methods. The scaling method codified

```
machine FlowSizeDistribution {
  probe sample = Probe { .interval = 100,
    .pattern = distributionRules() };
  time MBEPollingStats = 10;
  time SendingPollingRate = 1000;
  list MBEResults;
  list MLEResults;
  list SYNFlowsResults;
  packet samplePacket;
  state normal {
    when (MBEPollingStats) do
      { MBEResults = MBEResults +
        MomentBasedEstimator(); }
    when (sample) do {
      samplePacket = last_sample(sample);
      SYNFlowsResults = SYNFlows(samplePacket);
      MLEResults = MLEResults +
        MaximumLikelihoodEstimator(MBEResults,
          SYNFlowsResults);
    }
  }
  when (SendingPollingRate) do {
    send MLEResults to Harvester;
  }
}
```

List. 13: Flow size distribution seed example.

the heuristic that when sampling 1 out of N packets, since sampled flows have roughly $1/N$ of their packets sampled, the length of the original flow should be N times the sampled flow. The approach is implemented in List. 13 and only requires one state with two different polling rates to deliver (a) the statistics to the moment based estimator and (b) probe packets and start a maximum likelihood estimator. The past results are sent to the harvester.

C.10. Superspreader

List. 14 depicts superspreader detection [46]: sources that send traffic to a large number of distinct destinations. The seed counts source-destination pairs and classifies a source as superspreader once the number of pairs for that source crosses a given threshold, which depends on the network and its state. Once a superspreader is detected, the number of its connections can be limited and it can even be completely locked out.

C.11. SSH Brute Force

The code in List. 15 detects SSH brute force attacks [23]: repeated SSH login attempts by an attacker willing to gain shell access to one or multiple hosts. The seed first observes all SSH connections and, when an SSH brute force attack epoch is identified, switches to the `AttackParticipantAnalyzer` state. The seed then classifies the hosts appearing during the detected epochs as participants or non-participants of the attack, based on individual past history and "coordination glue", i.e., the degree to which a given host manifests patterns of probing similar to that of other hosts during the epoch. Participants of an attack will be excluded via TCAM rules.

```
machine Superspreader {
  place all;
  time pollingStats = 10;
  time pollAnomaly = 10;
  list conns;
  int switchThresh;
  int indivThresh;
  state normal {
    when (pollingStats) do {
      conns = getConnections();
      if (checkIndivConn(conns, indivThresh))
      then { transit suspect; }
      if (checkGlobalConn(conns, switchThresh))
      then { setIndividualTCAMRules(conns); }
    }
  }
  state suspect {
    when (enter) do { setSuspectedTCAMRule(); }
    when (pollingStats) do {
      if (isNormal()) then { transit normal; }
      send "suspect" to Harvester;
      transit superSpreader;
    }
  }
  state superSpreader {
    when (enter) do {
      setAttackTCAMRule();
      send "attack" to Harvester;
    }
  }
  when (recv int newSwTh
    from Harvester) do {
    switchThresh = newSwTh; }
  when (recv int newInTh
    from Harvester) do {
    indivThresh = newInTh; }
  when (recv list undoAttack
    from Harvester) do {
    setTCAMRule(undoAttack);
    transit normal; }
}
```

List. 14: Superspreader seed example.

C.12. Port Scan

The seed in List. 16 implements a threshold random walk algorithm [22] to rapidly detect portscanners based on observations (`getPortScans` function) of whether a given remote host connects successfully to newly-visited local addresses. The algorithm is motivated by the empirically-observed disparity between the frequency with which such connections are successful for benign hosts vs. known-to-be malicious hosts (`getDeltaY` function). With the result of the `getDeltaY` function a transition to corresponding state is executed, where necessary TCAM rules will be deployed. The thresholds for the approach are set by the harvester.

```
machine SSHBruteForce {
  place all midpoint range == 1;
  time pollingStats = 10;
  long globalFailIndic;
  long threshold;
  state AggregateSiteAnalyzer {
    when (pollingStats) do {
      globalFailIndic = globalFailIndic
        + getSSHConn();
      if (globalFailIndic >= threshold) then
        { transit AttackParticipantAnalyzer; }
    }
  }
  state AttackParticipantAnalyzer {
    when (enter)
    do { setTCAMRules(globalFailIndic); }
    when (pollingStats) do {
      calcAndRemoveAttacker();
      transit AggregateSiteAnalyzer;
    }
  }
  when (recv long ctrlTh from Harvester)
  do { threshold = ctrlTh; }
}
```

List. 15: SSH brute force seed example.

C.13. DNS Reflection

With a DNS reflection attack [28], systems running a certain TCP stack can be abused to amplify TCP traffic by a factor of 20× or higher. To prevent the attack, DNS calls from the inner system to suspected individual hosts have be checked. The seed implemented in List. 17 reflects four typical attack prevention states (`normal`, `anomaly`, `suspect`, `attack`). In the `normal` and `anomaly` states, polling and observing resources will be increased. In the `suspect` state, network resources will be reduced until an attack is identified (`attack` state) and the resources will be cut off for the attacker.

C.14. Entropy Estimation

To estimate the entropy of an unknown flow of packets, multiple probing techniques are necessary depending on the given flow and its state. Three different probing techniques have been discussed in the literature [33] to achieve good results. The seed implementation in List. 18 has four states. The `normal` state finds the best probing technique and transits to the state implementing the corresponding technique (states `linearProbing`, `balancedAllocation`, and `bloomFilter`). After computing the latest information in every state, the seed transits to the optimal probing schema. A summary of the past observations is sent to the harvester at the given `entropyStats` period.

C.15. FloodDefender

SDN-aimed DoS attacks can paralyze OpenFlow networks by exhausting the bandwidth, computational resources, and flow table spaces. FloodDefender [43] is a system to protect OpenFlow networks against SDN-aimed DoS attacks based

```
machine ThresholdRandomWalk {
  place all midpoint range == 1;
  time pollingStats = 10;
  list Y;
  long deltaY;
  long n1;
  state init {
    when (pollingStats) do {
      Y = Y + getPortScans();
      deltaY = getDeltaY(Y);
      if (deltaY >= n1) then
        { transit scanner; }
      if (deltaY <= n1) then
        { transit benign; }
    }
  }
  state scanner {
    when (enter) do {
      setExcludeScannerTCAMRules(Y);
      transit init;
    }
  }
  state benign {
    when (enter) do {
      setIncludeBenignTCAMRules(Y);
      transit init;
    }
  }
  when (recv long n1Update
      from Harvester) do
    { n1 = n1Update; }
}
```

List. 16: Port scan seed example.

on four modules: attack detection, table-miss engineering, packet filter, and flow table management. FloodDefender uses a queueing delay model to analyze how many neighbor switches should be used in the table-miss engineering. We propose a seed implementation in List. 19, made up of four different machines corresponding to the mentioned modules which also communicate with each other, a feature that is only shown in this example.

```
machine DNSReflection {
  place all dstIP IPRangeX range == 0;
  external prefixPatt IPRangeX;
  time pollNormal = 10;
  time pollAnomaly = 10;
  int obsTime;
  int obsTimer;
  int DNSThreshold;
  list ruleSpct;
  list ruleAtck;
  bool suspectSent = false;
  state normal {
    when (pollNormal) do {
      if (not isAnomaly()) then {
        send pollNormal to Harvester;
        obsTime = 0;
        transit anomaly;
      }
    }
  }
  state anomaly {
    when (pollAnomaly) do {
      if (isNormal()) then
        { transit normal; }
      if (obsTimer >= obsTime) then
        { transit suspect; }
      obsTime = obsTime + 1;
    }
  }
  state suspect {
    when (enter) do {
      setSuspectedTCAMRule(ruleSpct);
      suspectSent = false;
    }
    when (pollAnomaly) do {
      if (isNormal()) then
        { transit normal; }
      if (not suspectSent) then {
        suspectSent = true;
        send "suspect" to Harvester;
      }
    }
  }
  state attack {
    when (enter) do {
      setAttackTCAMRule(ruleAtck);
      send "attack" to Harvester;
    }
  }
  when (recv int newObsTimer
      from Harvester) do
    { obsTimer = newObsTimer; }
  when (recv list newRuleSpct
      from Harvester) do
    { ruleSpct = newRuleSpct; }
  when (recv list newRuleAtck
      from Harvester) do
    { ruleAtck = newRuleAtck; }
}
```

List. 17: DNS reflection seed example.

```
machine EntropyEstimation {
  Rule rule = entropyRules();
  time pollingStats = 10;
  time entropyStats = 1000;
  list pastObs;
  list EntropyValues;
  state normal {
    when (enter) do {
      transit findOptimalProb(rule);
    }
  }
  state linearProbing {
    when (pollingStats) do {
      pastObs = pastObs +
        getLinearProbingEntropy(rule);
      transit findOptimalProb(pastObs);
    }
  }
  state balancedAllocation {
    when (pollingStats) do {
      pastObs = pastObs +
        getBalancedAllocEntropy(rule);
      transit findOptimalProb(pastObs);
    }
  }
  state bloomFilter {
    when (pollingStats) do {
      pastObs = pastObs +
        getBloomFilterEntropy(rule);
      transit findOptimalProb(pastObs);
    }
  }
  when (entropyStats) do {
    send pastObs to Harvester;
  }
}
```

List. 18: Entropy estimation seed example.

```
machine AttackDetectionModule {
  place all;
  time pollingStats = 10;
  long threshold;
  state checkTableMiss {
    when (pollingStats) do {
      if (getNumTableMiss() > threshold)
        then { transit splitTableMiss; }
      transit stopAttack;
    }
  }
  state splitTableMiss {
    when (enter) do {
      deploySplitTableMiss();
      send "start"
        to AttackDetectionModule@Modules;
      send "tableMiss"
        to Harvester;
    }
  }
  state stopAttack {
    when (enter) do {
      send "stopAttack"
      to AttackDetectionModule@Modules;
    }
  }
}

machine PacketFilterModule {
  place all;
  list threshold;
  state startModule {
    when (enter) do {
      observePacketFilterParameter();
      setThreshold();
    }
  }
  state stopModule {
    when (enter) do { reset(); }
  }
  when (recv list newThreshold
    from AttackDetectionModule@Module) do {
    threshold = newThreshold;
    transit startModule;
  }
  when (recv "stopAttack"
    from AttackDetectionModule@Module) do {
    transit stopModule;
  }
}

machine TableMissEngineeringModule {
  place all;
  list schema;
  state startEngine {
    when (enter) do {
      installProtectionRules(schema);
      send "currentSchema"
        to AttackDetectionModule@Neighbors;
    }
  }
  state stopEngine {
    when (enter) do {
      uninstallProtectionRules(schema);
      send "uninstallSchema"
        to AttackDetectionModule@Neighbors;
    }
  }
  when (recv list newSchema
      from AttackDetectionModule@Neighbors) do {
    schema = newSchema;
    installProtectionRules(schema);
  }
  when (recv "uninstallSchema"
      from AttackDetectionModule@Neighbors) do
    { uninstallProtectionRules(schema); }
}

machine FlowTableManagement {
  place all;
  list rules;
  state startModule {
    when (enter) do
      { manageRules(rules); }
  }
  state stopModule {
    when (enter) do
      { manageRules(rules); }
  }
  when (recv list newRules
      from AttackDetectionModule@Module) do {
    rules = newRules;
    transit startModule;
  }
  when (recv "stopAttack"
      from AttackDetectionModule@Module) do
     { transit stopModule; }
}
```

List. 19: FloodDefender seed example.