MatchBox: A Semantic Foundation for Data Plane Portability

ERIC HAYDEN CAMPBELL, University of Texas at Austin, USA
ROBERT ZHANG, University of Texas at Austin, USA
DIVYANSHU SAXENA, University of Texas at Austin, USA
ADITYA AKELLA, University of Texas at Austin, USA
IŞIL DILLIG, University of Texas at Austin, USA

Match-action tables are the core abstraction underlying network packet-processing systems, from fixed-function switches to eBPF-based software dataplanes. However, their concrete syntax and semantics vary widely across programming environments, reflecting differences in hardware generations, engineering practices, and vendor design choices. This syntactic and semantic variation renders portability of match-action tables across environments a persistent challenge. This paper presents MatchBox, a system for translating match-action tables across heterogeneous environments. At its core is the *Match Algebra*, a compositional formalism for concisely and declaratively expressing transformations on match-action tables. To ensure unambiguous semantics, MatchBox introduces a static type system based on *guarded functional dependencies (GFDs)* that guarantees that every well-typed Match Algebra expression denotes a well-defined function. From such specifications, the MatchBox compiler efficiently computes compact target tables that are semantically faithful. Across case studies in programmable switches, multi-cloud firewalls, and eBPF systems, MatchBox enables concise, declarative portability specifications and realizes them as compact target tables.

CCS Concepts: • **Theory of Computation** → **Program Semantics**; **Type Structures**; • **Networks** → **Programming Interfaces**; • **Software and its Engineering** → **Domain-Specific Languages**; **Semantics**.

Additional Key Words and Phrases: match-action tables, functional dependencies, data plane portability

ACM Reference Format:
Eric Hayden Campbell, Robert Zhang, Divyanshu Saxena, Aditya Akella, and Işıl Dillig. 2026. MatchBox: A Semantic Foundation for Data Plane Portability. *Proc. ACM Program. Lang.* 10, PLDI, Article 199 (June 2026), 24 pages. https://doi.org/10.1145/3808277

1 A Common Language with No Common Meaning

Throughout the network stack, packet-processing programs rely on a common abstraction: the *match-action table* [6, 8, 39], which comprises a set of rules that inspect a packet's fields and apply actions to modify or forward it. Match-action tables are ubiquitous: hardware switches use them to steer packets through pipeline stages [41, 49]; firewalls use them to filter traffic [51]; while programmable switches [25, 50] and SmartNICs [20, 33] use them to express bespoke packet-processing logic. In short, match-action tables are the *lingua franca* for packet processing.

Unfortunately, this common language has no common meaning: two network programs may expose the same match-action table interface, and still assign different semantics to it. For example, in Figure 1, two switches S and T each expose tables $Route_X$ and ACL_X for $X \in \{S, T\}$. Although $Route_S$ and $Route_T$ both match on dst and may invoke route, they differ in their treatment of misses,

Authors' Contact Information: Eric Hayden Campbell, University of Texas at Austin, Austin, USA, eric.hayden.campbell@utexas.edu; Robert Zhang, University of Texas at Austin, Austin, USA, robertz@cs.utexas.edu; Divyanshu Saxena, University of Texas at Austin, Austin, USA, dsaxena@cs.utexas.edu; Aditya Akella, University of Texas at Austin, Austin, USA, akella@cs.utexas.edu; Işıl Dillig, University of Texas at Austin, Austin, USA, isil@cs.utexas.edu.

ACM 2475-1421/2026/6-ART199
https://doi.org/10.1145/3808277

Switch Interface for $X \in S, T$	
Routing table:	$Route_X(\text{dst}) \in \{\text{route}(p),\ \text{miss}\}$
Access-control list:	$ACL_X(\text{dst}) \in \{\text{allow},\ \text{deny}\}$

Interpretation	S	T
miss	*drop*	*flood*

Fig. 1. Two switches with identical interfaces but different interpretations of miss

i.e., packets whose route is unknown. On switch S, a *miss* causes the packet to be dropped; on switch T, it causes the packet to be *flooded* to all neighbors. Thus, while these routing tables *appear* interchangeable at the interface level, they induce observably different behavior (drop vs flood). What looks like a common abstraction is, in fact, an illusion.

Without a shared semantics, porting match-action tables across environments is challenging. In practice, engineers rely on a kitchen sink of ad hoc translation scripts to adapt tables written for one environment to another. These approaches are brittle: empirical studies attribute a significant fraction of network outages to errors in match-action tables [54], and a 2022 cross-sector survey found that exploitable network misconfigurations impose substantial costs on large organizations [52, 53]. Furthermore, as network engineers upgrade to 400/800 GbE networks in furtherance of AI workloads [29], these portability challenges are only becoming more acute.

A seemingly obvious remedy to this challenge is standardization: that is, give the match-action abstraction a precise unifying syntax and semantics. Historically, multiple efforts have sought to do exactly this. For example, OpenFlow [39] exposes a single monolithic match-action table that inspects all packet headers and triggers a list of actions. However, this "one-size-fits-all" design has struggled to keep pace with evolving hardware capabilities. Subsequent efforts, such as the Switch Abstraction Interface [43], have provided a menu of interfaces to fit application-specific needs. This suggests that heterogeneity is an inevitable feature of the ecosystem; rather than eliminating it, we must learn to work with it.

This paper presents MatchBox, a principled framework for porting match action tables across heterogeneous environments. MatchBox assigns a precise semantics to table transformations via a novel declarative language: *Match Algebra*. Its type system guarantees that every well-typed specification denotes a single realizable behavior, and its compiler realizes these specifications as concrete match-action tables in the target environment.

The design of the Match Algebra carefully balances *expressiveness* with *realizability*. On one hand, it must be expressive enough to capture transformations that arise in practice; on the other hand, it must be sufficiently constrained to ensure those transformations can be implemented as concrete match-action tables. This tension is fundamental: the very flexibility needed to express common transformations also admits ambiguous specifications that do not denote a single, implementable behavior. For example, reducing the number of match keys can merge previously-ßdistinct rules, allowing multiple actions to match the same packet (see Section 2). Such transformations are not necessarily *realizable* as match-action tables, which must select one action for each packet.

To address this issue, MatchBox's type system statically prohibits ambiguity through *guarded functional dependencies (GFDs)*, a novel extension of functional dependencies from database theory [1]. At a high level, GFDs provide a composable formalism for specifying the conditions under which a subset of match keys determines observable behavior. For instance, the GFD $\{\text{dst} \mid \text{vsn} = 4\} \rightarrow \{\text{port} \mid \text{port} \neq 0\}$ states that "when vsn = 4, dst uniquely determines the action and port, which is guaranteed to be non-zero." When an expression C satisfies a GFD, the transformation specified by C is guaranteed to induce a single action for each packet, and is therefore implementable as a match-action table.

The final piece of the MatchBox toolkit is a *compiler* for realizing well-typed Match Algebra expressions in the target environment. Given a specification and a set of source tables, the compiler produces concrete rules that can be installed in the target table. The key challenge here is compactness, as unnecessarily large match-action tables can increase resource usage (e.g., power consumption, or lookup time) [8]. The MatchBox compiler addresses this challenge through two insights: First, it defines a monadic structure over match-action tables to leverage the compression already present in the input tables. Second, to tame the potential explosion in the number of rules, it uses solver-aided reasoning to deal with complex filtering operations.

Taken together, MatchBox replaces ad hoc translation scripts with typed, declarative language, and a compiler for realizing them. We exercise MatchBox in three realistic scenarios drawn from across the networking stack: (1) evolving switch programs, (2) restructuring Virtual Private Cloud firewall configurations, and (3) adapting eBPF programs to new memory structures. In each domain, the Match Algebra permites high-level declarative transformation specifications; its type system proves their unambiguity; and finally, the MatchBox compiler produces compact match-action tables for the target environments.

To summarize, this paper makes the following contributions:

(1) We introduce the *Match Algebra*, a declarative formalism for transforming match-action tables.
(2) We develop a novel static type system based on *guarded functional dependencies (GFDs)* that ensures every well-typed expression denotes a deterministic function from packets to actions.
(3) We design and implement a monadic *solver-aided compiler* that faithfully implements well-typed Match Algebra specifications as concrete tables.
(4) We evaluate MatchBox in realistic switch, firewall, and eBPF scenarios, showing that it captures practical translation patterns while efficiently generating compact tables.

2 Motivating Example

In this section, we motivate the benefits of MatchBox through a concrete example that illustrates how subtle differences across hardware switch architectures cause portability challenges[1].. In a network, the *data plane*, which consists of interconnected hardware *switches*, is responsible for performing various operations (called *network functions*) over packets, including forwarding them to their destinations. However, the data plane's behavior is not fixed; the *control plane* dynamically (re-)configures it by installing rules in match-action tables.

As networks evolve, engineers upgrade old switches, fix firmware bugs, or introduce new hardware architectures, each of which may introduce new match-action tables, with different schemas or semantics. Likewise, eBPF-based data planes may refactor table structures, requiring updates to the table rules. To accommodate such changes, existing match-action tables must be *ported* to align with the target's structure while preserving their intended behavior. Unfortunately, in practice, seemingly minor differences can require substantial modifications to the underlying rules. To illustrate this issue, let's revisit the example switches S and T from Figure 1 which expose tables with identical interfaces but different semantics.

As shown in Figure 1, both S and T have a $Route_X$ table and an ACL_X table, for $X \in \{S, T\}$. Here, $Route_X$ selects an egress port p (via the route action) based on the packet's destination address dst, and ACL_X enforces access control by deciding whether access to that destination should be allowed (via allow) or denied (deny). We assume when no entry matches in an *ACL* table, the packet is denied. The core difference lies in the miss action of the *Route* table. In S, any packet that triggers miss is dropped, while $Route_T$ *floods* the packet to all neighbors (Figure 1).

[1]While this example focuses on dataplane switches, these challenges occur throughout the networking stack. For instance, Section 8 observes similar challenges in eBPF programs and virtual private cloud deployments

$Route_S$(dst)	
10.1.1.0/24	route(1)
10.1.0.0/16	route(2)
0.0.0.0/0	miss

ACL_S(dst)	
10.1.0.10/32	deny
10.1.10.0/24	deny
10.0.0.0/8	allow

Fig. 2. Tables for switch S. Packets without routes trigger miss in $Route_S$ and are dropped.

$Route_T$(dst)	
10.1.1.0/24	route(1)
10.1.0.0/16	route(2)
0.0.0.0/0	miss

ACL_T(dst)	
10.1.0.10/32	deny
10.1.10.0/24	deny
10.1.1.0/24	allow
10.1.0.0/16	allow

Fig. 3. Tables for switch T. Packets without routes trigger miss in $Route_T$ and are flooded.

Now, the challenge is to write code that can automatically configure T to match the semantics of the configured S program. To see why this is a challenge, consider the entries shown in Figure 2, where the matches are expressed using *classless inter-domain routing (CIDR)* notation, which expresses IPv4 addresses in the form $o_1.o_2.o_3.o_4/\ell$ where each o_i for $i = \{1, 2, 3, 4\}$ is a decimal octet, and ℓ is an integer between 0 and 32 expressing the prefix of bits that are relevant. By convention, irrelevant bits are marked with zero. For instance `10.1.1.0/24` matches all bitvectors whose first 8 bits denote 10, whose second 8 bits denote 1, and whose third 8 bits denote 1. Any bitvectors that match multiple prefixes trigger the action for the rule with the *longest prefix*—e.g., in $Route_S$, `10.1.1.10` triggers route(1), not route(2).

Thus, for the configuration of Figure 2, $Route_S$ sends all traffic destined to `10.1.1.0/24` out port 1 and all traffic to the broader prefix `10.1.0.0/16` out port 2, dropping any packet that does not match one of these entries. Next, the access-control list ACL_S overlays a security policy on top of this forwarding behavior: it blocks traffic to the specific destinations `10.1.0.10/32` and `10.1.10.0/24`, while permitting access to the rest of the `10.0.0.0/8` block. As a result, packets with destination `10.1.1.10` are forwarded out port 1 and allowed, whereas a packet to `10.1.0.10` is deny-ed by the ACL (for simplicity we equate deny and drop) despite having routing behavior determined by $Route_S$. Packets that lie outside `10.0.0.0/8` or do not match any entry in $Route_S$ are discarded, reflecting the default "drop-on-miss" behavior of switch S.

Now let us consider porting the entries in Figure 2 from switch S to switch T. The most obvious "solution" is to simply copy the entries of ACL_S into ACL_T and the entries of $Route_S$ into $Route_T$. However, this migration will cause the two switches to behave differently: packets that trigger miss in $Route_X$ and are allowed by ACL_X will be flooded when $X = T$, and dropped when $X = S$. Misconfigurations like this can cause *broadcast storms* that disrupt network availability [42].

```
def compute_acl_T(route_S, acl_S):
  acl_T = []
  denies = []
  for rule in route_S:
    if rule.action == "miss":
      denies.append(rule.cidr)
  for rule in acl_S:
    if rule.action == "deny":
      acl_T.append(rule)
    elif any(rule.cidr in d \
             for d in denies):
      rule.action = "deny"
      acl_T.append(rule)
    else:
      acl_T.append(rule)
  return acl_T
```

Fig. 4. Pseudocode for ACL_T.

The correct solution is to tweak the configuration of ACL_T to ensure that the packets that are dropped by $Route_S$ trigger deny in ACL_T, as shown in Figure 3. Intuitively, ACL_T now *whitelists* exactly those destinations that are both routed and allowed in S, and denies everything else. Concretely, traffic to `10.1.1.0/24` and `10.1.0.0/16` is still forwarded (out ports 1 and 2, respectively) and allowed, while packets to the more specific prefixes `10.1.0.10/32` and `10.1.10.0/24` remain denied. Any destination that falls under the broader `10.0.0.0/8` block but does not match one of these routed prefixes (e.g., `10.2.0.0/16`) is now explicitly denied by ACL_T. This configuration compensates for T's flood-on-miss behavior: even though $Route_T$ would flood packets that miss all routing entries, ACL_T prevents them from being forwarded, so the overall forwarding behavior remains the same.

In practice, production tables typically contain thousands of rules, so it is not feasible to manually rewrite each table entry. Thus, to deal with the portability challenge, engineers write translation

scripts to perform this migration. For example, Figure 4 shows simplified automation code for our running example. This code first extracts the list `denies` of match prefixes for the rules dropped by $Route_S$, then it walks the entries of ACL_S in priority order: for each non-deny rule, it checks whether the current CIDR prefix is contained within one one of the prefixes in `denies`, in which case, the action is changed to `deny`.

e prefixes in `denied`, (2) its match prefix is contained in some to-be-denied prefix in `denied`, in which case it must be denied, or, (3) it is overlapping an existing prefix and must be split into sub-addresses so the covered region can be denied. This logic is tricky to reason about and hard to get right. In fact, we invite the reader to find the subtle bugs[2] in the code shown in Figure 4.

With Match Algebra, we can replace the code in Figure 4 with the following:

$$\begin{aligned} Route_T &:= Route_S \\ ACL^* &:= Route_S \boxtimes ACL_S \\ ACL_T &:= ACL^* \lhd [(\mathsf{route}, \mathsf{allow}) \mapsto \mathsf{allow}; (\_, \_) \mapsto \mathsf{deny}] \end{aligned}$$

Here, the first line simply copies the $Route_S$ table into the $Route_T$ table. The second line performs a *parallel join* of $Route_S$ and ACL_S, producing a new intermediate table that produces an action pair (a_1, a_2) where a_1 is the action from $Route_S$ and a_2 is the action from ACL_S, along with the disjoint union of the arguments. This intermediate table, ACL^*, is shown in Figure 5. Finally, we need to transform this intermediate table into the final result by renaming actions through our *range transformation* operator $\cdot \lhd \cdot$. In particular, any action pair $(\mathsf{route}(p), \mathsf{allow})$ is mapped to allow, while everything else is mapped to deny. This yields the table shown in Figure 3.

$ACL^*(\mathsf{dst}) \in \left\{ \begin{array}{l} (\mathsf{route}(p), \mathsf{deny}), \\ (\mathsf{route}(p), \mathsf{allow}) \end{array} \right\}$	
`10.1.0.10/32`	(route(2), deny)
`10.1.10.0/24`	(route(2), deny)
`10.1.1.0/24`	(route(1), allow)
`10.1.0.0/16`	(route(2), allow) (route

Fig. 5. Intermediate Table

From vision to realization. As this example shows, our broader vision is to let engineers express translations using small, declarative rules rather than involved, ad hoc scripts like the one from Figure 4. However, realizing this vision exposes a deeper tension between two perspectives on match-action tables. From an *extensional* point of view, a match-action table specifies a function from packets to outcomes. The goal of the transformation in this example is to simply preserve this input–output behavior: given the same packet, the translated configuration should yield the same forwarding outcome as the original. From an *intensional* point of view, however, realizing this transformation is nontrivial: We must produce concrete entries in the target tables $Route_T$ and ACL_T, as shown in Figure 3. However, these extensional and intensional views may not *always* align: we will see in Section 5.1 that some declarative specifications describe packet behaviors that cannot be realized by any match-action table.

$ACL'_T(\mathsf{dst}) \in \{\mathsf{deny}, \mathsf{allow}\}$	
`10.1.0.10/32`	deny
`10.1.10.0/24`	deny
`10.1.1.10/32`	allow
`10.1.1.0/24`	allow
`10.1.0.0/16`	allow
`10.0.0.0/8`	deny

Fig. 6. $ACL'_T \equiv ACL_T$.

MatchBox bridges this gap through two core components. First, its static type system ensures that well-typed Match Algebra expressions have coinciding extensional and intensional semantics: for any source configuration (such as the one in Figure 2), their packet-level meaning is guaranteed to be realizable by a finite table. This guarantee relies on *guarded functional dependencies* (GFDs), which capture when one set of match fields uniquely determines an action (and which of its arguments). For instance, for our running example, the following type specifies that the action is

[2]Hint: Add `10.1.9.0/24` $\mapsto$ route(9) to $Route_S$. The computed ACL_T will drop packets matching this prefix, when it should route them. To fix this bug, we need to add the intersecting routed prefixes above the dropped ones in the *else* branch, and change the `exists` check to a reachability check, both of which are nontrivial.

uniquely defined by the destination field for both $Route_S$ and ACL_S tables:

$$Route_S : \{\mathsf{dst} \mid \top\} \rightharpoonup \{\mathsf{p} \mid \top\} \qquad\qquad ACL_S : \{\mathsf{dst} \mid \top\} \rightharpoonup \{\emptyset \mid \top\}$$

Based on this annotation, the type system can verify that the Match Algebra specification defining ACL_T is well-formed, meaning that it can be realized as a concrete match-action table.

Given such a well-typed expression, the MatchBox compiler then translates the match algebra specification into concrete target configurations, automatically generating the entries in Figure 3. Importantly, while there are many semantically equivalent match-action tables, the MatchBox compiler takes care to generate configurations with as few entries as possible, thereby ensuring that the resulting tables are efficient to deploy. In our running example, this means producing exactly the concise set of CIDR prefixes in Figure 3 instead of, e.g., the semantically equivalent table in Figure 6, which repeats and overshadows broader prefixes with unnecessarily specific rules.

3 A Formal Model of Match-Action Tables

Conceptually, match-action tables are lookup structures: given a concrete packet value, they return an action to execute (and its arguments). Seen in this way, tables are partial functions that inspect a packet (Packet) and select an action identifier $a \in \mathsf{Action}$ along with their arguments $v \in \mathsf{Args}$:

$$\mathsf{Packet} \rightharpoonup (\mathsf{Action} \times \mathsf{Args}),$$

This *extensional view* captures a table's observable input-output behavior. However, network operators do not manipulate forwarding behavior on the level of individual packets; rather, they configure match-action tables, using ordered sets of *rules* (a.k.a. *rows* or *entries*). Informally, a rule is a triple $\langle \mu, a, v \rangle \in \mathsf{Row}$ comprising a guard μ, which is a (restricted) predicate indicating a set of packets, and the corresponding action a and v. Then, *intensionally*, a table $T = \langle R, < \rangle$ is a pair comprising a set of rules $R \subseteq \mathsf{Row}$, and an order $(<) \subseteq R \times R$ on those rules, that is:

$$\mathsf{Table} \triangleq \{\langle R, < \rangle \mid R \subseteq \mathsf{Row}, (<) \subseteq R \times R, \text{where } < \text{ strict, total}\}$$

Informally, to execute on a packet, the match-action table T selects the highest-priority rule whose guard is satisfied by the packet, and returns the action and arguments in that rule. In the remainder of this section, we formalize the extensional vs. intensional semantics of match-action tables, as one of the core challenges we solve is to reconcile these two views.

Notation. To reason about packets and tables we need some notation. We presume a countable set of variables Var. For a function $f : \mathsf{Var} \rightharpoonup V$ for any set V, we write $f \downarrow x$ for $x \in \mathsf{Var}$ when f is defined on x and $f \uparrow x$ otherwise. Then, by abuse of notation, $\mathrm{dom}(f) = \{x \in \mathsf{Var} \mid f \downarrow x\}$. We also define $f[x \mapsto v]$ to be the function equivalent to f except on x, which it maps to v. For sets of variables $\{x_1, \ldots, x_n\} \subseteq \mathsf{Var}$ we write $\overline{x}$. Finally, define the function $f[\overline{x}]$ to be equivalent to f on all variables $x \in \overline{x}$ for which $f \downarrow x$ and undefined otherwise.

Remark. As MatchBox focuses on translating between fixed interfaces, we do not provide an interpretation of the actions, treating them as a given set of identifiers.

3.1 Match-Action Tables from an Extensional Perspective

Fundamentally, match-action tables manipulate network packets, which we model as finite maps from a set of variables (also called *packet fields*) to bitvectors $b \in \mathbf{2}^n$ (i.e., sequences of n bits in $\mathbf{2} = \{0, 1\}$), and we write $|b| = n$. We model packets as partial valuations $v : \mathsf{Var} \rightharpoonup \mathbf{2}^*$ and write $\mathsf{Val} \triangleq \mathsf{Var} \rightharpoonup \mathbf{2}^*$ for the set of all valuations. Further, we define $\mathsf{Packet} \triangleq \mathsf{Val}$ and $\mathsf{Args} \triangleq \mathsf{Val}$.

Semantically, tables denote partial functions from valuations v (that is, packets) to an *output* pair (a, v'), where a is an *action label* and v' is a valuation comprising the arguments to the action a.

In examples, we stylize these pairs $a(v')$, to invoke the fact that the execution environment (e.g. the data plane switch or cloud firewall) will execute code corresponding to a with arguments v'. However, from the perspective of MatchBox and the Match Algebra, actions are (combinations of) names drawn from an abstract set A. To complicate matters, some match-action interfaces permit operators to specify *lists* or *sets* of base actions, so we assume Action is the free monoid over A with identity ε and multiplication, which we write (a, b). Putting these together, we define the set of *semantic tables* $\mathcal{T}$ as follows: $\mathcal{T} \triangleq \mathsf{Val} \rightharpoonup \mathsf{Action} \times \mathsf{Val}$.

3.2 Match-Action Tables from an Intensional Perspective

Intensionally, match action tables are ordered sets of rows $\rho = \langle \mu, a, v \rangle \in \mathsf{Row}$ comprising an action a, arguments v, and a predicate μ that determines conditions under which the table selects a and v. For a row ρ, we use "dot notation" to get the respective components (eg. $\rho.\mu$).

However, to meet the extreme performance and operational requirements of network functions (e.g. 800 Gbps line rate processing in data centers [29], and rapid reconfiguration by control code [10]), the predicates μ are restricted to a simple form. In particular, a guard μ is a conjunction of membership checks, each represented by a *bitpattern* $p \in \mathbf{s}^n$, which is a sequence of n symbits $\mathbf{s} = \{0, 1, *\}$, where $*$ denotes a wildcard. Thus, symbits (and hence bitpatterns) denote sets of bitvectors: given $p \in \mathbf{s}^n$, we write $[\![p]\!] \subseteq \mathbf{2}^n$ for the set it represents, e.g. $[\![0*]\!] = \{00, 01\}$. For convenience, we use $b \in p$ as shorthand for $b \in [\![p]\!]$. The ternary domain admits natural set-like operations, lifted elementwise to $\mathbf{s}^n$. We use $\wedge$ for intersection: e.g., $1 \wedge * = 1$. Now, formally, a guard $\mu \in \mathsf{Guard}$ is a partial function $\mu : \mathsf{Var} \rightharpoonup \mathbf{s}^*$. We write $v \models \mu$ when for all $x \in \mathrm{dom}(\mu)$, $v(x) \in \mu(x)$. We also define $v \models \rho$ to hold iff $v \models \rho.\mu$ and say ρ *is enabled for* v .

Finally, we can formally define a match-action table structure Table as a pair $T = \langle R, < \rangle \in \mathsf{Table}$, where R is a finite set of rules and $<$ is a strict total order on R representing rule priorities. We write $\rho < \rho'$ to mean that ρ' has higher priority than ρ. The semantics of a table depend on the unique maximal element with respect to $<$. For any non-empty $S \subseteq R$, we define $\max_<(S)$ as the unique element of S that dominates all others under $<$. Intuitively, $\max_<(S)$ selects the "highest-priority rule" among a set of candidates.

Now, we can define the semantics of a table using function notation: $T(-) : \mathcal{T}$. First, we describe the set of enabled rules. Given an input valuation v, the set of enabled rules is $T|_v = \{\rho \in R \mid v \models \rho.\mu\}$. If $T|_v = \emptyset$, we say v *misses* in T and define $T(v)$ to be undefined, writing $T \uparrow v$. Otherwise, the selected rule is $\rho = \max_<(T|_v)$, and the table returns $T(v) = (\rho.a,\ \rho.v')$. We say T is defined on v, writing $T \downarrow v$. Hence, a table denotes a partial function mapping input valuations to actions and their arguments.

4 The Match Algebra

In this section, we introduce the Match Algebra, including the observations that led to its design. Match Algebra is inspired by classical algebraic theories of relations (namely, Tarski's relation algebra [28] and the relational algebra from database theory [1]) which provide representation-independent operators for querying and combining relations. In the same spirit, Match Algebra offers a logical view of network configurations, allowing operators to describe what a transformation means semantically without specifying how it is realized on a particular switch.

4.1 Design of the Match Algebra

Different environments expose different interfaces for inspecting or transforming packets. Thus, transforming tables often requires changing the *representation* of the policy without changing what the policy means. For instance, this can require adjusting which fields are inspected, how actions are named, or what information each stage produces for the next.

Consequently, the Match Algebra must capture nominal changes between table representations: different network programs may use different names for the same fields (e.g. ipv4Dst vs sw_dst) or actions (drop vs deny, or nop vs allow). Match Algebra supports these adaptations through two constructs. First, a *domain transformation* ($\beta \triangleright C$) reshapes a table's view of packet state by adding/renaming/modifying match fields ($x := e$) or restricting guards to certain fields ($\pi(\overline{x})$). Dually, *range transformers* ($C \triangleleft \alpha$) adjust the outputs of a table. For example, they allow actions to be renamed and their arguments to be modified.

Next, we observe that network behavior is almost never determined by a *single* table; instead, forwarding decisions emerge from *several* tables working together. Consequently, any language for transforming tables needs a way to express how tables *compose and decompose* to form overall behavior. For example, sometimes the output of a table simply flows into the next; other times, two tables impose independent requirements that must both hold; and in many cases, one table takes precedence over another. Match Algebra captures these distinct patterns directly through three combinators: sequential composition ($C_2 \circ C_1$) for staged policies, parallel join ($C_1 \boxtimes C_2$) for independent lookups, and preferential composition ($C_1 \succ C_2$) for controlled fallback.

$C ::= T$	Table Lit
$\mid t$	Table Var
$\mid C \boxtimes C$	Par Join
$\mid C \circ C$	Sequence
$\mid C \succ C$	Preference
$\mid C \triangleleft \alpha$	Range Tfx
$\mid \beta \triangleright C$	Domain Tfx
$\mid C \downharpoonright_\vartheta$	Refinement
$\alpha ::= [a \mapsto a]$	Act Rename
$\mid \beta$	Var Tfx
$\beta ::= x := e$	Write Var
$\mid \pi(X)$	Project Vars
$e ::= b \mid p \mid \cdots$	BitPat Expr
$\vartheta ::= \bot \mid e = e \mid \cdots$	BitPat Form
$x \in$ Var, $t \in$ TVar, $T \in$ Table	
$a \in$ Action, $b \in 2^*$, $p \in \mathbf{s}^*$	

Fig. 7. Syntax of the Match Algebra

Additionally, the Match Algebra gives us a way to decompose tables, both "vertically," that is, by field, and "horizontally," by row. First, we can leverage our domain transformers to specialize tables to a subset of their keys, and range transformers to select a subset of action arguments. For example, if a table T has keys $\overline{x} \cup \overline{y}$ and arguments $\overline{w} \cup \overline{z}$, we can split T "vertically" into two: writing $\pi(\overline{w}) \triangleright T \triangleleft \pi(\overline{w})$ projects table T to keys $\overline{x}$ and arguments $\overline{w}$, and writing $\pi(Y) \triangleright T \triangleleft \pi(\overline{z})$ does so on keys Y and arguments Z. Next, to decompose tables "horizontally," we define a *refinement* operator, $C \downharpoonright_\vartheta$ that restricts C to only be defined for inputs that satisfy ϑ. Then, $T \downharpoonright_\vartheta$ and $T \downharpoonright_{\neg\vartheta}$ give us a total horizontal partition for inputs that satisfy ϑ and those that do not.

Relational vs. Match Algebra. At first glance, the operators in Match Algebra seem similar to those from the relational algebra. For example, both parallel join ($C_1 \boxtimes C_2$) and composition ($C_2 \circ C_1$) are similar to the natural join of two relations ($R_1 \bowtie R_2$). Similarly, refinement ($C \downharpoonright_\vartheta$) is similar to selection $\sigma_\vartheta(R)$; and our projection operator ($\pi(X)$) seems similar to relational projection (π_X). This raises the question: *Do we even need a separate algebra for match-action tables? Can we simply re-purpose relational algebra for data plane portability?*

Unfortunately, it's not that easy: While database tables simply hold data, match-action tables *are code*. Accordingly, Match Algebra gives tables operational rather than relational meaning, capturing how they process packets instead of what input–output pairs they denote. Another way to see this is that, in relational algebra, the intension and the extension of a finite table are equivalent: the table's contents completely determine its meaning. Conversely, the intension of a match-action table is its collection of rules whose priority and guards determine control flow, while the extension is a (possibly partial) function mapping inputs to outputs.

To see the difference, consider the refinement operator. In relational algebra, refining a table T by ϑ simply removes some tuples. Extensionally, Match Algebra behaves the same way: it removes the corresponding input–output pairs from the function denoted by T. Intensionally, however, refinement alters the rule structure of the table itself. For example, refining a single rule $x = ** \mapsto$ allow by the predicate $x \in \{00, 11\}$ produces two rules, $x = 00 \mapsto$ allow and

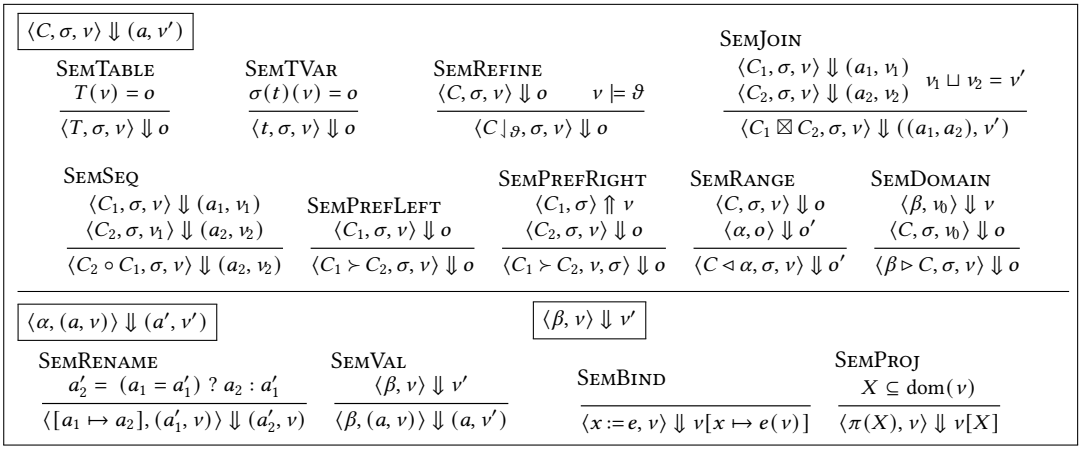

Fig. 8. Match Algebra semantics (top), and transformation semantics (bottom).

$x = 11 \mapsto$ allow. As such the Match Algebra reasons from both perspectives: Section 4.2 describes its extensional semantics, while the compiler in Section 6 constructs the intensional realization.

4.2 Syntax and Semantics

We now synthesize the observations outlined earlier into a clean algebra with declarative operators. The syntax of Match Algebra is shown in Figure 7, and its semantics are described in Figure 8 using judgments of the form $\langle C, \sigma, \nu\rangle \Downarrow o$, where $o = (a, \nu')$ is some action valuation pair. The meaning of this judgment is that an expression C, given configuration σ and valuation ν, yields action a and a new valuation ν'.

Tables are the core primitive of the Match Algebra: the simplest expressions are tables $T \in \mathsf{Table}$ and table variables $t \in \mathsf{TVar}$. Semantically, a table literal T takes in ν and produces $T(\nu)$ if it is defined (Figure 8, SemLit). Similarly, a table variable t looks up t in σ, i.e. $T = \sigma(t)$ (getting stuck if $\sigma(t)$ is undefined), and then returns $T(\nu)$ (if it is defined).

Next, the refine operation $C \lfloor_\vartheta$ specializes C to only be defined on inputs satisfying ϑ: the expression behaves like C on inputs ν that satisfy ϑ, written $\nu \models \vartheta$ (definition elided), and is undefined otherwise. Formally, if $\langle C, \sigma, \nu\rangle \Downarrow (a, \nu')$ and $\vartheta(\nu)$ holds, then $\langle C \lfloor_\vartheta, \sigma, \nu\rangle \Downarrow (a, \nu')$ (Figure 8, SemRefine); if $\nu \not\models \vartheta$ or C is undefined on ν, the result is undefined.

Our first composition operator, *parallel join*, runs the input valuation ν through both C_1 and C_2, to compute (a_1, ν_1) and (a_2, ν_2) respectively. Then, their outputs are merged using the semi-disjoint union operator $\nu_1 \sqcup \nu_2$, which succeeds only if the two valuations agree on all overlapping keys (Figure 8, SemJoin). When the merge succeeds, the result is $((a_1, a_2),\ \nu_1 \sqcup \nu_2)$. If either sub-expression fails or their outputs conflict, the join is undefined.

Sequential composition $C_2 \circ C_1$ passes the output valuation of C_1 into C_2. That is, SemSeq first executes C_1 on ν to obtain (a_1, ν_1), then executes C_2 on the resulting valuation ν_1 to produce (a_2, ν_2).

Preferential union $C_1 \succ C_2$ indicates that the second table should only be consulted when the first is undefined. The challenge in defining a general union operation is that it introduces significant ambiguity: if both C_1 and C_2 are defined, which behavior should the table produce? To avoid this ambiguity, the preferential union operator provides a built-in mechanism to resolve it: simply perform C_1's action. Operationally, $C_1 \succ C_2$ first attempts to evaluate C_1 on input ν. If $\langle C_1, \sigma, \nu\rangle \Downarrow (a, \nu')$, the result is (a, ν') (SemPrefLeft); otherwise, if C_1 is undefined on ν (which we write $\langle C_1, \sigma\rangle \Uparrow \nu$), SemPrefRight evaluates C_2, returning (a, ν') if $\langle C_2, \sigma, \nu\rangle \Downarrow (a, \nu')$.

The Match Algebra also can transform the match conditions of a table via the domain transformation operator ($\beta \triangleright C$). Its mirror, $C \triangleleft \alpha$ transforms table's action-valuation pairs ($C \triangleleft \alpha$). Formally, $x := e$ is evaluated by SemBind, which evaluates e with ν in the standard way, and updates the ν with the result. Next, SemProj, $\pi(X)$ restricts ν to the fields X, written $\nu[X]$. Finally, for action-valuation transforms $[a_1 \mapsto a_2]$, SemRename updates the input action to a_2 iff it is equivalent to a_1.

5 Guarded Functional Type System

The semantics of Match Algebra, as defined in the previous section, are intentionally permissive, allowing *ambiguous* behaviors wherein a single input valuation may correspond to multiple possible actions or output valuations. This ambiguity is not an accident: it arises naturally from the expressive constructs needed to represent real domain transformations. For example, when we forget some a table's keys ($\pi(X) \triangleright T$), we may forget variables that once determined the result. As a consequence, a single input ν may now correspond to multiple outputs $\{(a_1, \nu'_1), \ldots (a_n, \nu'_n)\}$. This is apparent from the SemDomain rule (Figure 8), which can be used to derive an output (a_i, ν'_i) for each element ν_0 in the preimage of $\pi(X)$ with respect to ν.

To make this more concrete, consider a source Termination MAC table which determines whether to "ROUTE" or "BRIDGE" a packet based on its ingress port, EtherType, and VLAN identifier. Figure 9 shows some possible rules installed in this table. Now, suppose we want to migrate these rules to an IPv4-specific target table (i.e. when ethType is 0x800) that does *not* permit matching on the ingress port field. We can attempt to perform this task via the match-algebra expression $Target = \pi(\text{vlanId}) \triangleright Source\downarrow_{\text{ethType==0x800}}$ which projects to the relevant match keys by applying a domain transformer. However, this expression is *ambiguous*: applying the Match Algebra specification does not uniquely determine the resulting rules, and all of the target tables shown in Figure 10 are consistent with its semantics. For example, given a packet with inport = 808 and ethType = 0x800, *Target* permits all of bridging, routing, or nop.

Source

inport	ethType	vlanId	action
808	0x0800	*	goto(nxt = ROUTE)
707	0x8100	4	goto(nxt = BRIDGE)
99	0x0800	*	goto(nxt = BRIDGE)
*	*	*	nop()

Fig. 9. A Termination Mac Table

Target

vlanId	action
*	goto(nxt = BRIDGE)

or

Target

vlanId	action
*	goto(nxt = ROUTE)

or

Target

vlanId	action
*	nop()

Fig. 10. Tables Consistent with $\pi(\text{vlanId}) \triangleright (Source\downarrow_{\text{ethType==0x800}})$

To prevent this form of ambiguity, we equip Match Algebra with a static type system that relies on the notion of *guarded functional dependencies (GFDs)*. At a high level, GFDs capture when a set of keys uniquely determines the output under a particular guard condition. A traditional functional dependency [1], written, for example as inport, ethType, vlanID $\rightarrow$ nxt indicates that knowing the values of inport, ethType, and vlanId suffices to determine the value of nxt. This functional dependency holds for the *Source* table. However, in general, we need to enrich traditional functional dependencies with *guards* to encode assumptions about the source tables, which may be needed for typechecking match algebra expressions that involve refinements.

In the absence of such guarantees, transformations may produce tables in which multiple rules match the same packet but prescribe different actions, which cannot be implemented as a match-action table. GFDs rule out precisely these cases by ensuring that transformed tables remain well-defined functions.

In our example, suppose *Source* satisfied the GFD $\Phi = \{\text{vlanID} \mid \text{ethType} = \text{0x800}\} \rightarrow \{\text{nxt} \mid \top\}$. This means that, when ethType is 0x800, the vlanID variable uniquely determines *Source*'s action

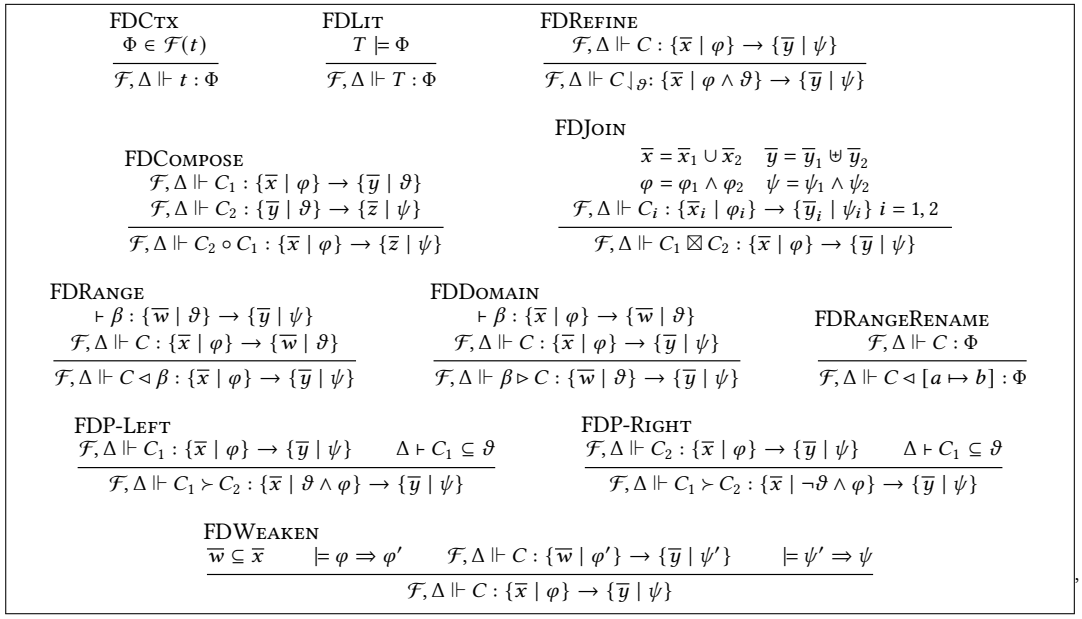

Fig. 11. Guarded functional dependency rules.

output nxt field. Then, $\pi(\mathsf{vlanId}) \triangleright \mathit{Source}\downarrow_{\mathsf{ethType==0x800}}$ has the type $\{\mathsf{vlanId} \mid \top\} \to \{\mathsf{nxt} \mid \top\}$, meaning that it is deterministic. In this example, GFDs tell us precisely when a field is redundant for determining behavior, even if it still appears syntactically in the table.

However, the input rules in Figure 9 do not satisfy Φ. In particular, if vlanId is 99, then the first rule indicates goto(nxt = ROUTE), while the last rule indicates nop(). Thus, the source table does *not* have type $\{\mathsf{vlanID} \mid \mathsf{ethType} = \mathtt{0x800}\} \to \{\mathsf{nxt} \mid \top\}$, and the configuration will be rejected.

5.1 Guarded Functional Dependencies

To formalize our GFD type system, we start with the following definitions:

Definition 1 (Valuation Equivalence). Two valuations ν and ν' are equivalent on $\overline{y}$, written $\nu \equiv_{\overline{y}} \nu'$ iff for each $y \in \overline{y}$, $\nu \downarrow y$, $\nu' \downarrow y$ and $\nu(y) = \nu'(y)$.

Definition 2 (GFD satisfaction). A table T satisfies a GFD $\Phi = \{\overline{x} \mid \varphi\} \to \{\overline{y} \mid \psi\}$, denoted $T \models \Phi$, iff for any valuations $\nu_1, \nu_2 \models \varphi$ such that $\nu_1 \equiv_{\overline{x}} \nu_2$, there exist $(a_i, \nu_i') = T(\nu_i)$ for $i = 1, 2$ such that:

$$\mathrm{dom}(\nu_1') = \overline{y} = \mathrm{dom}(\nu_2') \qquad \nu_1' \models \psi \qquad \nu_2' \models \psi \qquad (a_1, \nu_1') = (a_2, \nu_2').$$

While individual GFDs describe when a single table behaves functionally, Match Algebra expressions manipulate many tables whose behaviors depend on one another. To reason compositionally, the type system must keep track of which dependencies hold for each table and under what conditions those are valid. This bookkeeping is captured by two typing contexts: one that records the functional assumptions we can make about each table, and another that records the domain of each table. Specifically, the *GFD context* $\mathcal{F}$ associates table variables with sets of functional dependencies, and the *domain context* Δ associates table variables with formulae describing (overapproximating) their domain. The following definitions state what sets of configurations these contexts describe:

Definition 3 (GFD Context Satisfaction). A configuration σ satisfies a GFD context $\mathcal{F}$, written $\sigma \models \mathcal{F}$, iff $\sigma(t) \models \Phi$ for all $\Phi \in \mathcal{F}(t)$ and all $t \in \mathrm{dom}(\mathcal{F})$.

Definition 4 (Domain Context Satisfaction). A configuration σ satisfies a domain context Δ, written $\sigma \models \Delta$, iff $\nu \models \Delta(t)$ for all $t \in \mathrm{dom}(\Delta)$ and all ν on which $\sigma(t)$ is defined, i.e. $\sigma(t) \downarrow \nu$.

In essence, these context satisfaction definitions the assumed types remain valid at translation-time: each base table in the configuration actually satisfies the functional dependencies declared in $\mathcal{F}$ and the domain restrictions in Δ. For convenience, when $\sigma \models \mathcal{F}$ and $\sigma \models \Delta$, we write $\sigma \models \mathcal{F}; \Delta$

5.2 The GFD Type System

We now explain the typing rules from Figure 11. The first rule FDCtx looks up the functional dependencies declared in $\mathcal{F}$, and FDLit encodes the assumption that all tables correspond to deterministic functions. FDRefine refines the guard to account for the fact that the semantics of refinement is undefined when the predicate fails. The rule thus derives $\{\overline{x} \mid \varphi \wedge \vartheta\} \to \{\overline{y} \mid \psi\}$, indicating that, as long as the filter passes and the original assumptions of C are met, the same keys determine the same action and outputs. The composition rule FDCompose propagates dependencies through sequential execution by threading them across an intermediate state. If $\mathcal{F}, \Delta \Vdash C_1 : \{\overline{x} \mid \varphi\} \to \{\overline{y} \mid \vartheta\}$ holds, then $\overline{x}$ determines the action of C_1, the intermediate state $\overline{y}$, and ensures that the output satisfies ϑ. To compose it with a second program C_2, we must ensure that the outputs of C_1 are consistent with C_2's inputs, i.e. that C_2 satisfies $\{\overline{y} \mid \vartheta\} \to \{\overline{z} \mid \psi\}$. The resulting type then combines the input type of C_1 and the output type of C_2: that is $C_2 \circ C_1$ satisfies $\{\overline{x} \mid \varphi\} \to \{\overline{y} \mid \psi\}$.

The next rule, called FDJoin, captures how dependencies compose across parallel subexpressions. From $\mathcal{F}, \Delta \Vdash C_1 : \{\overline{x}_1 \mid \varphi_1\} \to \{\overline{y}_1 \mid \psi_1\}$ and $\mathcal{F}, \Delta \Vdash C_2 : \{\overline{x}_2 \mid \psi_2\} \to \{\overline{y}_2 \mid \psi_2\}$ we derive $\mathcal{F}, \Delta \Vdash C_1 \boxtimes C_2 : \{\overline{x}_1 \cup \overline{x}_2 \mid \varphi_1 \wedge \varphi_2\} \to \{\overline{y}_2 \uplus \overline{y}_2 \mid \psi_1 \wedge \psi_2\}$, where $\overline{y}_1 \uplus \overline{y}_2$ is the disjoint union, requiring that $\overline{y}_1 \cap \overline{y}_2 = \emptyset$, which ensures that the disjoint union of the output valuations succeeds. The remainder of the argument for soundness can be summarized as follows: Agreement on $\overline{x}_1 \cup \overline{x}_2$ implies agreement on each component's keys; on inputs satisfying both φ_1 and φ_2, C_1 and C_2 each behave deterministically and their outputs satisfy ψ_1 and ψ_2, so their combined result is deterministic and satisfies $\psi_1 \wedge \psi_2$.

The last combination operator, $C_1 \succ C_2$ has two rules due to the disjunctive nature of this operator. These rules are exactly why we need the domain context Δ. Starting with FDP-Right, we propagate the functional dependency of C_2 through $C_1 \succ C_2$. To do this, we need to be sure that C_2 will be executed instead of C_1. To this end, we utilize a judgment $\Delta \vdash C \subseteq \vartheta$ (defined in the Appendix **??**) indicating that ϑ is a sound over-approximation of the domain of C. In other words, if $\neg\vartheta$ holds, we can be sure that C_2 rather than C_1 will be executed. Thus, FDP-Right propagates the functional dependency of C_2 through $C_1 \succ C_2$ under the additional assumption $\neg\vartheta$. Similarly, the FDP-Left rule propagates the functional dependency of C_1. However, we need to be careful to not accidentally "pick up" extra rows from C_2: It may be the case that φ is weaker than C_1's domain ϑ, which means that any rows of C_2 whose matches intersect with φ could invalidate the functional dependency. To avoid this, we restrict the domain guard to $\varphi \wedge \vartheta$.

Next, we consider the typing rules for transformers. The FDRangeRename rule simply states that renaming the actions leaves the dependency unchanged, since it relabels the action symbol but does not affect which outputs are determined by which keys. The other range operations are defined using FDRange, which invokes an auxiliary judgement for computing the functional dependency induced by α. The auxiliary judgment is unsurprising, and for brevity has been relegated to Appendix **??**. The FDDomain rule works similarly, but composes the judgments on the other side. Finally FDWeaken allows us to weaken a derived GFD. In a single rule, it captures the three ways we can weaken $\{\overline{w} \mid \varphi'\} \to \{\overline{y} \mid \psi'\}$: by strengthening φ, weakening ψ, and adding elements to $\overline{x}$. Note that we cannot remove elements from $\overline{y}$ as in standard FDs as we need to be able to

soundly ensure the disjointness condition in FDJoin. The type-soundness theorem is stated below. The proof can be found in Appendix **??**.

Theorem 5. *Given* $\mathcal{F}, \Delta \Vdash C : \{\overline{x} \mid \varphi\} \rightarrow \{\overline{y} \mid \psi\}$, *and* σ *s.t.* $\sigma \models \mathcal{F}$ *and* $\sigma \models \Delta$, *for all input valuations* $\nu_1, \nu_2 \models \varphi$, *if* $\nu_1 \equiv_{\overline{x}} \nu_2$, *then* $\langle C, \sigma, \nu_i \rangle \Downarrow (a_i, \nu_i')$ *for* $i \in \{1, 2\}$, $a_1 = a_2$ *and* $\nu_1' = \nu_2'$.

The guarantee above ensures that every Match Algebra expression corresponds to a concrete match action table. Next we'll cover the design of a compiler that compute these tables.

6 The Match Algebra Configuration Compiler

With our type system for eliminating ambiguity, every well-typed match algebra expression uniquely defines a target table (up to semantic equivalence) for a given source table. However, in general, there are *many* semantically equivalent target tables. That is, while the specific rule sets may be different, they may have the same observable effect. Thus, what remains to be done is to find a concrete table that belongs to this set — a task that we refer to as *compilation*.

In theory, compilation is easy because the space of relevant valuations is always *finite* (there are finitely many finite-domain variables). Thus, in principle, given σ, we can enumerate each valuation ν, and if there exists (a, ν') such that $\langle C, \sigma, \nu \rangle \Downarrow (a, \nu')$, we can create a row $\langle \nu, a, \nu' \rangle$ in the target table. Because each relevant valuation is a finite tuple of finite-width bitvectors, the refinement and correctness arguments below range over the induced finite valuation space containing the inputs of interest together with the intermediate valuations produced during compilation and execution; we leave this space implicit in the notation. While this naive strategy is sound, the resulting table would be intractably large, as each row corresponds to a single packet. To make this concrete, a simple IPv6 routing table that inspects the 128-bit source and target addresses would require enumerating a packet space of size $2^{128+128} \approx 4 \times 10^{74}$. Thus, the challenge of compilation lies in producing a target table that not only realizes the intended behavior, but is *compact*. Compactness matters in practice for two reasons. First, real targets—from switching ASICs to cloud firewalls—impose strict limits on table size. Second, larger tables directly increase lookup cost (e.g., latency, power consumption), so minimizing rules is essential for efficient execution.

To address this challenge, we leverage a particular domain insight: *that source configurations are already highly compressed*. For instance, even though the domain of ACL_S from Figure 4 has 2^{32} packets, the example rule set shown in Figure 4 comprises only 4 rules. In practice, production tables can have thousands of rules [41], much smaller than the domains they define. Our key insight is to leverage this latent compression to guide compilation in two ways. First, we preserve the compact structure already implicit in the rules whenever possible. Second, rather than inventing new priorities, we maintain the semantic priorities that govern the original tables. In cases where transformations require introducing new rules, these rules are ordered relative to their originating rules. In this way the target table's priorities reflect the priority structure of the source tables.

6.1 Preserving Latent Compression with Monads

At a high level, the compiler tranforms tables one rule at a time. Each input rule is rewritten according to the translation, potentially producing one (or multiple) output rule(s), which are then combined into a new table. This process must preserve rule ordering and respect the semantics of match-action resolution.

Crucially, this rule-wise formulation does not force the compiler to eagerly expand the table into a fully enumerated representation. Instead, transformations are applied only where the semantics of a rule actually change, allowing the compiler to preserve the latent structure of the input table. In this sense, the compiler maintains the inherent compression present in real-world tables, rather than destroying it through unnecessary refinement.

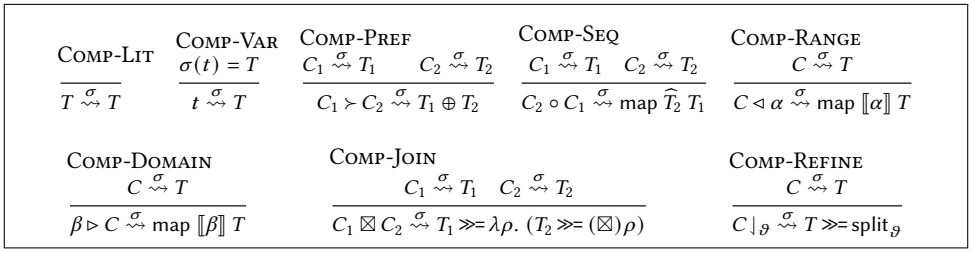

Fig. 12. Compilation procedure for generating target tables from Match Algebra expressions

Since table semantics are defined by selecting the highest-priority enabled rule, preserving and composing order is essential: it determines how conflicts between rules are resolved in the resulting table. Consequently, the design of the compiler hinges on the *extension* operator ($T_1 \oplus T_2$), which logically "stacks" the table computed by T_1 on the table computed by T_2. Letting $T_i = \langle R_i, <_i \rangle$ for $i = 1, 2$, the first step is to simply union the sets of rows $R_1 \cup R_2$, which completely preserves the wildcard structure of R_1 and R_2. Then the new order is the union of $<_1$ and $<_2$, which preserves the relative orders of each table, and the cross product of R_1 and R_2, which ensures that every element of R_1 is preferred to every element of R_2. Formally, $\langle R_1, <_1 \rangle \oplus \langle R_2, <_2 \rangle \triangleq \langle R_1 \cup R_2, <_1 \cup <_2 \cup (R_1 \times R_2) \rangle$.

Now we can define a monad for tables in terms of unit and $\gg\!=$. Then, we define a bind operator $(\gg\!=) : \text{Table} \to (\text{Row} \to \text{Table}) \to \text{Table}$ that composes tables using $\oplus$ to reflect the original order. Informally, $T \gg\!= f$ can be read as follows: for each row in T, apply a transformation f that may produce an ordered collection of output rules, and then combine the results in order using the extension operator. Intuitively, the extension operator provides the composition structure over tables, combining ordered sets of transformed rules while preserving their relative order. Accordingly, $\text{unit} : \text{Row} \to \text{Table}$ produces a table with the empty order. Formally, for $R = \{\rho_1, \rho_2, \ldots, \rho_n\}$, we define:

$$\text{unit}\ \rho \triangleq \langle \rho, \emptyset \rangle \qquad \langle R, < \rangle \gg\!= f \triangleq f(\rho_1) \oplus f(\rho_2) \oplus \cdots \oplus f(\rho_n) \text{ where } \rho_1 < \rho_2 < \cdots < \rho_n$$

By convention, bind is defined "strictly," that is, if any invocation of f is undefined, the entire operation is undefined. Intuitively, this is analogous to the list monad: each input rule produces a sequence of output rules, and bind concatenates these sequences in order. Additionally, we can define an order-preserving map operation $\text{map} : (\text{Row} \to \text{Row}) \to \text{Table} \to \text{Table}$ in the standard way: $\text{map}\ f\ T \triangleq T \gg\!= (\text{unit} \circ f)$.

6.2 The Compiler

The Match Algebra compiler is defined in Figure 12, via the judgment $C \overset{\sigma}{\rightsquigarrow} T$, which means expression C produces table T on configuration σ. The bases cases of compilation are simple: T compiles to itself and t to $\sigma(t)$. When C is preferential union, $C_1 \succ C_2$, Comp-Pref recursively evaluates $C_i \overset{\sigma}{\rightsquigarrow} T_i$ for $i = 1, 2$ and returns $T_1 \oplus T_2$.

The rules for range and domain transformations and composition directly invoke map. For sequential composition, Comp-Seq computes $C_i \overset{\sigma}{\rightsquigarrow} T_i$ for $i = 1, 2$, lifts T_2 to a function on rows (formally $\widehat{T_2}(\rho) = \langle \rho.v, a, v \rangle$, where $(a, v) = T_2(\rho.v)$) mapping it over T_1. As seen in the example from Section 2, map $\widehat{T_2}\ T_1$ preserves the classification logic (guards and priority) of T_1, while employing the corresponding actions of T_2. For a range transformation $C \triangleleft \alpha$, Comp-Range recursively computes $C \overset{\sigma}{\rightsquigarrow} T$, and then uses map to rewrite the rules in T. Specifically, we define the denotation of α as a function on rows, that is $[\![\alpha]\!] : \text{Row} \rightharpoonup \text{Row}$, defined analogously to the semantics in Figure 8. Then

the denotation of $C \triangleleft \alpha$ is map $[\![\alpha]\!]\ T$. Similarly, for domain transformations $\beta \triangleright C$, Comp-Domain computes $C \overset{\sigma}{\rightsquigarrow} T$, and $[\![\beta]\!] : \mathsf{Row} \rightharpoonup \mathsf{Row}$, which changes only the guards, is mapped over T, These are defined formally in Appendix **??**.

To compile join expressions $C_1 \boxtimes C_2$, we need to define an auxiliary operation, a "join" operator on rows, which computes the intersection of the guards, combines the actions using the assumed pairing operation (Section 3), and computes the disjoint union of the output valuations. Formally, we define $\boxtimes : \mathsf{Row} \to \mathsf{Row} \rightharpoonup \mathsf{Table}$ below:

$$\rho_1 \boxtimes \rho_2 \triangleq \begin{cases} \mathsf{unit}\ \langle \mu, (\rho_1.a, \rho_2.a), \nu \rangle, & \text{if } \mu = \rho_1.\mu \wedge \rho_2.\mu, \\ & \text{and } \nu = \rho_1.\nu \sqcup \rho_2.\nu \\ \mathsf{undefined} & \text{elif } \rho_1.\nu \sqcup \rho_2.\nu \text{ undef} \\ \langle \emptyset, \emptyset \rangle & \text{elif } \rho_1.\mu \wedge \rho_2.\mu \text{ undef} \end{cases}$$

$$(\mu \wedge \mu')(x) \triangleq \begin{cases} \mu(x) \wedge \mu(x), & \text{if } x \in \mathrm{dom}(\mu) \cap \mathrm{dom}(\mu') \\ \mu(x), & \text{if } x \in \mathrm{dom}(\mu) \setminus \mathrm{dom}(\mu') \\ \mu'(x), & \text{if } x \in \mathrm{dom}(\mu') \setminus \mathrm{dom}(\mu) \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

where $(\wedge) : \mathsf{Guard} \times \mathsf{Guard} \to \mathsf{Guard}$ computes the symbolic intersection of guards (right), and $(\sqcup) : \mathsf{Val} \times \mathsf{Val} \rightharpoonup \mathsf{Val}$ is the (semi)disjoint union of valuations. With this row-wise join operator in hand, Comp-Join recursively compiles $C_i \overset{\sigma}{\rightsquigarrow} T_i$ for $i = 1, 2$, and then applies $\boxtimes$ over all pairs of rows, ordering them lexicographically. This is captured via the concise monadic expression $T_1 \gg\!= \lambda\rho.\ T_2 \gg\!= (\boxtimes)\rho$. Note that we also take the "strict" semantics here, meaning that, if any $\boxtimes$ operation is undefined, the entire expression is undefined.

For the operators we have considered so far, it was possible to *always* take advantage of the compression already latent in σ. However, this is not the case for refinement, $C \lfloor_\vartheta$. At first glance, refinement appears straightforward: we might compile it by first invoking $C \overset{\sigma}{\rightsquigarrow} T$ and then retaining only those matches that satisfy ϑ. This works when T contains no wildcards. But since each guard μ denotes a set of valuations, refinement requires computing the intersection of μ and ϑ.

The challenge is that this intersection may not correspond to a single representable guard in the target match language. Instead, it may unfold into a collection of guards whose denotations jointly cover the intended region of the match space. For instance, intersecting $\mu = \{x \mapsto *, y \mapsto *\}$ with $x = y$ yields two guards, $\{x \mapsto 0, y \mapsto 0\}$ and $\{x \mapsto 1, y \mapsto 1\}$. The task, then, is to recover a representation that remains semantically equivalent while being as compact as possible. The Comp-Refine rule solves this problem through an abstract operator $\mathsf{split}_\vartheta(\rho) = T$ for which, for every relevant valuation ν, $\langle (\mathsf{unit}\ \ \rho \lfloor_\vartheta), \emptyset, \nu \rangle \Downarrow (a, \nu')$ if and only if $T(\nu) = (a, \nu')$. We apply this operator to the recursively computed T (that is, $C \overset{\sigma}{\rightsquigarrow} T$) and finally obtain $T \gg\!= \mathsf{split}_\vartheta$.

Algorithm 1 GreedySplit$(\rho, \overline{x}, \vartheta)$

Input: row ρ; predicate ϑ
Output: Table T

```
1:  if ρ.μ ⊨ ϑ then return unit ρ
2:  else
3:      T ← ⟨∅, ∅⟩
4:      φ ← enc(ρ.μ) ∧ ϑ
5:      while φ is satisfiable do
6:          ν ← GetModel(φ)
7:          μ* ← argmax_μ ω
8:                subject to ν ⊨ μ,
9:                      and ⊨ enc(μ) ⇒ φ
10:         T ← T ⊕ unit ⟨μ*, ρ.a, ρ.ν⟩
11:         φ ← φ ∧ ¬enc(μ*)
        return T
```

As may be evident, the efficiency of refinement compilation relies on the implementation of split_ϑ. Algorithm 1 presents one concrete instantiation of split_ϑ, using an OMT objective ω to seek a compact guard μ^*. For semantic correctness, however, the proof relies only on the side constraints of the query, namely that $\nu \models \mu^*$ and $\models \mathsf{enc}(\mu^*) \Rightarrow \varphi$; the objective is only used to bias the search toward *compact* tables.

At a high level, Algorithm 1 takes in a row ρ and a predicate ϑ, and returns a table T that matches on the intersection of $\rho.\mu$ and ϑ, and executes $\rho.a$ with valuation $\rho.\nu$. To start, it performs a quick short-circuit check (Line 1), to see whether $\rho.\mu$ denotes a subset of ϑ, and, if so, the algorithm just returns unit ρ (Line 2). Otherwise, we initialize a table T (Line 3), and maintain a symbolic "residual" set φ, which is initialized (Line 4) to the intersection of $\rho.\mu$ and ϑ. We use $\mathsf{enc}(\rho.\mu)$ to indicate the

symbolic encoding of the set $\rho.\mu$ defined in the obvious way. This symbolic set captures the portion of the intersection that is not yet captured by the guards of T. Consequently, the while loop (Line 5) iterates until φ is unsatisfiable, at which point we produce the table T. When φ is not empty, the SMT solver returns a model corresponding to a valuation v. We then solve the OMT query from lines 7–8, which seeks a maximally permissive guard subject to the side constraints $v \models \mu$ and $\models \mathsf{enc}(\mu) \Rightarrow \varphi$. These constraints are the part needed for soundness, while the objective ω is used to bias the search toward compact guards. After computing the resulting guard μ^*, we add the row with this new guard, ρ's action, and ρ's output valuation to the table T (Line 9) before excluding μ^* from the symbolic residual set φ (Line 10).

GreedySplit precisely captures the intersection of $\rho.\mu$ and ϑ, meaning that the disjunction ψ of all guards in T agrees with $\mathsf{enc}(\rho.\mu) \wedge \vartheta$ for every relevant valuation. Termination follows because the current residual space is finite and every iteration removes at least the current witness. Furthermore, every emitted guard is required to stay within the current residual, ensuring that the algorithm never overshoots. Thus, upon termination, the guards in T exactly cover $\mathsf{enc}(\rho.\mu) \wedge \vartheta$. We state this formally below; the proof can be found in Appendix **??**:

Lemma 1. *For any row ρ and predicate ϑ, Algorithm 1 terminates and returns a table T such that*

$$\forall v.\ \langle \mathsf{unit}\ \rho\downarrow_\vartheta, \emptyset, v\rangle \Downarrow (a, v') \iff T(v) = (a, v')$$

6.3 Metatheory

Now that we have a formal definition of the compiler, we can prove type soundness of our functional dependency types with respect to this compiler. Specifically, our compiler correctness theorem can be broken down into two components: For $C \overset{\sigma}{\rightsquigarrow} T$, we first prove (see Appendix **??**) that the T is defined on the same set of valuations as C.

Lemma 2 (Refinement). *Suppose $C \overset{\sigma}{\rightsquigarrow} T$. If $T(v) = (a, v')$, then $\langle C, \sigma, v\rangle \Downarrow (a, v')$.*

To argue the correctness of our compiler, Lemma 2 is not sufficient on its own: it only shows that every compiled behavior is semantically valid. To prove correctness, we rely on our type system and start by proving that compilation preserves GFDs (the full proof is in Appendix **??**):

Lemma 3 (Type-Preservation). *Suppose $\mathcal{F}, \Delta \Vdash C : \Phi$ and $C \overset{\sigma}{\rightsquigarrow} T$. If $\sigma \models \mathcal{F}; \Delta$, then $T \models \Phi$.*

We now state our main theorem: functional dependencies ensure compiler correctness, to do so, we need an auxiliary definition, which is when a table T is equivalent to $\langle C, \sigma\rangle$ "modulo" a GFD.

Definition 6 (Equivalence Modulo GFD). An expression C and configuration σ are equivalent to a table T modulo a GFD Φ, written $\Phi \models \langle C, \sigma\rangle \equiv T$, when they satisfy the following property, where $\Phi = \{\overline{x} \mid \varphi\} \rightarrow \{\overline{y} \mid \psi\}$: for all valuations v_1, v_2 and outputs (a_1, v_1'), (a_2, v_2'), if $v_1, v_2 \models \varphi$, $v_1 \equiv_{\overline{x}} v_2$, $\langle C, \sigma, v_1\rangle \Downarrow (a_1, v_1')$, and $T(v_2) = (a_2, v_2')$, then $a_1 = a_2$ and $v_1' = v_2'$.

Then our correctness theorem is a consequence of compiler refinement and type preservation.

Theorem 7 (Correctness). *If $\mathcal{F}, \Delta \Vdash C : \Phi$, $\sigma \models \mathcal{F}; \Delta$, and $C \overset{\sigma}{\rightsquigarrow} T$, then $\Phi \models \langle C, \sigma\rangle \equiv T$.*

Proof. By Lemmas 3 and 2. □

An immediate corollary is that if the match refinements are trivial, then the packets are equivalent. We write this as follows:

Corollary 8. *If $\Vdash C : \{\overline{x} \mid \top\} \rightarrow \{\overline{y} \mid \psi\}$, and $C \overset{\sigma}{\rightsquigarrow} T$, $\langle C, \sigma, v\rangle \Downarrow (a, v') \iff T(v) = (a, v')$*

This corollary is a specialization of Theorem 7, expressing the common desired theorem: an unconditionally well-typed programs compiles all configurations correctly.

7 Implementation

MatchBox is implemented in OCaml and uses Z3 as its SMT and OMT back end. The implementation provides a surface language, called MatchStix, that desugars to the core Match Algebra. The surface syntax provides an ASCII-friendly set of operators, for instance `C1 * C2` to represent $C_1 \boxtimes C_2$, `C1 >> C2` to represent $C_2 \circ C_1$. Additionally, MatchStix adopts a sequential-composition structure for transformations; that is, we overload `>>` to additionally mean $\triangleright$ and $\triangleleft$, with additional keywords (`rename`, `key`, `data`) to disambiguate. Finally, tables are defined using the "matchstick" operator `o--`. For instance, the example from Section 2 can be seen in Figure 13.

Under this saccharine surface, MatchStix provides a type system for reasoning about the structural well-formedness of tables, that is, their keys, actions, and action arguments (e.g. `RouteS` has one 32-bit key (`dst`), and one action (`route`), with a 9-bit argument named `port`). Tables that are not given matchstick definitions are *declared*, as `RouteS` and `ACLS`.

In MatchStix, the GFDs are typically implicit. For instance, in Figure 13, the GFDs are deduced from the types: e.g. `RouteS` is presumed to have GFD $\{\texttt{dst} \mid \top\} \rightarrow \{\texttt{port} \mid \top\}$. If additional GFDs or domain constraints are needed, they can be assumed: e.g., to (redundantly) assume `RouteS`'s GFD, write `RouteS : {dst | true} ---> {port | true}`.

The main hiccup in implementing the declarative GFD type system is nondeterminism. In particular, both FDP-Left and FDP-Right may apply for $C_1 \succ C_2$ (see Section 5.2). Our implementation modifies each rule to produce the complete set of GFDs derivable from the program. For a program C, and assumption $\mathcal{F}$ and Δ, this construction produces a finite set of GFDs $\{\Phi_1, \ldots, \Phi_n\}$ such that $\mathcal{F}; \Delta \Vdash C : \Phi_i$ for each i. Then for each matchstick `t o-- C`, we check whether there is some Φ_i that can be weakened to derive $\mathcal{F}(t)$.

```
RouteS (dst : 32) [route miss] {port : 9}.
ACLS (dst : 32) [allow deny] {}.
RouteT (dst : 32) [route miss] {port : 9}
  o-- RouteS.
ACLT (dst : 32) [allow deny] {}
  o-- (RouteS * ACLS) >>
      rename route;allow to allow >>
      rename route;deny to deny >>
      rename miss;allow to deny >>
      rename miss;deny to deny >>
      project data ().
```

Fig. 13. MatchStix example

The final piece of MatchBox is the configuration compiler. We represent the strict, totally ordered rule-sets as lists and leverage the list monad [14]. The challenge is implementing the guard generalization step of the GreedySplit algorithm, which in Section 6 we represented abstractly using argmax. In practice, we implement this argmax using Z3's optimization modulo theories (OMT) solver. In particular, we encode the guard μ as a conjunction of equations $x \,\&\, m_x = v_x \,\&\, m_x$ between each $x \in \mathrm{dom}(\mu)$ (where & is bit-wise and) and two "holes", v_x and m_x, which characterize the match value and the mask respectively. We reconstruct the bitpattern as follows: if the ith bit of m_x is zero, then the ith position of the corresponding bitpattern is $*$, otherwise it's the ith bit of v_x. As such, we can maximally generalize the bitpattern by minimizing each bit of m_x.

8 Case Studies

We evaluate MatchBox in three representative environments where network programs need to evolve over time to adapt to changing environments. In particular, we consider (1) programmable switches, (2) multi-cloud firewalls, and (3) eBPF programs.

8.1 Programmable Switch Evolution

Modern data centers increasingly rely on *programmable switches*, which are specialized hardware devices that executes packet-forwarding logic defined by software rather than fixed-function ASICs. Languages such as P4 [7], NPL [9], and Lyra [21] let engineers specify how packets are matched and what actions are applied, imbuing the data plane with the agility of software development.

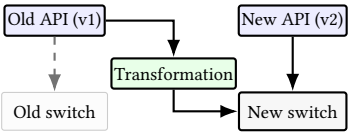

Fig. 14. Pipeline Evolution

This flexibility, however, comes at a cost. As engineers program and re-program switches to (e.g.) improve the clarity of interfaces, add features, or adapt to new performance requirements, the control software that manipulates these switches must evolve too. Small design changes in the switch interfaces can ripple outward, breaking configuration interfaces or invalidating controller assumptions[3]. To mitigate this, network engineers at Google [46] provide an evolving, versioned set of switch interfaces and compatibility functions that simulate the old interfaces on the new one (as in Figure 14). In this case study, we explore using MatchStix to write such transformation functions.

To understand how useful MatchBox is in this context of switch evolution, we source a small set of switch programs (written in P4 [7]) from prior work [10], comprising six realistic Ethernet/IPv4 programs. This set of programs is representative of common forms of refactoring observed by operators [10]. Each variant is capable of implementing the same forwarding behavior, but differs in table organization, match structure, and/or action semantics. These six programs yield 30 source-target pairs. For each scenario, we ask how naturally the transformation can be expressed in MatchStix, how many annotations it takes to pass the GFD type checker, and how efficiently the compiler computes the target tables.

Table 1. Summary of results for P4 programs.

Metric	Min	Median	Max
AST Size	3	20.5	47
Annotation Count	0	0	3
Typecheck Time (μs)	4	16	742

Results. For all 30 scenarios, we were able to automate the desired transformation by writing MatchStix programs, with Table 1 showing key statistics. In particular, the Match Algebra specifications are concise, ranging from 3 to 47 AST nodes with a median of 20.5 nodes, and are efficient to type-check, requiring no more than 1 ms. They also rarely require GFD type annotations ("Annotation Count"); among the 30 programs, only 10 (i.e., 33%) of them require explicit type annotations, and none of them requires more than 3. These 10 MatchStix programs require GFD annotations, because they compile from more expressive set of tables to less expressive ones (as we illustrated in Section 5.1). Appendix **??** shows one such program.

We next evaluate the effectiveness of MatchBox's compiler by running it on 10 input configurations of various sizes ranging from 6 to 10,517 for each translation. Figure 15 plots compilation time against the size of the input configuration. We group the data points by their source pipelines; observe that translating from the action_decompose and link_aggregation pipelines is slower than from the others, as these programs decompose their logic into separate tables, requiring the expensive parallel join (⊠) and composition (∘) operators. Nevertheless, compilation is fast, taking an average of 0.6 ms per input rule.

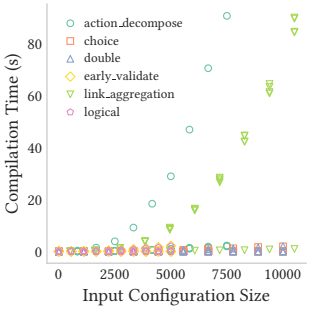

Fig. 15. Evolution Runtimes

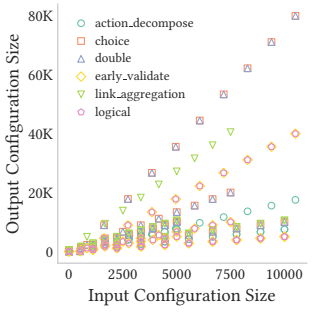

Fig. 16. Evolution Config size

Finally, Figure 16 plots the output configuration size against the input configuration size, grouped by *target* pipeline. Observe that the MatchBox compiler does not explode the number of rules in

[3]For a real-world example, see the ONOS engineers' discussion of their troubles with Qumran-MX (Avenir [10], §1, ¶3)

the output, with configuration size increasing roughly linearly. When larger outputs occur, it is because rules are (necessarily) duplicated across multiple tables.

8.2 Consistent Multi-Cloud Firewalls

Table 2. Summary of results for Cloud programs.

Metric	Min	Median	Max
AST Size	2	9	10
Annotation Count	0	0	2
Typecheck Time (μs)	2	8	15

Multi-cloud deployments are increasingly quotidian: organizations distribute workloads across various cloud providers, e.g. AWS, Azure, and GCP, to balance reliability and cost. Each platform exposes distinct firewall abstractions that differ in schema and semantics, making policy translation necessary. We use MatchBox to express these translations, implementing consistent access control across heterogeneous clouds.

To evaluate MatchBox's support for such scenarios, we modeled the firewall semantics of three major cloud providers (AWS, Azure, and Google Cloud) based on their publicly available documentation [3, 22, 40]. In each case, the provider's firewall interface can be represented as a (collection of) match-action table(s): each rule matches on packet fields such as source and destination IPs, ports, and protocols, while the corresponding action (i.e., allow, deny) determines how traffic is handled. However, each provider has subtle differences in their interfaces or semantics: for instance, AWS maintains distinct inbound and outbound tables [3], whereas Azure and Google Cloud use a single table with direction encoded as a match field [22, 40]. In total, we consider the six possible translations among the three provider models and evaluate MatchBox's ability to specify, typecheck, and compile these transformations efficiently.

Results. For this domain, the six MatchStix programs are quite concise, using between 2 and 10 AST nodes. All six transformations pass MatchBox's GFD type checker within 15 μs, and with a median of 8 μs. Only two require explicit GFDs (see Appendix **??**): translating from Google or Azure to AWS involves projecting and filtering a single-table firewall into two smaller ones for inbound and outbound traffic based on the `is_inbound`. Again, we see that more-expressive sets of tables require GFD annotations when translated to less-expressive ones.

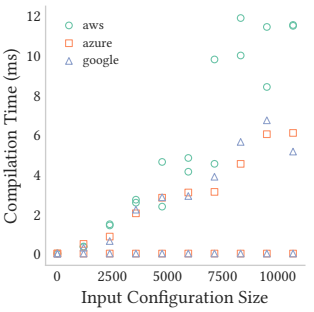

Fig. 17. Cloud Runtimes

Finally, to assess compiler effectiveness, we ran each program on 10 randomly generated (using Classbench [38], the canonical ACL benchmarking tool) input configurations for each translation, of size up to 10,695, which is larger than typical firewalls [3, 22, 40]. Figure 17 plots compilation time against input configuration size, with colored marker shapes for each *source* pipeline. Compilation is fast; it seemingly scales linearly with input size. Finally, the output size is nearly identical to the input size (figure omitted).

8.3 Architectural Changes in eBPF Programs

In this final case study, we examine the evolution of eBPF-based packet-processing programs. eBPF (extended Berkeley Packet Filter) provides a safe, in-kernel execution environment that lets operators attach custom logic to networking hooks and run it at near line rate without modifying kernel code or deploying kernel modules. Modern host networking stacks (e.g., Cilium [15] and Katran [5]) use eBPF to enforce access-control policies while routing traffic and balancing workloads across backend services. Although eBPF programs do not natively expose match-action tables, they rely on shared-memory finite maps to make configurable decisions. Each map associates keys (e.g., packet headers, connection identifiers) with values that represent lookup results or state. As such, these maps can be viewed as specific versions of match-action tables, that always select the unit action ε, with arguments encoding the output value.

Table 3. Summary of results for eBPF translations.

Metric	Min	Median	Max
AST Size	64	79.5	95
Annotation Count	0	16	32
Typecheck Time (μs)	27	663	3811

Maps in eBPF make for a good test-bed for MatchBox, since they frequently evolve as developers optimize for performance. For instance, as eBPF programs handle growing numbers of cores and concurrent packets, contention on shared maps can degrade throughput due to locking. To mitigate this, developers often migrate their code to use per-CPU maps, which shard state across cores to eliminate synchronization overhead. Here, we must change the structure of the shared maps while preserving logical behavior.

We evaluate how MatchBox can provide a shared-memory abstraction over lock-free per-CPU memory structure for firewalls, rate limiting, and routing. In eBPF, we write programs with *system-wide* configurations, which maintain a single shared table accessed by all CPUs, and programs for *per-CPU* configuration, which distribute the logic in the shared table across N CPUs (with $N = 32$). To explore MatchBox's use in this scenario, we write 6 MatchStix programs between the corresponding 3 pairs of eBPF programs: for a firewall, rate limiter, and router.

Results. The six MatchStix programs are larger than in the previous case studies (Table 3), with a median of 79.5 and a max of 95 AST nodes. Additionally, the translations from the system-wide architectures to the per-CPU architectures use refinement and projection to construct each per-CPU maps, requiring one GFD for each CPU (see Appendix **??**). Accordingly, checking these programs is more expensive (median = 663 μs, max = 3.8 ms).

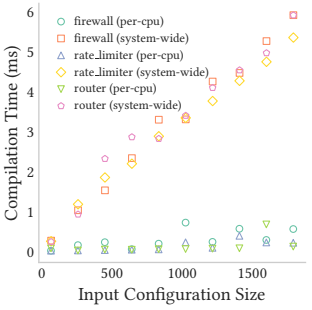

Fig. 18. eBPF Runtimes

Finally, we evaluate performance on 10 input configurations of various sizes ranging from 64 to 1,793 rules, again generated with the help of Classbench [38]. Figure 18 plots runtime against input size, with colored marker shapes indicating the *source* pipeline. Overall, compilation is fast, taking under 2 μs per rule on average. As expected, translations *from* the system-wide setup are slower than translations *to* the system-wide setup. Indeed, partitioning the system-wide map uses expensive refinement and projection, while the reverse requires only cheap preferential union.

9 Related Work

Match-Action Compilation and Synthesis. MatchBox builds on a rich history of research on match-action table compilation [4, 23, 24, 26, 35]. In particular, CoVisor [26] defined an early set of composition primitives for only OpenFlow tables. Further, languages such as NetKAT [4, 24] (rooted in Kleene Algebra with Tests [30]) and *ndlog* [36] (rooted in Datalog) provide declarative frameworks for specifying network-wide policies that are compiled into fixed match-action pipelines. However, these systems assume a static mapping between high-level policies and target table structures. In contrast, MatchBox is designed for *portability*: transforming configurations between environments with differing structure and semantics.

MatchBox's compiler bears similarities to NetGen [48] and Avenir [10], which use solver-aided techniques to generate configurations. In particular, NetGen synthesizes minimal edits to an existing network configuration so that the new data plane satisfies a revised policy, while Avenir compiles a configured abstract pipeline specification to a concrete hardware target by searching for rules via counter-example guided synthesis. In contrast, MatchBox provides a semantic abstraction for translating *between* match-action tables, guided by a declarative specification in the Match Algebra.

Match-Action Table Analysis. Header Space Analysis (HSA), NetKAT [4], Network Optimized Datalog [37], and Flow Algebra [32] leverage algebraic structure and with symbolic optimizations

for reasoning about reachability in networks. However, these approaches treat tables as input-output functions on packet state, requiring a non-trivial action modeling step. In contrast, the Match Algebra explicitly models actions and their arguments, allowing for explicit table transformations.

Type Systems. Prior work extended Haskell's type classes with functional dependencies [27], which constrain type parameters eliminate ambiguity during type inference. However since our GFD type system reasons about program behavior rather than type-level structure, it augments traditional functional dependencies with guards: logical refinements that constrain when a dependency must hold. Here, we take inspiration from refinement type systems [19] and, in particular, Liquid Types [47], which integrate logical predicates into typing to prove semantic properties.

In the context of match-action tables, Petr4 [17], P4Cub [45], and HOL4P4 [2] establish formal semantics and simple type systems for individual P4 programs [6], but do not consider porting between match-action tables. P4R-Type types the P4Runtime match-action interface [31]. Google provides a mechanism for expressing structural constraints on tables, used at Google [44], which can be seen as an early, intensional, and non-compositional precursor to our GFD system.

Semantic constraints on tables have also been studied in the context of data plane verification. In particular, p4v introduced the symbolic control-plane interface, a series of assumptions about the contents of tables [34]. Subsequent work [11, 18] provides mechanisms for inferring properties on tables. GFDs can be viewed as specifications on table contents; however, as hyperproperties [16], their inference is beyond the scope of these tools.

10 Conclusion & Future Work

Porting match-action tables across heterogeneous environments is difficult because packet-processing environments have distinct syntax, structure, and semantics. This paper presents MatchBox, a principled framework for porting tables across environments. At the core of MatchBox is the Match Algebra, a declarative language for match-action table transformations. Its GFD-based type system guarantees that well-typed specifications denote a single realizable behavior, and its compiler synthesizes compact target tables that are faithful to that specification. Our case studies across programmable switches, cloud firewalls, and eBPF show that MatchBox efficiently facilitates portability across the network stack.

Several directions remain for future work. First, although our compiler currently regenerates target tables from scratch, its operator structure suggests a natural path to incremental compilation: local changes to source rules should induce correspondingly local changes in the output. Second, while MatchBox guarantees that a specification is unambiguous and realizable, it does not by itself establish that a given translation preserves the intended end-to-end semantics between source and target environments. Integrating MatchBox with semantic verification tools, and perhaps using such tools to synthesize translations, is a promising direction for future work.

Data-Availability Statement

MatchBox is available on Zenodo [13] and GitHub [12].

Acknowledgments

This work was completed in a research group supported by NSF awards CCF-1918889, CNS-2120696, CCF-2210831, CCF-2319471, CCF-2422130, CCF-2403211, CCF-2505865, CCF-2326576, CCF-2403211, and CNS-2214015, as well as a DARPA award under agreement HR00112590133 and a gift from Amazon.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Anoud Alshnakat, Didrik Lundberg, Roberto Guanciale, and Mads Dam. 2024. HOL4P4: mechanized small-step semantics for P4. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 223–249.

[3] Amazon. [n. d.]. https://docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.

[5] Engineering at Meta. 2025. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95. doi:10.1145/2656877.2656890

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. doi:10.1145/2656877.2656890

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013*, Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan (Eds.). ACM, 99–110. doi:10.1145/2486001.2486011

[9] Broadcomm. 2025. Network Programming Language Specification v1.5.1.

[10] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, James Mickens and Renata Teixeira (Eds.). USENIX Association, 133–153. https://www.usenix.org/conference/nsdi21/presentation/campbell

[11] Eric Hayden Campbell, Hossein Hojjat, and Nate Foster. 2024. Computing Precise Control Interface Specifications. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 905–934.

[12] Eric Hayden Campbell, Robert Zhang, and Divyanshu Saxena. 2026. Matchbox. https://github.com/utopia-group/matchbox

[13] Eric Hayden Campbell, Robert Zhang, Divyanshu Saxena, Aditya Akella, and Isil Dillig. 2026. *Software Artifact for "MatchBox: A Semantic Foundation for Data Plane Portability"*. doi:10.5281/zenodo.19082812

[14] Jane Street Capital. [n. d.]. Module base.list. https://ocaml.janestreet.com/ocaml-core/v0.12/doc/base/Base/List/index.html

[15] Cilium. 2025. Cloud Native, eBPF-based Networking, Observability, and Security. https://cilium.io/.

[16] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[17] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: formal foundations for P4 data planes. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. doi:10.1145/3434322

[18] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 571–585.

[19] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

[20] Michael Galles and Francis Matus. 2021. Pensando Distributed Services Architecture. *IEEE Micro* 41, 2 (2021), 43–49. doi:10.1109/MM.2021.3058560

[21] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 435–450. doi:10.1145/3387514.3405879

[22] Google. [n. d.]. https://docs.cloud.google.com/firewall/docs/firewalls

[23] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Formal foundations for software defined networks. *Open Net Summit* (2013).

[24] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified network controllers. *Acm Sigplan Notices* 48, 6 (2013), 483–494.

[25] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 193–207. https://www.usenix.org/conference/nsdi22/presentation/hogan

[26] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 87–101. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jin

[27] Mark P Jones. 2000. Type classes with functional dependencies. In *European Symposium on Programming*. Springer, 230–244.

[28] Bjarni Jonnson and Alfred Tarski. 1952. Boolean algebras with operators. *American journal of mathematics* 74, 1 (1952), 127–162.

[29] Maria Korolov. 2025. Internet upgrade part of a move towards 400/800G connectivity. *NetworkWorld* (10 Feb. 2025). https://www.networkworld.com/article/3821050/internet-upgrade-upgrade-part-of-a-move-towards-400g-and-800g-connectivity.html Accessed: 2025-09-15.

[30] Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443.

[31] Jens Kanstrup Larsen, Roberto Guanciale, Philipp Haller, and Alceste Scalas. 2023. P4R-Type: a Verified API for P4 Control Plane Programs (Technical Report). *arXiv preprint arXiv:2309.03566* (2023).

[32] Christopher Leet, Robert Soulé, Yang Richard Yang, and Ying Zhang. 2021. Flow Algebra: Towards an Efficient, Unifying Framework for Network Management Tasks. In *40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021*. IEEE, 1–10. doi:10.1109/INFOCOM42981.2021.9488857

[33] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 243–259. https://www.usenix.org/conference/osdi20/presentation/lin

[34] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*. 490–503.

[35] Boon Thau Loo, Joseph M Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative routing: extensible routing with declarative queries. *ACM SIGCOMM Computer Communication Review* 35, 4 (2005), 289–300.

[36] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005. Declarative routing: extensible routing with declarative queries. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, Roch Guérin, Ramesh Govindan, and Greg Minshall (Eds.). ACM, 289–300. doi:10.1145/1080091.1080126

[37] Nuno P. Lopes, Nikolaj S. Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

[38] Jiří Matoušek, Gianni Antichi, Adam Lučanskỳ, Andrew W Moore, and Jan Kořenek. 2017. Classbench-ng: Recasting classbench after a decade of network evolution. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 204–216.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. 38, 2 (March 2008), 69–74. doi:10.1145/1355734.1355746

[40] Microsoft. [n. d.]. Azure Network Security Groups Overview. https://learn.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview

[41] Juniper Networks. [n. d.]. Layer 2 forwarding tables. https://www.juniper.net/documentation/us/en/software/junos/multicast-l2/topics/topic-map/layer-2-forwarding-tables.html

[42] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. 1999. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. 151–162.

[43] Open Compute Project. 2015. Switch Abstraction Interface. https://www.opencompute.org/projects/sai

[44] p4lang. [n. d.]. P4lang/P4-constraints: Constraints on P4 objects enforced at runtime. https://github.com/p4lang/p4-constraints

[45] Rudy Peterson, Eric Hayden Campbell, John Chen, Natalie Isak, Calvin Shyu, Ryan Doenges, Parisa Ataei, and Nate Foster. 2023. P4Cub: a little language for big routers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 303–319.

[46] Victor Rios. 2023. Babel: The Tower so Far. (Oct. 2023). P4 Workshop Poster. Available at https://p4.org/wp-content/uploads/2024/10/2024-P4-WS-Babel-Poster.pdf.
[47] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 159–169.
[48] Shambwaditya Saha, Santhosh Prabhu, and P Madhusudan. 2015. NetGen: Synthesizing data-plane configurations for network policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 1–6.
[49] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/2934872.2934900
[50] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: a language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 731–747. doi:10.1145/3452296.3472903
[51] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 213–226. doi:10.1145/2674005.2674989
[52] Titania. 2022. New Report Reveals Exploitable Network Misconfigurations Cost Organizations 9 Percent of Total Annual Revenue. https://www.titania.com/about-us/news/new-report-reveals-exploitable-network-misconfigurations-cost-organizations-9-percent-of-total-annual-revenue. Survey of 160 senior cybersecurity decision-makers across U.S. Military, Federal Government, Oil & Gas, Telecoms, and Financial Services; conducted by Coleman Parkes Research.
[53] Titania. 2023. Network Security Impact Report. https://www.titania.com/network-security-impact-report. Cross-sector survey finding exploitable misconfigurations cost organizations an average of 9% of annual revenue.
[54] Avishai Wool. 2009. Trends in Firewall Configuration Errors: Measuring the Holes in Swiss Cheese. In *IEEE Computer*. Large-scale study of firewall rule-sets showing frequent misconfigurations, including rule-ordering and translation errors across devices.

Received 2025-11-13; accepted 2026-04-03