Flash: Fast, Consistent Data Plane Verification for Large-Scale Network Settings

Dong Guo
Tongji University

Shenshen Chen
Tongji University

Kai Gao
Sichuan University

Qiao Xiang
Xiamen University

Ying Zhang
Meta Inc.

Y. Richard Yang
Yale University

ABSTRACT

Data plane verification can be an important technique to reduce network disruptions, and researchers have recently made significant progress in achieving fast data plane verification. However, as we apply existing data plane verification techniques to large-scale networks, two problems appear due to extremes. First, existing techniques cannot handle too-fast arrivals, which we call *update storms*, when a large number of data plane updates must be processed in a short time. Second, existing techniques cannot handle well too-slow arrivals, which we call *long-tail update arrivals*, when the updates from a number of switches take a long time to arrive.

This paper presents Flash, a novel system that achieves fast, consistent data plane verification when update arrivals can include update storms, long-tail update arrivals, or both. In particular, Flash introduces a novel technique called *fast inverse model transformation* to swiftly transform a large block of rule updates to a block of *conflict-free updates* to efficiently handle update storms. Flash also introduces *consistent, efficient, early detection*, a systematic mechanism and associated novel algorithms to detect data plane violations with incomplete information, to avoid being delayed by long-tail arrivals. We fully implement Flash and conduct extensive evaluations under various settings. Using the data plane of a large-scale network, we show that compared with state-of-the-art sequential per-update verification systems, Flash is 9,000× faster.

CCS CONCEPTS

• **Networks** → **Network reliability**; *Network monitoring*; • **Theory of computation** → **Logic and verification**.

KEYWORDS

Network Verification; Network Reliability; Network Monitoring

ACM Reference Format:
Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y. Richard Yang. 2022. Flash: Fast, Consistent Data Plane Verification for Large-Scale Network Settings. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3544216.3544246

Dong Guo and Shenshen Chen contribute equally.
Kai Gao and Qiao Xiang are co-corresponding authors.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544246

1 INTRODUCTION

Network faults such as forwarding loops, blackholes and access control violations are prevalent in large-scale computer networks, and can lead to disastrous financial and social consequences [1–3]. Thus, how to prevent network faults is a fundamental problem in the networking community.

A major advance is through network verification, which automatically checks network for errors in both control plane [4–14] and data plane [15–26]. In particular, we focus on the data plane verification in this paper, which checks the data plane and can find errors with a broad range of root causes.

There has been a long line of research on data plane verification [15–27]. Earlier tools develop verification algorithms on top of flow tables (*forward model*) (*e.g.*, [15, 17, 18, 27, 28]). Their performance is limited (*e.g.*, hundreds of milliseconds to seconds for a rule update) because flow table is not an efficient data representation for verification. As such, recent data plane verification tools introduce equivalent classes (*inverse model*) as the data representation for verification [21–26]. Although the number of equivalent classes can be exponential of network size, it is small in practice [21]. By developing efficient algorithms that transform the forward model to the inverse model, these tools achieve substantial speed up on verification. For example, the state-of-the-art APKeep [26] achieves a latency of tens of μs per rule update.

Motivated by the aforementioned advance, we start to construct a data plane verification tool, using the state-of-the-art, for a large production network, and evaluate the tool in a wide range of settings, including both online settings (*e.g.*, fast monitoring [29, 30]), and offline settings (*e.g.*, validation of FIBs derived from simulation [6, 31, 32]). The evaluations reveal that the state-of-the-art data plane verification techniques fall short in handling two problems, due to extremes which tend to appear in large-scale systems.

Handling Update Storms. First, existing techniques cannot handle too-fast data plane update arrivals, which we call *update storms*, when a large number of data plane updates should be processed in a short interval. Update storms can happen in both online settings and offline settings. In an online setting, for example, during a greenfield deployment or a major event (*e.g.*, disruption or recovery), a large number of switches can update their data planes, resulting in an aggregated large number of data plane updates (See Appendix A for examples). In an offline setting, state-of-the-art network simulation tools such as FastPlane [32] can compute the FIB of a network with more than 2,000 routers in a few hundreds of seconds, and the

scale of the RIB and FIB entries can be up to hundreds of millions. To conduct design searches, one may run simulations many times, and evaluate the result of each run quickly.

Designed to handle relatively smooth, limited-rate update arrivals, existing techniques process updates one-by-one and do not perform well. We have evaluated straightforward optimizations such as batch processing, in which multiple updates are sorted by switch, by IP address, or by operation type (*e.g.*, add/delete). We have also applied divide-and-conquer [19] and partitioned a dataset of 6 million FIB updates in a Fabric topology with ~6,000 switches into 112 partitions, where each partition uses the state-of-the-art model-based data plane verifier [26], the system still takes tens of minutes to hours to complete the verification.

Handling Long-Tail Arrival. Second, existing mechanisms cannot handle too-slow arrivals, which we call *long-tail update arrivals*, when the updates from a number of switches can take a long time to arrive. Long-tail arrivals can happen for multiple reasons, such as FIB computation crashed or dampening [33], update packets loss due to incast congestion [34]. When update storms and long-tail arrivals happen together, the arrivals of data plane updates at the verification system may exhibit a bimodel pattern: an initial large update storm followed by additional updates that slowly trickle in from switches whose updates are delayed.

Existing data plane verification systems are designed with complete knowledge of the data plane, and hence they either wait until the full information is obtained, which can result in a long delay, or proceed with transient, potentially inconsistent information, which can result in false decisions. This issue is already reported by previous studies [7, 26], which use timeout as a compromise.

This paper presents Flash, a fast data plane verification system for networks where data plane updates can trigger update storms, long-tail update arrivals, or both. In particular, Flash contributes two new techniques to advance the state of the art on data plane verification: *fast inverse model transformation* (Fast IMT) to handle update storms, and *consistent, efficient early detection* (CE2D) to make fast, consistent decisions despite long-tail arrivals. Flash is a single system that integrates both techniques.

Fast IMT To Handle Update Storms (§3). We make a key observation that processing updates sequentially causes redundant or unnecessary computation and is inefficient when handling update storms. However, it is difficult to identify computation that can be aggregated across native rule updates, because a single rule update can be complex and result in a large number of operations. Fast IMT first uses an efficient, merge-based algorithm, to decompose a large block of native updates into a set of composable updates, which we call *atomic conflict-free overwrites*. It then applies two aggregation operators, first on actions, and then on predicates, to generate compact conflict-free overwrites, which are applied to update the inverse model. Given its similarity to the map-reduce framework, we call Fast IMT the novel map-reduce2 (MR2) algorithm. Combining MR2 with engineering efforts including *subspace partition*, *fast look-up of overlapping rules*, and *persistent action tree*, Fast IMT substantially reduces computation overhead for high scalability.

Consistent, Efficient, Early Detection to Handle Long-Tail Arrivals (§4). We make a key observation that network errors can be identified without collecting the complete data plane, and term

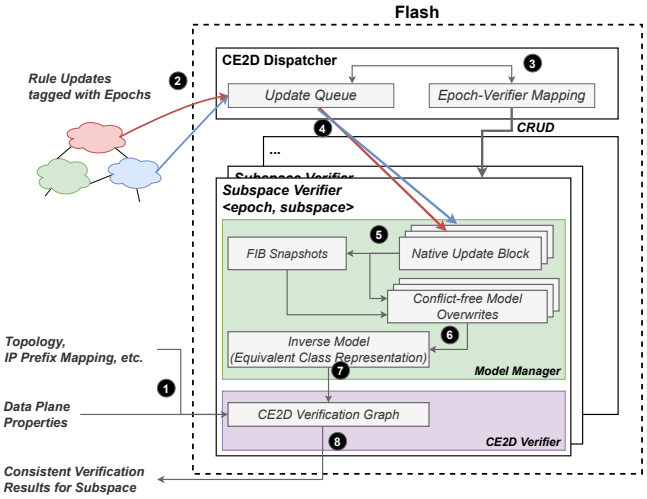


Figure 1: Architecture and workflow of Flash.

this process *early detection*. For example, a forwarding loop detected between two devices will exist regardless of how any other device forwards the packet. Therefore, this error can be reported without the FIB information of other devices, as long as the two devices are in a converged state (*i.e.*, not updating their FIB unless the network state changes). Flash is built on top of this observation, and introduces a systematic mechanism to ensure that Flash *always performs early detection on a consistent network state*. Specifically, Flash uses *epochs* to differentiate FIB updates computed from different network states, and applies early detection to the model constructed for *synchronized* devices in an epoch, *i.e.*, whose FIB is computed from the same network state. Flash captures the partial order of epochs for efficient scheduling. We then leverage automata theory to develop novel algorithms to achieve efficient early detection.

Results (§5). We implement Flash in this research and release it as an open-source software[1] (see details of the software in §5.1 and Appendix F). We conduct extensive evaluations of Flash with a wide range of topologies, data planes, update arrival patterns and policies. Using the data plane of a large-scale network (LNet), we show that Flash is 9,000× faster than state-of-art. Even if state-of-the-art tools integrate subspace partition, Flash is still over 70× faster. Flash achieves the preceding results by introducing both efficient algorithms (*i.e.*, Fast IMT and CE2D) and starting multiple verifiers in parallel. We also evaluate the computation overhead and operational cost of Flash, and show that the operational cost of Flash to verify a large-scale network (with >6,000 nodes and 3.7×10^7 rules) can be as low as $2.74/hour ($24.016/year) for dedicated servers and $4.352/hour ($0.07/run) for on-demand one-shot verification.

This work does not raise any ethical issues.

2 OVERVIEW

As introduced in §1, the ability of Flash to handle *update storms* and *long-tail arrivals* depends on two key novel techniques: *fast inverse model transformation* and *consistent, efficient early detection*.

[1] https://github.com/snlab/flash

In this section, we give the architecture and workflow of Flash to illustrate how these techniques are combined into a single system, and leave the details in §3 and §4.

Figure 1 illustrates the architecture and workflow of Flash. There are two major components in the system.

Subspace Verifier. In a nutshell, a subspace verifier provides the same functionality as an ordinary data plane verifier: it receives FIB updates, reconstructs the data plane state, and verifies whether the properties are violated. In Flash, the subspace verifier is specialized for data plane verification tasks for large-scale networks. It consists of two parts: the *model manager* maintains a snapshot of the data plane based on the received FIB updates, and an equivalent class representation of the data plane computed using Fast IMT (§3); the *CE2D verifier* monitors the equivalent class representation and maintains a CE2D verification graph using consistent early detection algorithms in §4.

In Flash, each subspace verifier maintains the complete FIB snapshots but only verifies the properties for FIB snapshots of a specific *epoch* and a specific packet header subspace. While a subspace verifier can be configured to work as a standalone high-performance data plane verifier, it can only guarantee the correctness of consistent early detection by collaborating with CE2D dispatcher.

CE2D Dispatcher. The CE2D dispatcher has two responsibilities. First, it is responsible for managing the life cycles of subspace verifiers, *i.e.*, creation, destruction, and reconfiguration. It also maintains a mapping from an epoch tag to the set of subspace verifiers who are responsible for verifying the properties for FIBs of the epoch. Second, the dispatcher is responsible for forwarding FIB updates to subspace verifiers based on the epoch-verifier mapping.

Workflow. A typical workflow of using the Flash system is as illustrated in Figure 1. First, operators specify the verification requirements (❶), *i.e.*, data plane properties to be verified, using a specification language based on regular expressions. Static configurations such as the network topology and IP prefix mappings are also required to build the CE2D verification graph. After the system is up and running, it can receive FIB updates from routers, proxies or network simulators (❷). To get consistent early detection results, these FIB updates should be tagged with an epoch tag.

Upon receiving a new epoch from a device (❸), the CE2D dispatcher finds subspace verifiers whose epoch is outdated, stops their execution, and reconfigures them to verify the latest epoch, as specified §4.1. It then updates the epoch-verifier mapping and forwards FIB updates accordingly (❹).

Each subspace verifier maintains an inverse model, *i.e.*, the equivalent class representation of the data plane. When a subspace verifier receives new FIB updates, it first dispatches them into blocks, which are used to compute the latest FIB snapshot and the conflict-free inverse model overwrites using Fast IMT in §3 (❺). The conflict-free inverse model overwrites are then applied to obtain the latest inverse model that is consistent with the new FIB snapshot (❻).

With the inverse model, the CE2D verifier updates the CE2D verification graph (❼) and applies the early detection algorithm. If a deterministic result is returned, the verifier returns a consistent verification results for the subspace (❽).

Table 1: Key Notations.

Symbol	*Meaning*
N	number of devices in a network
$\mathcal{R}$	the rule-based representation of data plane C
r_{ik}, r	the k-th rule in the i-th device's FIB / a rule
m_{ik}, m_r	the match of r_{ik} /r
pri_{ik}, pri_r	the priority of r_{ik} /r
a_{ik}, a_r	the actions of r_{ik} /r
e_{ik}, e_r	the effective predicate for r_{ik}/r
$p_i(y)$	the predicate on the i-th device when the action is y
w	a conflict-free overwrite
a_w, p_w	the action/predicate of a conflict-free overwrite u
M	the equivalent-class representation of data plane C
p^j	the predicate of the j-th element in M
$\vec{y}^j$	the action vector of the j-th element in M

3 FAST IMT

In this section, we give more details of Fast IMT. We first define the problem (§3.1), then give a high-level overview (§3.2), followed by the specification of the core merge-based block update decomposition algorithm (§3.3). More optimizations and data structures to improve speed and resource consumption in practice are given in (§3.4). See Table 1 for key notations.

3.1 Problem Definition and Background

A network data plane configuration C with N devices (routers or switches) can have two different representations that define the same forwarding behavior for any packet header h: the rule-based representation (forward model) and the equivalent class representation (inverse model).

Rule-based Representation (Forward Model). The rule-based representation of a network data plane is denoted as $\mathcal{R} = \{R_i\}_N$, where R_i denotes the forwarding table on the i-th device. A forwarding table consists of multiple forwarding rules in the format of $\langle match, priority, action\rangle$ where the *match* field is a *predicate* (*i.e.*, a Boolean function on the packet header space), the *priority* field is an integer that specifies the priority of the rule, and the *action* is the action to be applied on the packets (*e.g.*, forwarding, discarding).

Let r_{ik} denote the k-th rule in the i-th forwarding table, and let m_{ik}, pri_{ik}, and a_{ik} denote the matching predicate, priority, and action of r_{ik} respectively. The forwarding behavior of C is a function[2]

$$\vec{b}^C_{\mathcal{R}}(h) = (b_1(h), \ldots, b_N(h)),$$

where the action on the i-th device $b_i(h)$ is determined by the rule of the highest priority that can match h:

$$b_i(h) = a_{ik^*}, \text{ where } k^* = \underset{r_{ik}\in R_i, m_{ik}(h)=1}{\arg\max} pri_{ik}.$$

Example (Rule-based Representation). The table labelled $\mathcal{R}$ on the upper left corner of Figure 2 is an example of the FIBs (rule-based representation) of the 3-node network above the table. For compact figure, we show the FIBs of the 3 switches in 3 columns (*i.e.*, R_i is for switch S_i). For simplicity, we assume that the rules are forwarding rules and use their next hop as the action, and that the rules are

[2]This requires that a data plane configuration C does not have syntax errors caused by ambiguous or conflicting forwarding rules, *i.e.*, two rules with overlapping match, same priority but conflicting actions. This typically indicates an error or can be resolved using prior work such as FlowVisor [35].

numbered by the row number. To illustrate how to read the table, consider an example packet header h_1 whose destination IP address is 10.0.1.2. At S_1 (using R_1), both rule r_{11} ($\langle dip = 10.0.1.0/24, 2, A\rangle$) and rule r_{13} ($\langle dip = 0.0.0.0/0, 0, S_3\rangle$) in R_1 can match it, but rule r_{11} has a higher priority. Thus, the output of $b_1(h)$ is A. Looking up h_1 on S_2 (using R_2) and on S_3 (using R_3) with the same procedure, the final output is a vector: $b^C_{\mathcal{R}}(h_1) = (A, S_1, S_1)$.

Equivalent Class Representation (Inverse Model). The equivalent class representation of a network data plane is denoted as $M = \{(p^j, \vec{y}^j)\}_j$, where $(p^j, \vec{y}^j)$ denote the j-th element in M. $\forall j$, p^j is a predicate and $\vec{y}^j$ is an N-dimension action vector, where y^j_i specifies the action on the i-th device. In a valid equivalent class representation M, the entries are 1) *unique*: $\forall j \neq j', \vec{y}^j \neq \vec{y}^{j'}$; 2) *mutually exclusive*: $\forall j \neq j', \forall h, p^j(h) \wedge p^{j'}(h) = 0$; and 3) *complementary*: $\forall h, \exists j, p^j(h) = 1$. Given the characteristics of p^j, they represent atomic predicates originally developed in [21].

In the equivalent class representation, the forwarding behavior of C can be expressed as a function

$$\vec{b}^C_M(h) = \vec{y}^{j^*}, \text{ where } p^{j^*}(h) = 1.$$

Inverse Model Transformation. While the network uses the forward model (rule-based representation), the inverse model (equivalent class representation), on the other hand, provides an efficient data structure for use cases such that given the forwarding behavior $\vec{y}^j$, find the header spaces p^j. In general, a data plane verifier checks whether an unacceptable $\vec{y}^j$ exists. Hence, the goal of data plane verification is to transform the forward model $\mathcal{R}$ to the inverse model M, and $\mathcal{R}$ is equivalent with M, denoted as $\mathcal{R} \sim M$, which is defined as, if and only if $\forall h$, $\vec{b}^C_{\mathcal{R}}(h) = \vec{b}^C_M(h)$.

Example (Equivalent Class Representation). The table labelled M on the lower left corner of Figure 2 is the inverse model of $\mathcal{R}$. One can observe that the network has 2 behaviors: (A, S_1, S_1) for those packet headers in $p_1 \vee p_2$, and (S_3, S_3, GW) for the rest. A data plane verifier typically builds a graph data structure to represent the vectors and runs graph algorithms to check properties on the graph (*e.g.*, no loop). The execution time of the graph algorithms typically is negligible compared with the time to construct M.

Rule Updates and Inverse Model Updates. Upon updates in the data plane such as rule insertions, deletions or modifications, the rule-based representation transfers from the initial state $\mathcal{R}$ to the final state $\mathcal{R}'$. Accordingly, the equivalence-class model must also transfer from the initial state M to a final state M' where $\mathcal{R}' \sim M'$. The right hand side of Figure 2 shows an example of the updated models of the network, when rules are inserted in the 3 switches to handle HTTP to the two subnets specially.

The preceding is a generalization of the computation process of global atomic predicates [21] and APKeep [26]: the global AP is solving the special case where there are no initial rules, *i.e.*, $\mathcal{R} = (\emptyset, \ldots, \emptyset)$, while the APKeep work is solving the special case where each update has only one rule.

3.2 Fast IMT Intuition

With the precise problem definition, we now give the design of Fast IMT. Instead of giving the algorithm bottom up, we focus on building intuitions in this section. A formal, bottom-up model of Fast IMT and its correctness is in Appendix C.

Intuition I: Direct Transformation Revealing Native Update Complexity. First consider direct transformation, which is computationally inefficient but gives us intuition on computing the effect of an update in the forward model. In particular, consider how to compute the predicate p^j for $\vec{y}^j$, for each equivalent class $(p^j, \vec{y}^j)$. Let $p_i(a)$ denote the predicate that selects the header spaces for which the i-th device takes action a. Then $p_i(y^j_i)$ is the union of the *effective predicate* of each rule r_{ik}, whose action a_{ik} is y^j_i, and the *effective predicate* e_{ik} of rule r_{ik} 1) satisfies the match condition of r_{ik}, and 2) is not matched by a rule with a higher priority, *i.e.*,

$$e_{ik} = m_{ik} \wedge \neg \bigvee_{pri_{ik'} > pri_{ik}} m_{ik'}. \tag{1}$$

For the header space to take $\vec{y}^j$, the predicate p^j must satisfy all $p_i(y^j_i)$, *i.e.*, by taking their conjunction:

$$p^j = \bigwedge_{\forall i} p_i(y^j_i) = \bigwedge_{\forall i} \left(\bigvee_{a_{ik} = y^j_i} e_{ik} \right). \tag{2}$$

Equations (1) and (2) show that the effects of a native update (*i.e.*, an update in the forward model) can be global and quite complex in the inverse model. The insertion of a new rule can reduce the effective predicates of a large number of rules with lower priorities, and these rules can have different actions. Meanwhile, the deletion of a rule can increase the effective predicates of a large number of rules with lower priorities, and these rules can also have different actions. In both cases, multiple p^js may need to be updated.

Intuition II: Per-Rule Updates Can be Inefficient. We now build intuition on that per-rule updates are inefficient. Consider two rules r_1 and r_2 with k_1 and k_2 ($k_1 < k_2$) rules that have a higher priority on the same device. To compute the effective predicates for r_1 and r_2, they need to visit k_1 and k_2 rules. However, as $k_1 < k_2$, computing the union of the first k_1 rules when processing r_2 is redundant, which is already computed when processing r_1. Upon an update storm, the redundancy can be quite substantial.

Such redundancy also exists when updating the p^js. To be concrete, consider the updates in Figure 2, which install 6 new rules, with 2 new rules in each switch shown in the top middle of the figure. A simple approach to update the inverse model is to apply the 6 updates one by one. On the other hand, one can verify from the example that clearly the 6 updates can be combined and applied together. Specifically, the 6 updates have only two distinct match conditions (p_4 and p_5). At the final inverse model shown in the lower right corner, p_4 and p_5 do not appear individually, but appear only as the union $p_3 = p_4 \vee p_5$.

Intuition III: Overwrite Operators Allowing Update Composition. To understand the essence why the updates can be combined, consider the first update at S_3 (first row of ΔR_3 in Figure 2), as an *inverse model overwrite* operator (or *overwrite* for short): $(\Delta p = p_4, \Delta y)$, where Δy specifies that it sets the action at S_3 to S_2, written as $\{y_3 = S_2\}$. The effect of the operator is that for each equivalent class $(p^j, \vec{y}^j)$, if the intersection $p^j \wedge \Delta p$ is empty, the operator has no effect; otherwise, the intersected header space should be moved to the other equivalent class with device S_3 taking action S_2 and the rest of $\vec{y}^j$ not changed. The computation process is similar to database *join* and we refer to it as a *cross product*.

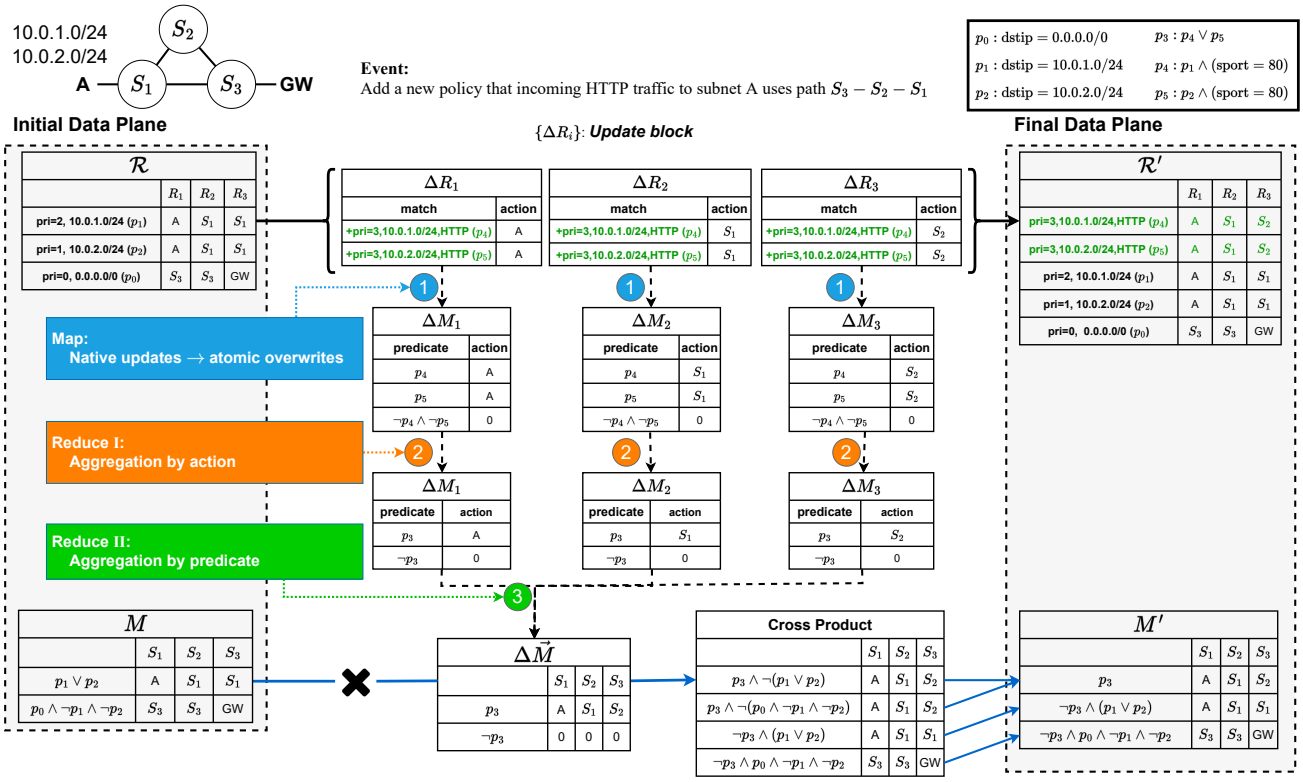


Figure 2: A simple example illustrating Fast IMT.

Now consider the second rule insertion (p_5, S_2), Δy and the process are the same except that $\Delta p = p_5$. If we use $(\Delta p = p_4 \vee p_5, \Delta y)$, the two processes can be composed, improving efficiency.

Next consider the relationship between two overwrites at two switches: the first update $(p_4, \{y_1 = A\})$ at S_2 and the first at S_3 $(p_4, \{y_2 = S_1\})$. One can see that they have the same predicate (p_4). We observe that this happens frequently, as often an update is to set up a new network-wide flow, and the predicate selects the same flow. Still using the overwrite operator perspective, we can compose the two updates into a single one $(\Delta p = p_4, \Delta y)$, where Δy is $\{y_2 = S_1, y_3 = S2\}$.

Both examples can be considered as applying a "reduce" operator in the map-reduce framework [36]. In the first "reduce", the key is Δy, and the predicates (Δp) are reduced (by predicate disjunction), for those with the same key. In the second "reduce", the key is Δp, and the updates (Δy) are reduced (by combining function effects). Figure 2 illustrates these two "reduce" operators, shown as Reduce I and Reduce II, respectively. One need to note that the "reduce" operators guarantee the *conflict-free property*. We say that two overwrite operators $(\Delta p, \Delta y)$ and $(\Delta p', \Delta y')$ have a conflict, if only if Δp and $\Delta p'$ intersect, and they write different actions at the same device but with different actions.

Note that the preceding "map-reduce" process applies only to conflict-free overwrite operators. As we discuss in Intuition I, a native rule update can be complex and is not a conflict-free overwrite. The remaining key idea of Fast IMT is to turn native rule updates into conflict-free overwrites[3], which we show in the next subsection. In particular, if an overwrite only changes the action of a single device, we call it an *atomic overwrite*, and we denote the set of atomic overwrite operators for device i as ΔM_i. This process can be considered as a "map" operation in map-reduce, where each native rule update for device i is mapped to a set of atomic overwrites. Since Fast IMT can be considered a "map" followed by two "reduce", we also refer to it as the MR2 algorithm.

3.3 Efficiently Computing ΔM_i

Given an update block with K updates for a flow table R_i with T entries, a naive, sequential algorithm can compute ΔM_i in $O(KT)$ complexity. This, however, is inefficient. We apply the list merge idea and design Algorithm 1, which consists of two phases: *merging the native updates in the block and finding expanding rules* (L3), and then *computing atomic overwrites for the expanding rules* (L5).

Merging Native Updates. The merging (L7-28) is similar to merging two sorted list, where r and δ represent the heads of R_i and ΔR_i respectively[4]. A Boolean variable is used to indicate whether a rule with higher priority is deleted, which means the effective predicate of the current rule r may expand. The algorithm iterates through the updates (L12-25). It first locates where the update should be

[3]We use "overwrite" to refer to "conflict-free overwrite" in the rest of the paper, as long as there is no ambiguity.

[4]Note that with the default wildcard rule with the lowest priority, r will never reach the end of R_i so no check on r is needed. The same observation applies to CalculateAtomicOverwrite(R_i', R_{diff}) as well.

Algorithm 1: Decomposing Native Rule Update Blocks into Atomic Overwrites through Merging.

```
   Input  : ΔR_i - the block of native updates on the i-th device
   Input  : R_i - the sorted initial rule set before the update
   Output : R'_i - the sorted final rule set after the update
   Output : ΔM_i - the set of atomic overwrites that is equivalent to ΔR_i
1  ΔR_i ← remove canceling updates in ΔR_i
   // (insert-after-delete or delete-after-insert)
2  ΔR_i ← sort ΔR_i by priority in the descending order
3  R_diff ← MergeBlockAndDiff(R_i, ΔR_i)
4  R'_i ← R_i // updates have been applied to R_i
5  ΔM_i ← CalculateAtomicOverwrite(R'_i, R_diff)
6  return R'_i, ΔM_i
7  Function MergeBlockAndDiff(R_i, ΔR_i)
8      R_diff ← empty list
9      higher_priority_rule_deleted ← false
10     r ← the rule with the highest priority in R_i
11     δ = (op, r_δ) ← the update with the highest priority in ΔR_i
12     while δ ≠ NULL do
13         if r_δ is inserted/deleted after r then
14             if higher_priority_rule_deleted then
15                 Append r to R_diff // r may expand
16             r ← get the next rule in R_i
17         else
18             if op is insertion then
19                 Insert r_δ before r // rule insertion
20                 Append r to R_diff // New rules expand
21             else if op is a deletion (r_δ = r) then
22                 Delete r from R_i // rule deletion
23                 higher_priority_rule_deleted ← true
24                 r ← get the next rule in R_i
25             δ ← get the next update in ΔR_i
26     if higher_priority_rule_deleted then
27         Append remaining rules in R_i to R_diff
28     return R_diff
29 Function CalculateAtomicOverwrite(R'_i, R_diff)
30     ΔM_i ← ∅
31     r ← the rule with the highest priority in R'_i
32     r_δ ← the rule with the highest priority in R_diff
33     p ← 0 // accumulative predicate (⋁_{pri_ik' > pri_ik} m_ik')
34     p_c ← 1 // predicate of "no-overwrite" action
35     while r_δ ≠ NULL do
36         while r_δ has a lower/equal priority than r and r ≠ r_δ do
37             p ← p ∨ m_r
38             r ← get the next rule in R'_i
39         eff ← m_{r_δ} ∧ ¬p // effective predicate of r_δ
40         ΔM_i ← ΔM_i ∪ {(eff, {y_i = a_{r_δ}})}
41         p_c ← p_c ∧ ¬eff
42         r_δ ← get the next rule in R_diff
43     ΔM_i ← ΔM_i ∪ {(p_c, ∅)}
44     return ΔM_i
```

applied (L13-16). For an insertion update (L18-20), r_δ should be inserted before r and the new rule r_δ is also expanding. For a rule deletion (L21-24), $r = r_\delta$ and should be deleted. In the meantime, rules with a lower priority in R_i may expand (L14-15, L26-27) so the indicator is updated (L23). For K rule updates in a table with T rules, this step takes $O(K \lg K + T)$ simple operations.

Computing Atomic Overwrites. To compute atomic overwrites, a critical step is to compute the effective predicates of each expanding rule. Flash uses the property that both R'_i and R_{diff} are sorted. The procedure is in L29-44. First, r and r_δ are set to the head of R' and R_{diff} respectively. The algorithm iterates through R_{diff} (L35-42). First, it locates where r_δ is in R'_i. The predicate p is the union of all rules that have a higher priority than r_δ (L37), and the effective predicate of r_δ, *eff*, is computed in L39. Then an atomic overwrite $(eff, \{y_i = a_{r_\delta}\})$ is added to ΔM_i (L40). The *complementary predicate* is the predicate for the special "no-update" overwrite, and it is maintained by subtracting the predicates of overwrites with a concrete action (L41). This step takes $O(T+K)$ predicate operations as each rule is visited only once.

Thus, the algorithm takes $O(K \lg K + T)$ simple operations and $O(T+K)$ predicate operations, *i.e.*, conjunction ($\wedge$), disjunction ($\vee$), or negation ($\neg$), in the worst case. In practice, $T > K$ and predicate operations are much more expensive than a simple operation. Thus, Algorithm 1 is much faster compared to per-update processing, which takes $O(KT)$ simple and predicate operations.

3.4 Fast IMT in Action

While Fast IMT has reduced substantially amount of redundant computation for a large number of updates, there are still engineering optimizations that can be adopted to further improve the speed and resource consumption.

Input Space Partition. The complexity of Fast IMT depends on the sizes of the inverse model and the updates, which are both related to the number of rules in the data plane. By partitioning the space into multiple subspaces, the number of valid pairs in the update process can be reduced, as well as the number of affected rules. Thus, it can substantially improve the speed of model update.

Fast Look-up for Overlapped Rules. The key step in computing atomic overwrites (ΔM_i) is to calculate the effective predicate of each affected rule. One can see that the effective predicate of a rule r will affect or be affected by another rule r' only if their matches overlap, *i.e.*, $m_r \wedge m_{r'} \neq \emptyset$. In some common scenarios, for example, when the data plane mainly consists of longest prefix matching rules, the number of overlapped rules is usually much smaller than the number of rules on a device. To speed up the computation of atomic overwrites, Flash uses a multi-dimension prefix Trie [24] to enable fast look-up for overlapped rules.

Persistent Action Tree. A common operation is to compute the new actions $\vec{y}^*$ by overwriting some elements in the old actions $\vec{y}$. A naive solution, *e.g.*, storing the vector as an array, may take linear time and units. However, the overwriting process of multiple output vectors may have the same sub output vector. Let $\|\vec{y}\|_{\neq 0}$ denote the number of non-zero elements in $\vec{y}$. In the overwriting process, $\vec{y}$ is from M and $\|\vec{y}\|_{\neq 0}$ is usually large. In contrast, $\|\vec{y}^*\|_{\neq 0}$ often equals to one. In Flash, a data structure called persistent action tree (PAT) is introduced. PAT is a persistent [37] balanced binary search tree, which only creates a chain until each node is modified by $\vec{y}^*$. Thus, a single overwriting takes at most $O(\|\vec{y}^*\|_{\neq 0} \times \lg \|\vec{y}\|_{\neq 0})$ time instead of $O(\|\vec{y}^*\|_{\neq 0} + \|\vec{y}\|_{\neq 0})$.

4 CONSISTENT, EFFICIENT EARLY DETECTION

In this section, we introduce how Flash systematically achieves consistent, efficient, early detection. The motivation of consistent early detection comes from the following observation: logically centralized data plane verifiers like Flash need to collect FIB updates either from a centralized database (as in SDN) or from distributed devices, while large-scale networks today use distributed protocols. Existing verifiers require complete knowledge of the data plane,

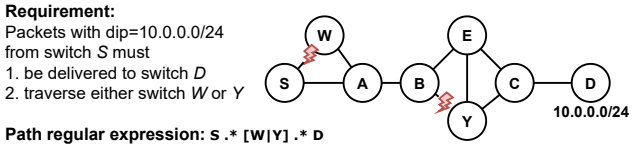


Figure 3: An example of network update setting.

which may take a long time to arrive or even trigger out-of-order deliveries. If the verifier waits until it receives all the updates, it may suffer from intolerable delays or even become unavailable. However, if a verifier simply applies all updates to a single model, it may either report a transient error or even wrong results.

Specifically, Flash proposes solutions to the following two key problems: (1) How to dispatch the rule updates to construct models that are consistent with the converged state? (2) How to efficiently compute verification results with incomplete information? We refer to our solutions as *consistent model construction* and *efficient early detection* (CE2D), respectively.

4.1 Consistent Model Construction

To achieve consistent model construction, Flash must (1) identify consistent FIB updates, *i.e.*, those that are computed from the same network state; (2) identify *potential* converged states; (3) dispatch consistent FIB updates (for *correctness*) of a potential converged state (for *efficiency*) to the same verifier[5]. We have designed mechanisms to construct consistent models for multiple routing protocols. Below, we focus on broadcast state synchronization protocols (or sync-state protocols for short), including link-state protocol OSPF/I-SIS and general state synchronization protocol OpenR. We discuss vector-based protocols (*e.g.*, BGP) in Appendix D.1.

Identifying Consistent FIB Updates with Epoch. Flash uses *network state* to denote the information which is used by routing software to compute the forwarding rules (*e.g.*, link states and prefix configuration). As a network is evolving, its state also changes triggered by different events (*e.g.*, link failure). In sync-state protocols, the state changes are propagated to the entire network.

Flash differentiates the rule updates computed from different network states by dividing them into *epochs*. Specifically, each epoch represents a snapshot of the global network state. As it is not possible to determine whether updates are from the same network state by only looking at the data plane, Flash augments the routing software with an agent. The agent computes the *tag*[6], a unique identifier of an epoch, and associates the tag with FIB updates computed from the state. Flash requires that the message delivery between the agent and the dispatcher is serialized, *i.e.*, updates from the same device are always received in the same order as they are generated. However, Flash does not have any constraints on messages from agents on different devices.

[5]Note that it is not a hard constraint to only run verifiers for converged states, which can be useful when trying to locate the source of an error. However, it can reduce the resource overhead when the network operators are only interested in the verification result of potentially converged state.

[6]In our OpenR implementation, Flash uses the hash value of the keys and versions of the state variables as the epoch tag. We use an XOR based hash function in Boost [38] that can compute the hash value of 1 million entries within 10ms on commodity switches (*e.g.*, Barefoot S9180-32X [39]). To reduce the probability of hash collision, Flash may use multiple hash functions and concatenate the results.

Identifying Potential Converged State through Epoch Dependency Tracking. As Flash requires a strict order between a device and the dispatcher, if the dispatcher receives updates with t_1 on device i before receiving those with t_2, t_1 cannot be the converged state. Similarly, if we see t_2 before t_3 on device j, even if t_2 is the most recent tag on device i, we can infer that t_2 cannot be the converged state. This is referred to as the "happens-before" relation in distributed systems, denoted as $t_1 \prec t_2$. Flash maintains the most recent tag for each device and a set of "active" epochs which has no succeeding epochs. Once a new tag t_{new} is received from device i, whose old tag is $t_i = t_{\text{old}}$, Flash removes t_{old} from the active set and replaces t_i with t_{new}. If t_{new} is not marked as inactive (by another device), Flash adds t_{new} to the active set. An epoch, whose tag is in the active set, is a potential converged state.

Dispatching Consistent FIB Updates As the updates are associated with epoch tags, Flash maintains a mapping from an epoch tag to a verifier. Specifically, upon receiving updates with a new epoch tag t from a device, Flash first appends the updates to the update queue of the device, and checks whether t is in the active set. If t is not in the active set, which means there will be future updates on the same device, no further action is required. Otherwise, if t is in the active set, Flash finds (or creates one if not present) the verifier for t, and feeds the updates from the device's update queue to that verifier. In practice, Flash may need to adopt a back-off mechanism to avoid rapid creation of verifiers due to control plane bugs or improper handling of unstable links.

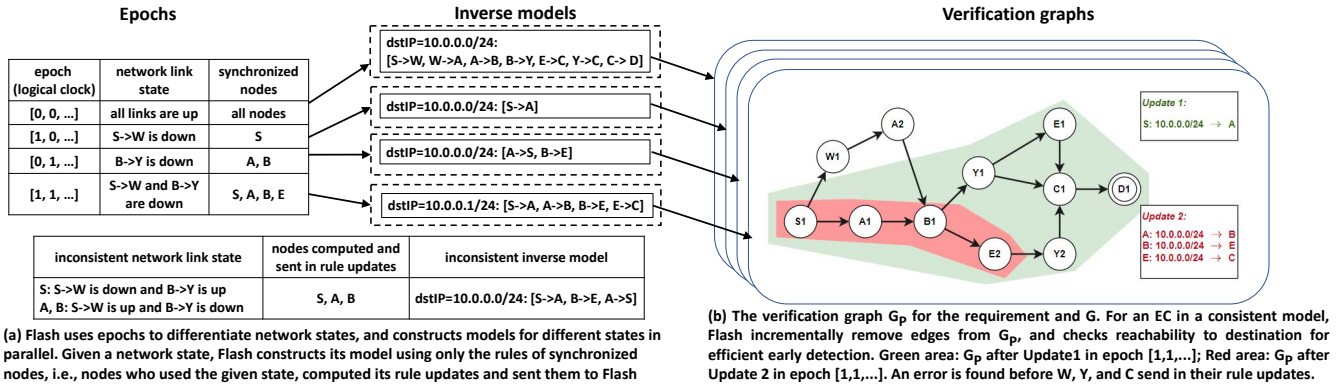
An example update setting is as shown in Figure 3. Consider the updates triggered by two link failures: (S, W) and (B, Y). Assume the initial tag is $t_0 = [0, 0, \dots]$[7] for each device, where the first/second element is the version of link $(S, W)/(B, Y)$. Assume that at time T_1, Flash receives updates from S with tag $t_1 = [1, 0, \dots]$ (after seeing the failure of (S, W)), and from switches A and B with tag $t_2 = [0, 1, \dots]$ (after seeing failure of (B, Y)). At T_1, t_1 and t_2 are potential converged states and are put in different verifiers[8]. However, if at time $T_2 > T_1$, Flash receives updates from switch S, A and B with epoch tag $t_3 = [1, 1, \dots]$, the dispatcher will mark t_1 and t_2 as inactive, and creates a new verifier for t_3 as t_3 is now active. Then, if Flash receives updates tagged with t_2 from switch E, Flash simply appends the updates to the queue of E, and does not dispatch them to any verifier. When it receives updates with t_3 from E, it flushes the updates to the verifier associated with tag t_3. Figure 4 (a) shows the models constructed for each epoch and shows an example of an inconsistent state at the bottom.

4.2 Consistent Early Detection for General Regular Expression Requirements

We first specify how Flash performs consistent early detection for general requirements specified in regular expressions (see Appendix B for detail), including waypoint routing, shortest path, anycast, and multicast, etc. Specifically, Flash leverages the automata theory [40] to transform this problem into an incremental reachability query problem on a decremental verification graph [41].

[7]To better illustrate the partial *happens-before* relationship between epochs, we use the logical clock vector as the tag only in this example.

[8]Note that with a logical clock vector, one may even conclude that both t_1 and t_2 will not be the converged state. However, in practice, it may not be able to derive the causal relationship with the "happens-before" observations.



(a) Flash uses epochs to differentiate network states, and constructs models for different states in parallel. Given a network state, Flash constructs its model using only the rules of synchronized nodes, i.e., nodes who used the given state, computed its rule updates and sent them to Flash

(b) The verification graph G_P for the requirement and G. For an EC in a consistent model, Flash incrementally remove edges from G_P, and checks reachability to destination for efficient early detection. Green area: G_P after Update1 in epoch [1,1,...]; Red area: G_P after Update 2 in epoch [1,1,...]. An error is found before W, Y, and C send in their rule updates.

Figure 4: An illustration to consistent, efficient early detection for the update setting in Figure 3.

Verification Graph and Consistent Partial Verification. Flash computes the verification graph [40, 42–44] as the cross product automata G_p of the network automata and the requirement expression automata, for each packet space H and a set of sources $srcs$. The initial verification graph contains all paths in G that (1) start from $srcs$, and (2) match all the regular expressions.

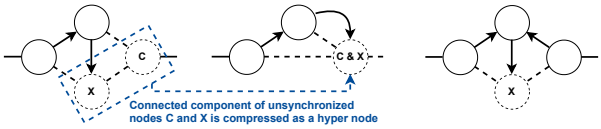
Consider the network and requirement in Figure 3. The requirement is that packets in subspace h that enters at S must reach D along a simple path traversing one of W and Y. Figure 4(b) shows the verification graph with initial state $S1$ and accepting state $D1$.

With this graph, verification of the requirement expression is equivalent to finding a path that can reach an accept state in the verification graph. Specifically, if there exists such a path consisting of only synchronized nodes, the requirement is consistently satisfied. And if there exists a path consisting of only synchronized nodes to the reject state, the requirement is consistently unsatisfied. Otherwise, the verification result is *unknown*.

Decremental Update and Reachability Query. With more nodes becoming synchronized, the set of possible requirement-compliant paths in the network for this epoch are monotonically decreasing. Consider the example in Figure 3, after receiving *Update 1* as in Figure 4(b), the verification graph only contains the nodes and edges in the green area. After further receiving *Update 2*, no valid path is in G_P, which means the requirement in Figure 3 cannot be satisfied, no matter how h is forwarded by other devices. The reachability query in such a decremental graph (*i.e.*, edges are always removed, but never added) has a constant time complexity [41]. The details and pseudocode of the algorithm, as well as how it is extended to perform early detection for more complex traffic patterns (*e.g.*, anycast, multicast, and coverage requirements [27]), is in Appendix D.2.

4.3 Consistent Early Loop Detection

Loop detection is a basic task. However, expressing loop detection requires complex regular expressions and hence is not efficient. Thus, we design a specific CE2D algorithm to check loop(s). A naive approach is to delete unsynchronized nodes and then check if there is a loop in the synchronized nodes, which is simple but can miss early detection opportunities. An alternative is to keep the unsynchronized nodes but assume that an unsynchronized node can take any potential next hop, and the system enumerates all



(a) C and X are unsynchronized. (b) X is unsynchronized.

Figure 5: Examples of all-pair loop-detection.

combinations, which can be computationally expensive. Flash combines the best of these two approaches by introducing a technique called *hyper node compression*.

Hyper Node Compression. Each connected component of unsynchronized nodes is compressed as a hyper node to reduce the cost of enumeration. Let U denote a connected component of unsynchronized nodes, which is replaced by a hyper node w. For each $(u, v) \in E, u \notin U$ and $v \in U$, remove (u, v) and add an edge (u, w) to E. Verifying this graph with hyper nodes will give the same result as the second approach, but avoids the enumeration of paths in U.

Consider the example in Figure 5. In Figure 5(a), when $sync = \{A, B\}$, we can merge unsynchronized nodes C and X into a hyper node, denoted as $C\&X$. In this case, there can potentially be a loop, *e.g.*, $B \to A \to C\&X \to B$, which is $B \to A \to X \to B$ in the original graph, or be no loop, *e.g.*, $B \to A \to C\&X \to out$, which is $B \to A \to X \to C \to out$. Thus, the final result is not determined. However, if C is synchronized, as in Figure 5(b), Flash reports that as long as X does not drop the packet [9], there will be a loop in the final state. This cannot be detected if only synchronized nodes are verified.

Incremental Detection. Instead of detecting loops from all possible nodes, Flash only detects loops that contain the nodes that just become synchronized. Let V_S and V'_S denote the old and new set of synchronized nodes accordingly. If no loop is detected in the old graph, a new loop in the new graph must contain a node in $V'_S \setminus V_S$. Combining hyper node abstraction and incremental detection, Flash incrementally examines new synchronized nodes. If a loop composed of only synchronized nodes is found, the error is reported. Details of the algorithm are given in Appendix D.3.

[9] In certain cases such as ACL, dropping packets is considered acceptable. This algorithm only needs a small revision to work in that case, *i.e.*, making explicit "DROP" action as forwarding to a "virtual switch".

Table 2: Settings used in the evaluation.

Setting	Topology		FIB Generation	FIB Scale	Update Generation	Update Scale	Arrival Pattern
	Name	$\|V\|/\|E\|$					
LNet-apsp Subspace / LNet-apsp	LNet	6,016 / 43,008	StdFIB: Shortest path from each node to the hosts connected to the rack switches	$\frac{2.9\times10^5}{3.2\times10^7}$	Insert each rule in a sequence and then delete it in the same order from the sequence	$\frac{5.8\times10^5}{6.4\times10^7}$	Updates burst into the verifier
LNet-ecmp Subspace / LNet-ecmp			StdFIB*: StdFIB with source match ECMP	$\frac{3.0\times10^5}{3.7\times10^7}$		$\frac{6.0\times10^6}{7.4\times10^7}$	
LNet-smr Subspace / LNet-smr			StdFIB* with suffix match routing	$\frac{2.6\times10^6}{3.7\times10^7}$		$\frac{5.2\times10^6}{7.4\times10^7}$	
Airtel-trace	Airtel 1	68 / 260	Extracted from dataset	6.89×10^4	Extracted from dataset (as a single sequence)	1.42×10^7	
Stanford-trace	Stanford	16 / 37		3.84×10^3	Same as LNet-apsp	7.68×10^3	
I2-trace	Internet2	9 / 28		1.26×10^5		2.25×10^5	
I2-OpenR-loop			Generated by correct OpenR software	216	Each OpenR switch computes the correct FIB updates upon receiving state syncing messages from link events	Dynamic	Updates are sent to the verifier according to the reaction of real OpenR software under link events
I2-OpenR/1buggy-loop-lt			Generated by buggy OpenR software		Same as the I2-OpenR-loop setting, with one switch running a buggy OpenR software that generates incorrect FIB updates		Same as the I2-OpenR-loop setting with one or more switches are configured with an init/max 60s FIB computation backoff
I2-trace-loop-lt			Extracted from dataset	1.26×10^5	Extracted from dataset	$<1.26\times10^5$	

5 EVALUATION

We fully implement Flash and conduct extensive evaluations using various settings, including a real data plane sampled from a large network (LNet), to answer key questions including but not limited to the following: (1) What is the performance of Flash in a real-world, large-scale network with a large number of updates? How robust is the performance of Flash under different settings? (§5.2) (2) What is the performance of Flash with the arrival pattern of FIB updates from real routing suits? (§5.3) (3) How does each optimization technique contribute to the overall performance improvement? (§5.4) (4) What is the overhead (cost) of Flash in terms of computational demand? (§5.5)

5.1 Methodology

Settings. The performance of network verification depends on multiple factors, including topology, FIB pattern, FIB update arrival pattern, and the requirements. For Fast IMT, the settings follow the naming convention of *A-B*, where *A* denotes the network topology, and *B* denotes the FIB generation. For example, LNet-apsp denotes that the setting is using the LNet, a proprietary network based on the Fabric network architecture [45], and the FIB rules are generated by running an all-pair shortest path algorithm. For CE2D, the settings follow the naming convention of *A-B-C*[*-D*], where *A* denotes the network topology, *B* denotes the FIB generation, *C* denotes the property to be verified (all-pair reachability and loop-freeness), and the optional *D*, if present, indicates that there are long-tail arrivals. For example, I2-OpenR/1buggy-loop-lt represents that 1) the network is using the Internet2 topology; 2) the FIB rules are generated by real OpenR software except for one switch, which is running a buggy OpenR instance; 3) the verifier is configured to check the loop-freeness requirement; and 4) there are long-tail arrivals. The details of the settings are summarized in Table 2.

Server Configuration. All evaluations are conducted on cloud Ubuntu servers with 8 vCPUs (2.5GHz) and 32GB memory. The OS is Ubuntu(x64) 18.04.4 LTS with OpenJDK v17.0.3 installed.

Flash Verifier Implementation. Flash is implemented in ~4,000 lines of Java code. We release the source code in the form of a library with all presented features. Developers can easily write adapters that feed rule updates to Flash and extend CE2D to achieve customized verification functions.

Flash Device Agent Implementation. Our device agent (if the consistent, early detection feature is needed) can be implemented by patching an open-source modular routing platform adopted in real-world production networks. In particular, we take the open-source routing software OpenR [33] as the platform in our evaluation. OpenR is a state synchronization protocol that stores its state variable (*i.e.*, Adj, Prefix) in a K-V store, and computes the routes using a *Decision* and *Fib* module. Our extension has around 150 lines of C++ code. Through the extension, rule updates computed from the same network state are encapsulated into Thrift [46] messages with tags attached, and are sent to the verification system.

Delta-net and APKeep Implementations. We compare Flash with two state-of-the-art data plane verification systems, Delta-net and APKeep, because their evaluations have demonstrated that they achieve the highest performance. (1) **Delta-net***: Since we do not have access to the source code, we *implement* Delta-net ourselves in Java following the pseudocode in [25]. Given that Delta-net represents each longest-prefix match as an interval, we directly extend it to handle multi-field match and generic ternary match by representing each match as multiple intervals. We use **Delta-net*** instead of **Delta-net** in the results to indicate that it is our implementation. (2) **APKeep***: Similar to Delta-net, APKeep has no open source implementation and we implement APKeep following the pesudocode in [26]. We use **APKeep*** to indicate our implementation. We use the default "delay merge" parameter

Table 3: Overall performance.

Setting	Total Model Update Time (s)			Memory Usage (MB)			# Predicate Operations (10^5)		
	Delta-net* (speedup)	APKeep* (speedup)	Flash	Delta-net*	APKeep*	Flash	Delta-net* (speedup)	APKeep* (speedup)	Flash
LNet-apsp Subspace	**0.7** (0.3×)	34.1 (15×)	2.3	**29**	66	55	**6** (0.4×)	71 (5×)	14
LNet-ecmp Subspace	26 (1.4×)	> 36,000 (>1895×)	**19**	1,249	> 1,096	**211**	281 (20×)	> 9,529 (681×)	**14**
LNet-smr Subspace	285 (71×)	1,004 (251×)	**4**	6,792	31	**15**	1,485 (26×)	239 (4×)	**57**
Airtel-trace	**12** (0.9×)	85 (6.5×)	13	**5**	37	70	141 (141×)	1,321 (1,321×)	**1**
Stanford-trace	**0.06** (0.5×)	0.58 (4.8×)	0.12	**3**	9	9	**1** (1×)	4.3 (4.3×)	**1**
I2-trace	**0.4** (0.3×)	6.9 (4.9×)	1.4	**16**	52	52	4 (2×)	56 (28×)	**2**

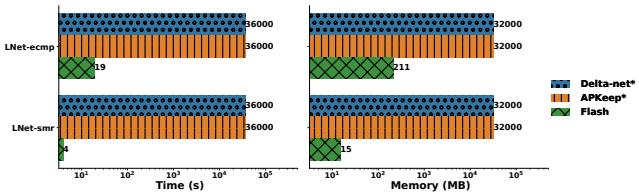


Figure 6: Total model update time (s) and memory consumption (MB) for update storms in baseline settings.

(*i.e.*, 0) in our evaluations. We apply optimizations to Delta-net* and APKeep*, and ensure that they have similar results on datasets reported in [25] and [26] respectively.

5.2 Effects of Fast IMT

In this section, we present the evaluation of Fast IMT by focusing on model construction in different network settings.

Baseline: Benefits of Flash upon Update Storms in Large-scale Networks with Complex Forwarding Behaviors. We first evaluate Delta-net*, APKeep*, and Flash as they are, in our target large-scale networks (LNet-ecmp and LNet-smr), to demonstrate the overall performance gain of Flash. In the baseline evaluation, we generate the updates by putting the rule insertions of all the switches in a sequence, feed the update sequence to the verifier, and measure the execution time and memory consumption to construct the inverse model. We kill the JVM if the execution time exceeds 10 hours. Figure 6 shows the result: Flash (green) finishes in 19 and 4 seconds for LNet-ecmp and LNet-smr respectively, while neither Delta-net* (blue) nor APKeep* (orange) can finish within 10 hours. Hence, for LNet-smr, Flash outperforms Delta-net* and APKeep* by 9,000× (10 hours divided by 4 seconds). The memory consumption of Flash (green) is also substantially lower (up to 2 orders of magnitude in both settings). While the setting seems extreme, it does happen when the verifier is bootstrapping or running on demand (*e.g.*, for network planning [31, 47] or reachability analysis of Virtual Private Cloud [48]). Thus, *we conclude that Flash is fast and memory efficient for data plane verification in large-scale networks.*

Benefits and Robustness of Fast IMT. Then, we analyze the benefits of Fast IMT as a standalone model construction method, by applying the subspace partition idea to Delta-net* and APKeep* in large-scale networks, as subspace partition can be widely effective for large networks. Besides execution time and memory consumption, we also count the number of predicate operations (see §3.3) for each evaluation setting.

The top rows of Table 3 show the results. For example, the row with the name "LNet-smr Subspace" shows the results when subspace partition is applied to all in the LNet-smr. Still, we observe longer model update time for Delta-net* and APKeep*, which is 1.4× (26s/19s) and >1895× respectively that of Flash in LNet-ecmp, and 71× (285s/4s) and 251× in LNet-smr. The memory footprint of Flash is also smaller in these two settings. For example, Flash uses 15MB in LNet-smr, while Delta-net* and APKeep* use 6,792MB and 31MB respectively. Except the LNet-apsp Subspace setting, Flash also reduces the number of predicate operations substantially (20×/681× for Delta-net* and APKeep* respectively in LNet-ecmp, and 26×/4× in LNet-smr).

For completeness, we next use smaller networks and report the result for three settings widely used in literature: Airtel-trace [25], Stanford-trace [18] and I2-trace [49]. The last 3 rows of Table 3 show the result. We can see that the model update time of APKeep* is 4.8-6.5× that of Flash. In such smaller networks, Delta-net* performs the best, with model update time at only 30-90% of Flash. This is true also for some large networks, for example, for LNet-apsp Subspace. However, we see that the number of predicate operations of Delta-net* is higher (up to 141× in Airtel-trace) than Flash. The efficiency of Delta-net* comes from its simple, efficient data structure, which works efficiently for prefix-based rules. However, this representation can suffer significant performance degradation for non-prefix rules, as shown in the LNet-smr and LNet-ecmp results; similar degradation results are reported in [26].

Thus, *we conclude that Flash is robustly fast under various topologies, FIBs, and update settings. In settings with large networks and complex forwarding behaviors (e.g.,* LNet-smr *and* LNet-ecmp*), Flash achieves substantial gains compared with the state-of-the-art.*

Impact of Block Size Threshold when Cooperating with CE2D. Last, as updates may continuously arrive in practice, Flash may choose to update the inverse model and perform CE2D before processing all the rule updates. This behavior is configured by the block size threshold (BST) parameter B, which forces Flash to update the model after processing equal or more than B rule updates. To understand how this parameter affects the performance of Flash, we vary the BST value and measure the model construction time for the settings used in Table 3.

Figure 7 shows the results. The x-axis denotes the proportion of the BST value and the FIB scale, as the FIB scale determines the number of updates in our evaluation and varies significantly in each setting. The y-axis is the normalized model update speed, computed as T_{baseline}/T_x, which categorizes how fast the model update speed is compared to the baseline, where the BST value is infinite and the model is updated after processing all updates. We make the following observations: First, as the threshold increases, the overall trend is that the model update speed will increase and stay at a relatively high level. Second, most settings can reach more than 60% and up to >100% (LNet-apsp, LNet-ecmp, and I2-trace) efficiency

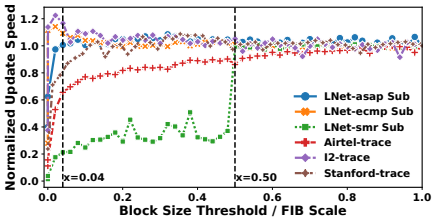


Figure 7: Effects of block size threshold on the model update speed.

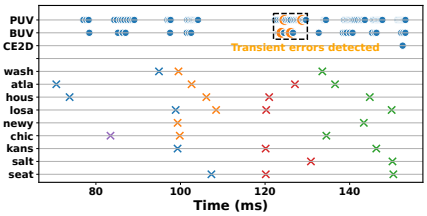


Figure 8: Timeline of FIB updates and verification reports.

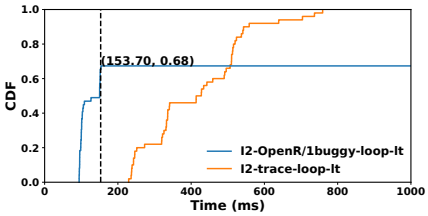


Figure 9: CE2D verification report time.

of the baseline with $x \approx 0.04$. The only exception is `LNet-ecmp` Subspace, which only reaches ~0.2 when $x \approx 0.04$ and > 0.8 when $x \approx 0.5$. However, as the speed-up of Flash in `LNet-smr` Subspace is >70×, Flash can still be as >14× fast as the state-of-the-art. Thus, we conclude that *Fast IMT is more efficient with a larger block size threshold. However, performance improvements can still be obtained when the threshold is no less than 4% of the FIB scale.*

5.3 Effects of CE2D

Now we present the evaluation results of CE2D to demonstrate its efficacy, performance benefits, and robustness in extreme settings.

Ability to Achieve Consistent Early Detection. We first conduct an evaluation to verify that CE2D can provide consistent early detection results. Specifically, we run a simulation using the *I2-OpenR-loop* setting. We use Mininet [50] to construct a network with the Internet2 topology. A real OpenR instance is running on each switch and connects to Flash through the host network. We trigger the FIB computation by bringing down two links (chic-alta and chic-kans) consecutively in Mininet. We compare CE2D with two strategies: (1) per-update verification (PUV), which checks the property after processing a single rule update (*e.g.*, [18, 25, 26]), and (2) block-update verification (BUV), which checks the property after processing a block of updates (*e.g.*, [47]). We record on the verifier the time of the FIB updates from each switch, and the time when the verifier reports a deterministic result.

The results are shown in Figure 8. A cross point (x, y) represents that the FIB update of switch y is received at time x since the link down events are triggered, and the color indicates the epoch tag. A dot point (x, y) represents that the verifier strategy y reports a deterministic result at time x, where orange indicates a loop and blue indicates no loops. As we can see, the two simultaneous link failures trigger the FIB re-computation multiple times on each switch. With both PUV and BUV, the verifier reports two transient loops (the orange dots), which is inconsistent with the final verification result. Meanwhile, *Flash does not report false-positive errors and guarantees that the verification result is consistent.*

Benefits of CE2D upon Long-tail Arrivals. One may question how much performance the CE2D can improve when long-tail arrivals happen. Thus, we evaluate the effects of CE2D in two settings where loops occur: `I2-OpenR/1buggy-loop-lt`, `I2-trace-loop-lt`. For each setting, we run 50 independent random trials and in each trial, we simulate the long-tail effect by configuring one random node to delay 60s before sending the updates. We measure the time when Flash reports a deterministic result.

The results are shown in Figure 9. The x-axis is the report time and the y-axis is the Cumulative Distribution Function (CDF). We limit the range of the x-axis to [0ms, 1,000ms]: if a curve does not reach y=1 at 1,000ms, the result is achieved at 60s, *i.e.*, after receiving the updates from the dampened node. We see in the figure that point (153.7ms, 0.68) is on the curve of `I2-OpenR/1buggy-loop-lt`, indicating that Flash can detect the loop in less than 153.7ms in 68% of the trials, which is substantially smaller (> 390×) than the 60s baseline. In `I2-trace-loop-lt`, Flash can detect the loop early within 760ms in all the trials, yielding a 79x speed-up. Thus, *we conclude that the improvement of CE2D can be quite common (68% to 100%) and substantial (79 to > 390) upon long-tail arrivals.*

Effects of Multiple Dampened Switches. Now we investigate the effects of CE2D when there are multiple dampened switches. We use the `I2-trace-loop-lt` setting, which uses the real network topology and update sequence from the Internet2 dataset. We enumerate the number of dampened devices D from 1 to 7. For each D, we configure Flash to check loops using CE2D and run 50 independent random trials. The results are shown in Figure 10. In 72.5% (*i.e.*, 145/200) of the cases, Flash can detect consistent loops within 800ms, 75× as fast as a complete verification. dampened devices D grows, the probability of successful consistent early detection becomes lower. For example, CE2D can still detect the error within 800ms in more than 90% of the cases when $D \leq 3$, and in ~20% of the cases when $D = 7$, *i.e.*, 77.8% of the switches are dampened. Thus, we conclude that *CE2D can detect errors early even with limited and partial knowledge of the data plane.*

5.4 Micro Benchmark

In this section, we present micro benchmarks that evaluate the effects of several optimization techniques in Flash.

Effects of PAT in Large-scale Networks. We show the effects of persistent action tree (PAT, see §3.4) by analyzing the results in Table 3. Note that the total model update time includes both the time to create and delete equivalence classes (denoted as T_{EC}, which is handled by the persistent action tree (PAT) in Flash (§3.4) and independent of the number of predicate operations, and the time to process the predicates (denoted as T_{OP}). In Table 3, we see that the model construction time improvement is larger than the #predicate operations improvement of Flash over APKeep* in the top 3 large-scale network settings, *e.g.*, 15x and 5x for `LNet-apsp` respectively. The reason is that in smaller networks, the model construction time is dominated by T_{OP}, and the performance gain mainly comes

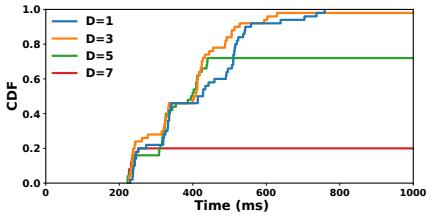


Figure 10: Early loop detection time for different number of dampened switches.

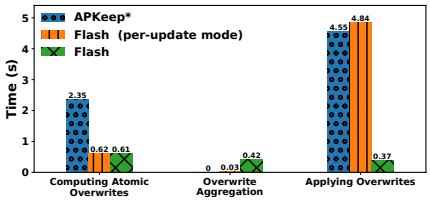


Figure 11: Time breakdown of model construction for `I2-trace`.

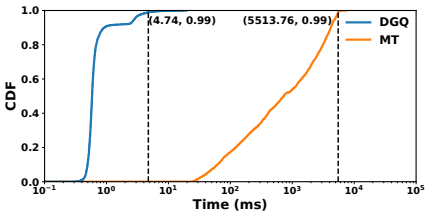


Figure 12: Execution time of all-pair ToR-to-ToR reachability check.

from the reduction of #predicate operations[10]. Meanwhile, *in large networks, T_{EC} becomes dominant in the overall performance, and Flash benefits more from the performance gain of PAT.*

Effects of MR2. To better understand how MR2 (§3.2) contributes to the overall performance gain of Flash, we conduct a breakdown analysis of the 3 phases in the model construction: computing atomic overwrites (Map), overwrite aggregation (*i.e.*, Reduce I/II), and applying overwrites. We measure the total time of each phase for APKeep*, Flash and a variant of Flash, referred to as Flash (per-update mode), where the block size threshold is set to 1.

Figure 11 shows the time breakdown of APKeep*, Flash (per-update mode) and Flash in the `I2-trace` setting. While Flash introduces some overhead of overwrite aggregation (0.42s), computing atomic overwrites (3.85×) and applying overwrites (12.30×) are accelerated compared with APKeep*. Note that it takes longer to apply the overwrites in Flash (per-update mode) (4.84s) than APKeep* (4.55s) because the cross product computation is not optimized for a single rule update. Thus, *we conclude that with overwrite aggregation, MR2 enables Flash to improve the performance by substantially reducing the time to compute the atomic conflict-free model overwrites and to apply the aggregated overwrites.*

Efficiency of Decremental Verification Graph. Last, we show the efficiency of performing consistent early detection for regular-expression-based requirements using the decremental verification graph approach (§4.2). Specifically, we use the `LNet-apsp` Subspace setting and check all-pair ToR-to-ToR reachability. Flash generates 5,376 verification graphs in total, and 48 verification graphs for each subspace verifier. The rule insertions of each switch are packed as a batch. We verify the reachability after processing each batch, and measure the execution time of the verification for a single subspace verifier using (1) the decremental graph query (DGQ, see §4.2) approach, and (2) model traversal (MT, *i.e.*, traversing the model from each source ToR using depth-first traversal).

Figure 12 shows the CDF of verification time. We can see that DGQ (blue) is closer to the y-axis than MT (orange). The median, mean, 99-percentile, and maximum time of DGQ and MT are 0.58/0.84/4.74/19.57ms and 772.98/1,522.22/5,513.76/7,466.87ms respectively. Compared with MT, Flash improves the 99-percentile execution time by ~1,163× (4.74ms v.s. 5,513.76ms). Thus, *we can conclude that the decremental verification graph approach substantially improves the verification performance of regular expression based requirements and enables efficient consistent early detection.*

[10]Note that JDD [51], the BDD library used in Flash, uses caches for predicate operations. Thus, T_{OP} does not scale linearly with the #predicate operations improvement.

5.5 Computational Overhead Quantification

One concern is that Flash may have large computational overheads and hence demand a large amount of computational resources. To quantify the demand, we consider the resource overhead and operational cost of the largest setting in Table 2: `LNet-ecmp` (6,016 nodes and 3.7×10^7 rules). We consider two deployment settings.

We first evaluate the setting of deploying dedicated servers for continuous verification. For `LNet-ecmp`, Flash partitions a subspace for each pod and hence has a total of 112 subspaces. Each subspace verifier requires 1 (v)CPU, 211MB memory for the inverse model, and 336MB memory for the verification graphs to check all-pair-ToR reachability, and each machine requires <4GB memory to run the JVM and store the rules. Thus, the total computation overhead is 112 (v)CPUs, <62GB (61.26GB) (for model and verification graphs) + 4GB (for rules and JVM) memory. If the subspace verifiers are deployed on k machine with dedicated resources, each machine requires $\lceil \frac{112}{k} \rceil$ (v)CPUs and $\lceil \frac{62}{k} \rceil$ + 4GB memory. To get a sense of the cost of the resources in a data center, we apply the availability and pricing of AWS EC2 (US Ohio) [52] on 2022/7/1, to obtain that Flash needs 4 *c6g.8xlarge* (32 vCPUs and 64GB memory) instances. The estimated cost is $2.74/hour.

We next evaluate the one-shot deployment setting where an operator uses on-demand computation resources to verify `LNet-ecmp`. Also selecting 4 *c6g.8xlarge* (32 vCPUs and 64GB memory) instances, Flash completes the one-shot verification in 21s: 1s to receive the rule updates, 19s for model construction, and 939ms for reachability check. Assuming each instance stays up for 1 minute and using AWS pricing data, the system costs $4.352/hour or $0.07/run.

6 RELATED WORK

We now discuss related studies in various fields that motivate the design and implementation of Flash.

Network Verification for Large-scale Networks. Due to the great importance, network verification for large-scale networks have been studied by various studies (*e.g.*, [6, 9, 10, 19, 20, 30–32]). Some designs focus on the control plane (*e.g.*, [9, 10]), which is complementary to data plane verification that Flash focuses on, and some only apply to specific network structures (*e.g.*, symmetry and surgery [20]) and specific requirements (*e.g.*, RCDC [30]) while Flash targets more generic network and requirements. Libra [19] proposes to scale the verification of large-scale networks through distributed computing. It divides the header space into minimal atomic subspaces, where each subspace must have the same forwarding behavior and is essentially an equivalent class in Flash, and distributes the verification of these subspaces to a cluster of

verifiers. The design of subspace partition in Flash is motivated by Libra. However, the main purpose of subspace partition in Flash is to reduce queuing of FIB updates and the memory cost. Thus, Flash allows multiple ECs to be verified in a subspace.

Some large-scale networks use simulation-based control plane verification tools (*e.g.*, [6, 31, 32]). Flash can be a good complement to these studies by verifying the generated rules. Recent studies (*e.g.*, [47, 53]) are pioneers in this direction.

Data Plane Representations for Verification. Data plane representation is the core data structure of data plane verification. Some data plane verification tools [15, 17, 18, 27, 28] use the flow table representation, and develop computation models on top of it (*e.g.*, the SAT model [17], the header space algebra and packet transformation function [18], and the Datalog model [27]). Another set of data plane verification tools [21–26] use equivalent classes as the data plane representation. Equivalent classes decouple the header space analysis and the verification of requirements, and both steps can be efficiently carried out if the number of ECs is small. Recent studies (*e.g.*, [26, 54]) also mitigate the explosion of EC by maintaining multiple sets of ECs and lazily computing the cross product. The EC approaches also benefit from efficient data structures to manipulate header spaces: Delta-net [25] develops an interval-based data structure which is efficient when handling prefix-based rules, while others (*e.g.*, APKeep [26] and Flash) use Binary Decision Diagrams (BDD) for memory efficiency under more general settings. Among the EC approaches, Flash is the first to design efficient data structures to store and manipulate the actions, which only become critical when applying to large-scale networks. DNA [47] also independently develops two ideas of batching updates. Specifically, the "batch insertion and deletion" idea is the same as the "remove canceling updates" (L1 in Algorithm 1), and the "batch forwarding behaviors on the same device" idea is similar to "aggregation by action" (Reduce I) in Flash but is more restrictive: the aggregated updates must have the same actions before and after the update. As Flash takes a more in-depth analysis of the block update problem, the optimizations developed in Fast IMT are generic and can be applied to other EC implementations.

Automata Theory in Networking. The decremental verification graph idea is motivated by studies [42–44, 55] that apply automata theory to policy routing. These studies model network and routing constraints as automatons. By computing the product automaton, these studies can identify network paths that satisfy the routing constraints. Flash also computes the product automaton but uses it for verification purposes, combining with recent approach in fast reachability check [41].

7 EXTENSIONS AND DISCUSSIONS

We discuss three aspects where Flash can potentially be extended: requirement specification, data plane models, and implementations.

Requirement Specification. Currently, Flash uses path regular expressions (PRE) to express requirement specifications. While PRE is simple and easy to comprehend, it typically requires assumptions such as shortest paths to avoid state explosions, and cannot be easily extended to support non-shortest-path algorithms. As Flash uses the requirement specification to build the decremental verification graph (DVG) that encodes all valid paths, one potential extension is to take the DVG directly as an input. Thus, Flash can potentially be interfaced with the frontend of many SDN programming languages (*e.g.*, [42, 43, 55–59]) which also computes the set of all valid paths.

Data Plane Models. Flash focuses on the data plane model where packets are forwarded only based on headers and there are no header rewrites, as discussions with operators suggest that header rewrites mostly take place at end hosts (*e.g.*, [60]) in our target large-scale network. However, it might be desirable to extend Flash for stateful routing (*e.g.*, P4 [61] and NPL [62]) or to support common header rewrites, such as NAT or tunnels. *Stateful routing* requires extensions to Fast IMT. Specifically, the forwarding function model $\vec{b}(h)$ needs to be revised as $\vec{b}(h, s)$ where s denotes the states. If the actions may change the header or the state, extensions to CE2D are required. There are two directions to handle *header/state rewrites* in Flash. The first direction (*e.g.*, [26]) is to guarantee that any packet, if rewritten, belongs to exact one EC before and after the rewrite. Another direction is to enable recursive queries (*e.g.*, [54]). CE2D must be extended, *e.g.*, by adding links between the decremental verification graph of different ECs. As the non-determinism increases when header rewrites can be performed on unsynchronized nodes, the benefits of CE2D may be reduced. Both approaches can break the subspace partition, as the EC after header rewrite may belong to another partition or even on another machine.

Implementation. Flash already comes with many optimizations. One potential extension to further improve the performance benefits of Flash is to leverage parallelism and pipelining. Currently, the Fast IMT and CE2D of a subspace verifier run on the same core. With more CPUs, we can decouple these two steps, and parallelize the requirement verification for each EC. Fast IMT may also be parallelized by leveraging recent BDD libraries (*e.g.*, [63]) that allow efficient concurrent BDD/predicate operations.

8 CONCLUSION

We presente Flash, a fast and scalable system that addresses two important issues in data plane verification for large-scale networks: update storm and long-tail update arrival. Flash contributes two new ideas: *fast inverse model transformation* that achieves throughput-optimized block update processing, and *consistent, efficient, early detection* that enables consistent verification with partially complete data plane information. Extensive experiments on large-scale datasets demonstrate the efficacy and efficiency.

ACKNOWLEDGMENT

We thank all the constructive comments from the anonymous reviewers. The authors are grateful to the support: Y.Richard Yang's research is supported in part by a Facebook Systems Networking Award, Kai Gao is supported by NSFC Grant No. #61902266, Qiao Xiang is supported in part by NSFC Award #62172345, an Alibaba Innovative Research Award, Open Research Project of Zhejiang Lab #2022QA0AB05, Future Network Innovation Research Award of Ministry of Education of China #2021FNA02008 and Tan Kah Kee Innovation Laboratory Award #HRTP-2022-34.

REFERENCES

[1] Pavan Gumaste. Amazon AWS outage, 2020. URL https://www.whizlabs.com/blog/amazon-aws-outage/.

[2] Jay Peters. Prolonged AWS outage takes down a big chunk of the internet, 2020. URL https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet.

[3] Alex Hern. Google Suffers Global Outage with Gmail, YouTube and Majority of Services Affected, 2020. URL https://www.theguardian.com/technology/2020/dec/14/google-suffers-worldwide-outage-with-gmail-youtube-and-other-services-down.

[4] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, pages 765–780, Amsterdam, Netherlands, 2016.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, Los Angeles CA USA, August 2017. ACM. URL https://dl.acm.org/doi/10.1145/3098822.3098834.

[6] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 469–483, 2015.

[7] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. 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*, SIGCOMM'20, pages 599–614, Virtual Event USA, July 2020. ACM. URL https://dl.acm.org/doi/10.1145/3387514.3406217.

[8] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 217–232. USENIX Association, 2016.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 476–489, Budapest, Hungary, 2018. Association for Computing Machinery. URL https://doi.org/10.1145/3230543.3230583.

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract Interpretation of Distributed Network Control Planes. In *Proceedings of the ACM on Programming Languages*, volume 4 of *POPL'19*, pages 1–27, New York, NY, USA, December 2019. Association for Computing Machinery. URL https://doi.org/10.1145/3371110.

[11] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 300–313, New York, NY, USA, 2016. ACM. URL http://doi.acm.org/10.1145/2934872.2934876.

[12] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically Repairing Network Control Planes Using an Abstract Representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 359–373, New York, NY, USA, 2017. ACM. URL http://doi.acm.org/10.1145/3132747.3132753.

[13] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and General Network Verification. In *arXiv:1906.02043 [Cs]*, June 2019. URL http://arxiv.org/abs/1906.02043.

[14] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable Network Configuration Verification through Model Checking. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 953–967, 2020.

[15] G. G. Xie, D. A. Maltz, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3 of *INFOCOM'05*, pages 2170–2183 vol. 3. IEEE, March 2005.

[16] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, pages 37–44, 2010.

[17] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, Toronto, Ontario, Canada, 2011. Association for Computing Machinery. URL https://doi.org/10.1145/2018436.2018470.

[18] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'12, pages 113–126, San Jose, CA, April 2012. USENIX Association. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[19] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pages 87–99, Seattle, WA, April 2014. USENIX Association. URL https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng.

[20] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. Scaling Network Verification using Symmetry and Surgery. *ACM SIGPLAN Notices*, 51(1):69–83, 2016.

[21] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. In *2013 21st IEEE International Conference on Network Protocols*, ICNP '13, pages 1–11, 2013.

[22] Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon S Lam. Practical Network-wide Packet Behavior Identification by AP Classifier. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT'15, pages 1–13, 2015.

[23] Hongkun Yang and Simon S Lam. Scalable Verification of Networks with Packet Transformers Using Atomic Predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.

[24] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 15–27, Lombard, IL, April 2013. USENIX Association. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid.

[25] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 735–749, Boston, MA, 2017. USENIX Association. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex.

[26] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 241–255, Santa Clara, CA, February 2020. USENIX Association. URL https://www.usenix.org/conference/nsdi20/presentation/zhang-peng.

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

[28] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 99–111, Lombard, IL, April 2013. USENIX Association. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian.

[29] Stefano Vissicchio, Luca Cittadini, Olivier Bonaventure, Geoffrey G Xie, and Laurent Vanbever. On the Co-existence of Distributed and Centralized Routing Control-planes. In *2015 IEEE Conference on Computer Communications*, INFOCOM '15, pages 469–477. IEEE, 2015.

[30] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating Datacenters at Scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM'19, pages 200–213, Beijing China, August 2019. ACM. URL https://dl.acm.org/doi/10.1145/3341302.3342094.

[31] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 599–613, Shanghai China, October 2017. ACM. URL https://dl.acm.org/doi/10.1145/3132747.3132759.

[32] Nuno P. Lopes and Andrey Rybalchenko. Fast BGP Simulation of Large Datacenters. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 11388, pages 386–408. Springer International Publishing, Cham, 2019. URL http://link.springer.com/10.1007/978-3-030-11245-5_18.

[33] Facebook. Facebook open routing, 2020. URL https://github.com/facebook/openr.

[34] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM. URL http://doi.acm.org/10.1145/1851182.1851192.

[35] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. FlowVisor: A Network Virtualization Layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009. URL http://sb.tmit.bme.hu/mediawiki/images/c/c0/FlowVisor.pdf.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008. URL http://doi.acm.org/10.1145/1327452.1327492.

[37] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making Data Structures Persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[38] Daniel James. Boost Container Hash Library, 2022. URL https://www.boost.org/doc/libs/1_79_0/libs/container_hash/doc/html/hash.html.

[39] Ufi Space. Barefoot S9180-32X switch, 2022. URL https://www.ufispace.com/uploads/able/files/productfilemanager/000045467d1fc648d792c404372956a0.pdf.

[40] Harry R Lewis and Christos H Papadimitriou. Elements of the Theory of Computation. *ACM SIGACT News*, 29(3):62–78, 1998.

[41] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Fully Dynamic Single-source Reachability in Practice: An Experimental Study. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments*, volume ALENEX'20, pages 106–119. SIAM, 2020.

[42] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM. URL http://doi.acm.org/10.1145/2674005.2674989.

[43] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 328–341, New York, NY, USA, 2016. ACM. URL http://doi.acm.org/10.1145/2934872.2934909.

[44] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A Programmable System for Performance-aware Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 701–721, Santa Clara, CA, February 2020. USENIX Association. URL https://www.usenix.org/conference/nsdi20/presentation/hsu.

[45] Alexey Andreyev. Introducing Data Center Fabric, the Next-generation Facebook Data Center Network - Engineering at Meta, 2014. URL https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[46] Apache. The Apache Thrift Software Framework, 2021. URL https://thrift.apache.org.

[47] Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. Differential Network Analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'22, pages 601–615, Renton, WA, April 2022. USENIX Association. URL https://www.usenix.org/conference/nsdi22/presentation/zhang-peng.

[48] AWS documentation. What is VPC Reachability Analyzer? - Amazon Virtual Private Cloud, 2022. URL https://docs.aws.amazon.com/vpc/latest/reachability/what-is-reachability-analyzer.html.

[49] The Internet2 Observatory. The internet2 dataset, 2021. URL http://www.internet2.du/research-solutions/research-support/observatory.

[50] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, Monterey, California, 2010. Association for Computing Machinery.

[51] Arash Vahidi. A BDD and Z-BDD Library written in Java, 2020. URL https://bitbucket.org/vahidi/jdd.

[52] Amazon Web Services. AWS Pricing Calulator, 2022. URL https://calculator.aws.

[53] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. Symbolic Router Execution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'22, Amsterdam, Netherlands, 2022. ACM.

[54] Ryan Beckett and Aarti Gupta. Katra: Realtime Verification for Multilayer Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 617–634, Renton, WA, April 2022. USENIX Association. URL https://www.usenix.org/conference/nsdi22/presentation/beckett.

[55] Hao Li, Peng Zhang, Guangda Sun, Chengchen Hu, Danfeng Shan, Tian Pan, and Qiang Fu. An Intermediate Representation for Network Programming Languages. In *4th Asia-Pacific Workshop on Networking*, APNet'20, pages 1–7, Seoul Republic of Korea, August 2020. ACM. URL https://dl.acm.org/doi/10.1145/3411029.3411030.

[56] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM. URL http://doi.acm.org/10.1145/2785956.2787506.

[57] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, NSDI'15, pages 59–72, Oakland, CA, 2015. USENIX Association. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim.

[58] Kai Gao, Taishi Nojima, and Y. Richard Yang. Trident: Toward a Unified SDN Programming Framework with Automatic Updates. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 386–401, New York, NY, USA, 2018. ACM. URL http://doi.acm.org/10.1145/3230543.3230562.

[59] Eman Ramadan, Hesham Mekky, Cheng Jin, Braulio Dumba, and Zhi-Li Zhang. Taproot: Resilient diversity routing with bounded latency. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, pages 135–147, New York, NY, USA, 2021. Association for Computing Machinery. URL https://doi.org/10.1145/3482898.3483364.

[60] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'16, pages 523–535, Santa Clara, CA, 2016. USENIX Association. URL https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud.

[61] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM. URL http://doi.acm.org/10.1145/2486001.2486011.

[62] Broadcom. NPL - Network Programming Language Specification v1.3. Technical report, 2019.

[63] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddNF: An Efficient Data Structure for Header Spaces. In Roderick Bloem and Eli Arbel, editors, *Hardware and Software: Verification and Testing*, volume 10028, pages 49–64. Springer International Publishing, Cham, 2016. URL http://link.springer.com/10.1007/978-3-319-49052-6_4.

[64] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. Association for Computing Machinery. URL https://doi.org/10.1145/2785956.2787508.

[65] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. SWIFT: Predictive Fast Reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 460–473, New York, NY, USA, 2017. ACM. URL http://doi.acm.org/10.1145/3098822.3098856.

[66] FRRouting project, 2021. URL https://frrouting.org.

[67] B. Quoitin and S. Uhlig. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network*, 19(6):12–19, November 2005. URL http://ieeexplore.ieee.org/document/1541716/.

[68] Shahrooz Pouryousef, Lixin Gao, and Arun Venkataramani. Towards Logically Centralized Interdomain Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'20, pages 739–757, Santa Clara, CA, February 2020. USENIX Association. URL https://www.usenix.org/conference/nsdi20/presentation/pouryousef.

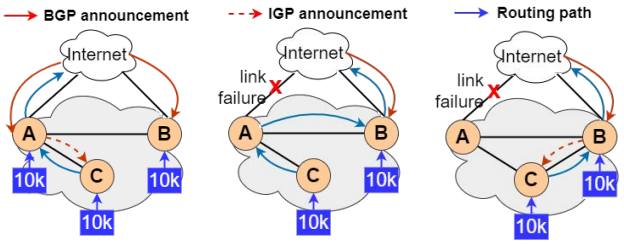


Figure 13: Update storm in inter/intra-domain routing. Red solid lines: BGP announcements; red dashed lines: IGP announcements; blue lines: routing paths. (a) - Initial state. (b) - Inter-domain link failure. (c) - Intra-domain link recovery.

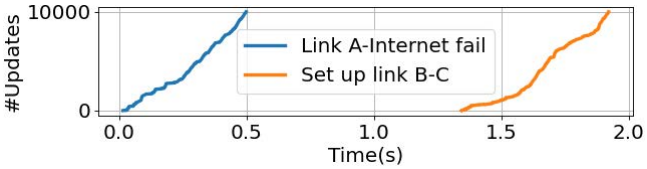


Figure 14: Accumulative distribution of updates.

Appendices are supporting material that has not been peer-reviewed.

A UPDATE STORM ANALYSIS

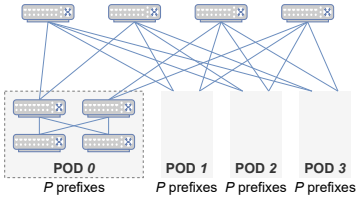
As scales and complexities of computer networks grow [64], it is not uncommon that a large number of data plane updates are triggered by a single root cause both in real networks and in large-scale simulations.

Prior work [65] has shown that a single failure in the Internet topology can trigger bursts of BGP withdraws (up to 100,000), which can lead to FIB changes in intra-domain networks. For example, consider the network in Figure 13(a), assume it receives 10K prefixes from AS-1 and AS-2 and the optimal path for the 10K prefixes is through border router A, as shown in Figure 13(a). After the failure of a remote inter-domain link, the prefixes are withdrawn from AS-1 and the new optimal path is to forward packets through router B, as shown in Figure 13(b). For a 3-node topology, the change will trigger 30K FIB updates.

A link change in the intra-domain may also trigger a large number of FIB updates. Consider the network in Figure 13(c), when a new link is set up from C to B, the optimal intra-domain path is changed from C - A - B to C - B. Thus, it will trigger 10K updates for the selected prefixes. We evaluate the scenario of Figure 13 using FRR[66], Figure 14 shows the accumulative distribution of updates when we trigger the link events. When link A-Internet fails, we receive 10K[11] burst updates from A in 0.48s, and 1.36s after link B-C is set up, we receive 10K burst updates from C in 0.58s.

Network simulations [6, 13, 31, 32, 67] have been reported to effectively reduce the risk of conducting complex network operations

[11] We also received 10K FIB updates from C when link A-Internet fails, but the updates from C have not changed its original FIB, thus these updates are not reported.



K	P	$\|R\|$	$\|\Delta R\|$
4	2	160	56
8	4	2,560	512
16	8	40,960	4,352
32	16	655,360	35,840
32	32	1,310,720	71,680

Figure 15: Update storm in network planning: connecting a new POD to the data center network. K - Fat tree parameter; P - prefix per pod; $|R|$ - # of total rules after the change; $|\Delta R|$ - # of modified rules.

reqs	::=	*req**	
req	::=	(*packet_space*, *sources*, *P*)	
P	::=	*P* **and** *P*	(Set Intersection)
	|	*P* **or** *P*	(Set Union)
	|	**not** *P*	(Complementary)
	|	**cover** *P*	(Coverage)
	|	*hop**	(Regular Expression)
hop	::=	ID	(Select by ID)
	|	[*label op val*]	(Select by label)
	|	.	(Any)
	|	*	(Repeat)
	|	^	(Start of Path)
	|	$	(End of Path)
	|	>	(Packet Destination)
op	::=	=	(Equals)
	|	**contains**	(Contains)
	|	**matches**	(Matches a regular expression)

Figure 16: Requirement Specification Language.

in production networks. State-of-the-art network simulation tools such as FastPlane [32] can compute the RIB and FIB of a network of more than 2,000 routers in a few hundreds of seconds, and the scale of the RIB and FIB entries can be up to hundreds of millions. For example, consider the scenario shown in Figure 15, assume the network has a K-ary Fat tree topology and each POD has P prefixes. When $K = 32$ and $P = 32$, 1,310,720 FIB rules and 71,680 FIB changes are generated by BGP simulation when a new pod is added to a data center network.

Feeding these entries to a data plane verifier leads to both *long verification time* and *large memory consumption*. The large number of updates can also be challenging even for incremental data plane verifiers [25, 26, 28]. While one can resort to the divide-and-conquer approach [19], the cost depends on both the partition method and the complexity of the data plane and can still be very large. For example, in a dataset of 6 million FIB updates in a Clos topology with 6,000 switches, even if we divide the header space into 112 partitions, a reproduced version of a state-of-the-art data plane verifier [26] takes tens of minutes to hours to complete the verification for one partition. Further increasing the number of partitions can certainly help reduce the scalability challenges posed by the update storm but inevitably increases the cost to verify one network configuration.

B A DECLARATIVE REQUIREMENT SPECIFICATION LANGUAGE.

Verification Requirement Specification Language. Figure 16 gives a simplified grammar of the requirement specification language in Flash. For switches with external ports (*e.g.*, a ToR switch

or a border router), Flash attaches a virtual node to each external port. For each virtual node, Flash assigns the set of IP prefixes to the *prefixes* label, which indicates that the virtual node owns the prefixes. In this way, operators can select any external destinations. On a high-level, a requirement is specified by a tuple of (*packet_space*, *sources*, *path_set*). The semantic means for every packet in *packet_space*, when it enters the network from any device in *sources*, the network must be forward it along at least one sequence of devices belonging to *path_set* specified as a regular expression. For example, tuple (*sip* = 10.0.1.0/24 *and dip* = 10.0.2.0/24, $[S], S.*W.*>\$$) specifies that for any packet with a source IP in 10.0.1.0/24 and a destination IP in 10.0.2.0/24 entering network at device S, it must be able to reach at least one device with an external port reachable to the destination of a packet while waypointing device W. The language can expressive requirement on common communication patterns such as unicast, anycast and multicast. It also provides a **cover** key word, when used together with *path_set*, means every $pkt \in packet\_space$ entering the network from a device $v \in sources$, it must be forwarded along all paths in *path_set* that starts from v. With this key word, the language can express requirements such as "all redundant shortest paths should be available" [27].

C A FORMAL THEORY OF IMT

C.1 Basic Concepts

Input and Output. The inverse model transformation (IMT) can be applied beyond networking. Thus, we use the term *input* as a generalization of *header field*, and the term *output* as a generalization of *network-wide forwarding behaviors*. First, we define the basic elements of IMT theory. IMT does not deal with a concrete input but with a *set of inputs*, *predicate* is a boolean function that specifies a specific set of inputs and is a key concept in IMT.

Definition 1 (Input Space, Predicate and Selection.). *Let L denote the number of bits for an input, the input space $\mathcal{X} \triangleq \{0,1\}^L$ is the set of all possible input values. A predicate is a boolean function $p : \mathcal{X} \mapsto \{0,1\}$, and the selection of a predicate is defined as $\sigma_p \triangleq \{x \in \mathcal{X} | p(x) = 1\} \subseteq \mathcal{X}$. Let $\mathcal{P}$ denote the space of predicates on $\mathcal{X}$, i.e., $\mathcal{P} \triangleq \mathcal{X} \mapsto \{0,1\}$. Let p0 denote the predicate that $\forall x \in \mathcal{X}$, $p0(x) = 0$, i.e., $\sigma_{p0} = \emptyset$.*

As introduced in §3.2, a basic operator to compute model update is the *overwrite* operator. In the paper, we use $y_3 = S_2$ to denote the behavior of rewriting the output (*i.e.*, actions in the context of the paper) on S_3 to forwarding to S_2. The output space Y_i specifies valid values for the i-th output (*e.g.*, the next hop on the i-th device). This operator can be expressed as a vector in the output space, with the introduction of "no-update", *i.e.*, $\mathbf{0}$ (note that this is different from a numerical zero, which can be potential valid output). For example, $y_3 = S_2$ can be expressed as $(\mathbf{0}, \mathbf{0}, S_2)$.

Definition 2 (Output Space and Output Overwrite.). *For a system with N outputs, let V_i denote the space of all valid values for the i-th output. Let $Y_i \triangleq V_i \cup \{\mathbf{0}\}$ denote the space of the i-th output, a binary operation* overwrite $\leftarrow_i$*: $Y_i \times Y_i \mapsto Y_i$ is defined as:*

$$a \leftarrow_i b = \begin{cases} a & \text{if } b = \mathbf{0}, \\ b & \text{otherwise} \end{cases}$$

Let $\mathcal{Y} \triangleq Y_1 \times \cdots \times Y_N$, we define $\leftarrow$: $\mathcal{Y} \times \mathcal{Y} \mapsto \mathcal{Y}$ as below: $\vec{a} \leftarrow \vec{b} = (a_1 \leftarrow_1 b_1, \ldots, a_i \leftarrow_i b_i, \ldots, a_N \leftarrow_N b_N)$. Let $\mathbf{0}_K = \underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{K}$, we have $\vec{\mathbf{0}} = (\mathbf{0}_N)$ is the identity element.

Forward Model. We now give the formal model of the forward model, *i.e.*, rule-based representation. In particular, we focus on the case where rules have no conflicts, as mentioned in §3.1.

Definition 3 (Rule). *A rule r consists of 3 parts: a match predicate (denoted as $r.pred \in \mathcal{P}$), a priority (denoted as $r.pr \in \mathbb{N}$), and an output vector, denoted as $(r.\vec{y})$.*

Definition 4 (Well-behaved Forward Model (Rule-based Representation)). *A rule-based representation R is well-behaved if and only if (1) $\forall r_i \neq r_j \in R, \sigma_{r_i} \cap \sigma_{r_j} \neq \emptyset \Leftrightarrow (r_i.pr \neq r_j.pr) \vee (r_i.\vec{y} \leftarrow r_j.\vec{y} = r_j.\vec{y} \leftarrow r_i.\vec{y})$, i.e., there are no conflicts for overlapped rules; (2) $\forall i \in [1, N], \forall x \in \mathcal{X}, \exists r \in R, (x \in \sigma_r) \wedge (r.y_i \neq \mathbf{0}) = 1$, i.e., outputs for domain $\mathcal{X}$ is fully specified.*

Based on the well-behaved forward model, we can define its *behavior function*, which represents the outputs for a given input. In the context of networking, the output vector gives the action of a packet header on each device. For each input x, its behavior is specified by the rule r that can match the input, *i.e.*, $r.pred(x) = 1$, and has the highest priority. This behavior model mimics the FIB forwarding behavior.

Definition 5 (Behavior Function of R). *Each well-behaved forward model R uniquely defines a behavior function $\vec{b}_R : \mathcal{X} \mapsto \mathcal{Y}$. Specifically, let $R_i \triangleq \{r \in R | r.y_i \neq \mathbf{0}\}$. $\forall i \in [1, N], \forall x \in \mathcal{X}, \vec{b}_R(x) \triangleq (b_{R_1}(x), \ldots, b_{R_N}(x))$ where*

$$b_{R_i}(x) \triangleq r.y_i, r = \underset{r \in \{r \in R_i | x \in \sigma_r\}}{\arg\max} r.pr.$$

Inverse Model. We now give the definition of the inverse model. Each inverse model is a set of $(p, \vec{y})$ pairs, where $p \in \mathcal{P}$ is a predicate and $\vec{y} \in \mathcal{Y}$ is an output vector. Clearly, the space of models is a subset of $\mathcal{P} \times \mathcal{Y}$. However, inverse models must satisfy the constraints, as in Definition 6. We define a boolean function im which takes a subset of $\mathcal{P} \times \mathcal{Y}$, and returns 1 if it is an inverse model, and 0 otherwise.

Definition 6 (Inverse Model (Equivalent-class Representation)). *$M = \{(p^j, \vec{y}^j)\}_{|M|}$ is an inverse model, or simply a model, i.e., $\text{im}(M) = 1$, if and only if (1) $\forall i \neq j \in [1, |M|], \vec{y}_i \neq \vec{y}_j$, i.e., each output vector is unique; (2) $\forall i \neq j \in [1, |M|], \sigma_{p_i} \cap \sigma_{p_j} = \emptyset$, i.e., selections of the predicates are mutually exclusive; (3) $\bigcup_{i \in [1, |M|]} \sigma_{p_i} = \mathcal{X}$, i.e., all the predicates fully cover the target input space $\mathcal{X}$.*

Similarly, we can also define the behavior function of M. As the predicates are mutually exclusive but complete, the predicate of one and only one pair $(p, \vec{y})$ can match the input. The output vector of the input x is the associated $\vec{y}$. From the definition of an inverse model's behavior function, we can see that it is very easy to compute its inverse function, which is exactly the reason why we call M the *inverse model*.

Definition 7 (Behavior Function of M). *Each inverse model M uniquely defines a behavior function $\vec{b}_M : \mathcal{X} \mapsto \mathcal{Y}$. Specifically,*

$$\vec{b}_M(x) \triangleq \vec{y}_M(p), \exists p \in \mathcal{P}_M, p(x) = 1.$$

Definition 8 (Inverse Behavior Function). *Each inverse model $M \in \mathcal{M}$ define an inverse behavior function $b_M^{-1} : \mathcal{Y} \mapsto \mathcal{X}^*$:*

$$b_M^{-1}(\vec{y}) = \begin{cases} \sigma_{p_M(\vec{y})} & \text{if } \vec{y} \in \mathcal{Y}_M \\ \emptyset & \text{otherwise.} \end{cases}$$

The key operator, which is used to manipulate inverse models, is defined in Definition 9. This operator is to *overwrite* (see Figure 17 for a formal definition) the outputs of a model M with refined outputs in M'. If y_i of a refined output contain $\mathbf{0}$ for some $i \in [1, N]$, which means *"do not modify"*, the operator only performs a partial update.

Definition 9 (Model Overwrite). *The model overwrite operator $\otimes : \mathcal{M} \times \mathcal{M} \mapsto \mathcal{M}$ is defined as*

$$M \otimes M' = \left\{ \left(\bigvee_{\forall (\vec{y}, p_i) \in M^\otimes} p_i, \vec{y} \right) \mid \forall \vec{y} \in \mathcal{Y}_{M^\otimes} \right\}$$

and

$$M^\otimes = \{p_M(\vec{y}) \wedge p_{M'}(\vec{y}')), (\vec{y} \leftarrow \vec{y}' \mid \forall \vec{y} \in \mathcal{Y}_M, \forall \vec{y}' \in \mathcal{Y}_{M'}\}.$$

The $(p, \vec{y})$ pairs in $M^\otimes$ enumerate all possible refinement combinations, and $M^\otimes \subset \mathcal{Y} \times \mathcal{P}$ satisfies all properties of an inverse model except the uniqueness of output vectors. By ensuring this property, the final result of the model overwrite operator is also an inverse model. We can easily prove that model overwrite is associative but not commutative in the general case.

Lemma 1 (Model Overwrite is Associative.). $(M_1 \otimes M_2) \otimes M_3 = M_1 \otimes (M_2 \otimes M_3)$.

Equivalent Model. If a forward model R and an inverse model M define the same behavior function, we say these two models are *equivalent.*

Definition 10. *Equivalent Inverse Model. We say a well-behaved forward model R and an inverse model M are equivalent, if and only if $\forall x \in \mathcal{X}, \vec{b}_R(x) = \vec{b}_M(x)$*

In practice, the system typically specifies the forward rule R (*e.g.*, forwarding rules installed on the data plane) but not the inverse model M. Thus, we define the inverse model transformation (IMT) problem.

Definition 11 (Inverse Model Transformation). *For a well-behaved forward model R, find $M \in \mathcal{M}$ such that $R \sim M$.*

C.2 Natural Transformation

We lay the foundation of Fast IMT by introducing *natural transformation*. We refer to this approach as *natural transformation* because its computation process is the most natural way to derive an inverse model from a forward model. It is an adoption of the approach introduced in [21].

The idea is to leverage that the inverse model both defines a behavior function and an inverse function. Thus, we can construct M by computing the inverse function of $\vec{b}_R$. First, we find the pre-image for each value of the i-th output, and then the pre-image of an output vector $\vec{y}$ is the intersection of the pre-images of y_i, $\forall i \in [1, N]$. Second, we enumerate all combinations of each y_i and the set of potential $\vec{y}$ contains those whose pre-image is not empty.

	L, N	$\in$	$\mathbb{N}^+$
Rule Set Space	$\mathcal{R}$	=	$\{R \mid R \subseteq \mathbb{N}^+ \times \mathcal{P} \times \mathcal{Y}\}$
Inverse Model Space	$\mathcal{M}$	=	$\{\forall M \subset \mathcal{P} \times \mathcal{Y} \mid \text{im}(M) = 1\}$
Input	x	$\in$	$\mathcal{X} = \{0,1\}^L$
Predicate	p	$\in$	$\mathcal{P} = \mathcal{X} \mapsto \{0,1\}$
Selection	σ_p	=	$\{\forall x \in \mathcal{X} \mid p(x) = 1\}$
Output	$\vec{y}, \vec{0}$	$\in$	$\mathcal{Y} = Y_1 \times \cdots \times Y_N$
Rule set	R, U	$\in$	$\mathcal{R}$
Rule	r	=	$(pr, pred, \vec{y}) \in R$
Rule set for the i-th output	R_i	=	$\{r \in R \mid r.y_i \neq \mathbf{0}\}$
Behavior function	$\vec{b}_R, \vec{b}_M$	:	$\mathcal{X} \mapsto \mathcal{Y}$
i-th output defined by R	b_{R_i}	=	$r.y_i,\ r = \underset{r \in R_i, r.pred(x)=1}{\arg\max} \{r.pr\}$
Output set of R	Y_{R_i}	=	$\{r.y_i \mid \forall r \in R_i\}$
Rules with output y_i in R_i	$R_i^{-1}(y_i)$	=	$\{r \in R_i \mid r.y_i = y_i\}$
Inverse model	M, χ	=	$\{(p, \vec{y})\}_K \in \mathcal{M}$
Inverse behavior function	$\vec{b}_R^{-1}, \vec{b}_M^{-1}$	:	$\mathcal{Y} \mapsto \mathcal{X}^*$
Output space of a model	$\mathcal{Y}_M$	=	$\{\vec{y} \mid \exists (p, \vec{y}) \in M\}$
Output of p in M	$\vec{y}_M(p)$	=	$\vec{y}, \exists (\vec{y}, p) \in M$
Predicate space of M	$\mathcal{P}_M$	=	$\{p \mid \exists (p, \vec{y}) \in M\}$
Predicate of $\vec{y}$ in M	$p_M(\vec{y})$	=	$p, \exists (p, \vec{y}) \in M$
Natural transformation	ϕ	:	$\mathcal{R} \mapsto \mathcal{M}$

$$a \leftarrow_i b = \begin{cases} a & \text{if } b = \mathbf{0}, \\ b & \text{otherwise} \end{cases} \qquad \text{(Overwrite on } Y_i\text{)}$$

$$\vec{y} \leftarrow \vec{y}' = (y_1 \leftarrow_1 y_1', \ \ldots, \ y_N \leftarrow_N y_N') \qquad \text{(Overwrite on } \mathcal{Y}\text{)}$$

$$\text{eff}(r, R_i) \triangleq r.pred \wedge \neg \bigvee_{r' \in R_i, r'.pr > r.pr} r'.pred \qquad \text{(Effective Predicate)}$$

$$\text{vectorize}_i(y) = (\underbrace{0, \ldots, 0}_{i-1}, y, \underbrace{0, \ldots, 0}_{N-i}) \qquad \text{(Output Expansion)}$$

Figure 17: Key notations & basic operations.

For the first step, it can be observed that **each x will be matched by one rule r in R_i,** $\forall i \in [1, N]$. The rule selection, denoted as $r_{R_i} : \mathcal{X} \mapsto \mathbb{N}^+ \times \mathcal{P} \times \mathcal{Y}$, is a function. Thus, one can compute the *effective predicate* (see Figure 17), which represents the pre-image of r, *i.e.*, $r_{R_i}^{-1}(r)$. In the context of networking, the effective predicate represents the union of all packet header values that will be matched by the rule. Based on the computation logic of the forward model, the input must be matched by $r.pred$ and must not be matched by any rule with a higher priority, *i.e.*,

$$\text{eff}(r, R_i) \triangleq r.pred \wedge \neg \bigvee_{r' \in R_i, r'.pr > r.pr} r'.pred.$$

Note this is exactly Equation (1).

Then, for the i-th output, the pre-image of y_i can be computed as the union of all rules whose output is equal to y_i, *i.e.*,

$$\text{PreImage}(y_i) = \bigcup_{r \in R_i^{-1}(y_i)} \sigma_{\text{eff}(r, R_i)} = \sigma_{\bigvee_{r \in R_i^{-1}(y_i)} \text{eff}(r, R_i)},$$

which is exactly Equation (2).

For the second step, instead of directly enumerating all possible combinations, we observe that by expanding $y_i \in Y_i$ to $\mathcal{Y}$ (see Figure 17), and pairing it with the pre-image we just computed for each y_i, an inverse model is obtained for each $i \in [1, N]$. We denote this model as $\Phi_i(R)$ and the enumeration can be equally yet more efficiently computed using the model overwrite operator $\otimes$.

Hence, we define the natural transformation, which is an instance of direct transformation, in Definition 12.

Definition 12 (Natural Transformation). *For a well-behaved rule set R, the natural transformation $\phi : \mathcal{R} \mapsto \mathcal{M}$ is defined as*

$\phi(R) = \Phi_1(R) \otimes \cdots \otimes \Phi_N(R)$ *where*

$$\Phi_i(R) = \left\{ \left(\bigvee_{r \in R_i^{-1}(y_i)} \text{eff}(r, R_i), \text{vectorize}_i(y_i) \right) \mid \forall y_i \in Y_{R_i} \right\}.$$

With the definition of natural transformation, we now prove its equivalence property.

Theorem 1 (Natural Transformation is Equivalent). $R \sim \phi(R)$.

Proof. $\forall x \in \mathcal{X}$, let $\vec{y} = \vec{b}_R(x)$. Given the definition of equivalence, we need to prove that $\vec{b}_{\phi(R)}(x) = \vec{y}$.

Consider the i-th output, given the definition of behavior model of R, there exists exactly one rule r^* that matches x and has the output y_i. Thus, we have

$$x \in \sigma_{r^*.pred} \setminus \bigcup_{r \in R_i, r.pr > r^*.pr} r.pred$$
$$= \sigma_{r^*.pred \wedge \neg \bigvee_{r \in R_i, r.pr > r^*.pr} r.pred} = \sigma_{\text{eff}(r^*, R_i)}.$$

When $N = 1$, $\phi(R) = \Phi_1(R)$. Let $p^* = \bigvee_{r \in R_i^{-1}(y_i)} \text{eff}(r, R_i)$, we have $x \in \sigma_{\text{eff}(r^*, r_i)} \subseteq \sigma_{p^*}$ since $r^* \in R_i^{-1}(y_i)$. Given the definition of the behavior function of an inverse model, we have $\vec{b}_{\phi(R)}(x) = y_i = \vec{b}_R(x)$ and the theorem holds for $N = 1$.

Now consider the case where $N > 1$. Again we know there exists exactly one rule $r_i^* \in R_i$ that matches x and returns the output y_i on the i-th output. Thus, let $p_i^* = \bigvee_{r \in R_i^{-1}(y_i)} \text{eff}(r, R_i)$, we have $x \in \sigma_{p_i^*}$.

Let $p^* = \bigvee_{i \in [1,N]} p_i^*$, its associated output in $\phi(R)$ is $\vec{y}^* = \text{vectorize}_1(y_1) \leftarrow \cdots \leftarrow \text{vectorize}_i(y_i) \leftarrow \cdots \leftarrow \text{vectorize}_N(y_N)$. Clearly, $x \in \bigcap_{i \in [1,N]} \sigma_{p_i^*} = \sigma_{\bigwedge_{i \in [1,N]} p_{p_i^*}} = \sigma_{p^*}$. Given the definition of the behavior function of the inverse model, $\vec{b}_{\phi(R)}(x) = \vec{y}^*$. Note that $\forall j \neq i \in [1, N]$, the i-th element of $\text{vectorize}_j(y_j)$ is $\mathbf{0}$ ("no-overwrite"). Thus, $\vec{b}_{\phi(R)}(x) = \vec{y}^* = \vec{y} = \vec{b}_R(x)$, and the theorem holds for $N > 1$. □

C.3 Fast IMT

We start from the natural transformation and now introduce Fast IMT. We outline the basic idea: we show that Fast IMT can compute a set of *atomic* overwrites for a block of rule updates. While normal overwrites are only associative, atomic overwrites are commutative. Thus, they can be reorganized and aggregated (*i.e.*, with Reduce I and Reduce II), while still guarantee that the final inverse model is equivalent.

Atomic Overwrites Generated by Fast IMT. Let R denote the forward model before the updates, and R' denote the one after the updates. Consider the atomic overwrites generated on the i-th device. Fast IMT computes an atomic overwrite for each rule in R_{diff} (L40 in Algorithm 1), which is either a new rule (L20 in Algorithm 1), or a rule whose priority is smaller than at least one deleted rule (L15 in Algorithm 1). Note that the action is $\{y_i = a_{r_\delta}\}$, which is essentially $\text{vectorize}_i(r_\delta.y_i)$. Fast IMT selects these rules because their effective predicate is potentially *expanding*, *i.e.*, $\sigma_{\text{eff}(r,R)} \subseteq \sigma_{\text{eff}(r,R')}$.

Definition 13 (Expanding Rules). *Let R and R' denote the forward model before and after some updates, a rule is* an expanding rule *in R' if and only if (1) $r \in R' \setminus R$: r is just inserted into R', or (2) $r \in R$ and $\exists r' \in R \setminus R'$, $r'.pr > r.pr$: a rule r' with a higher priority is deleted.*

Definition 14 (Atomic Overwrites). *Let R and R' denote the forward model before and after the updates, and ΔR_i denote the set of expanding rules on the i-th output. The set of atomic overwrites $\Delta M = \bigotimes_{i \in [1,N]} \Delta M_i$, where*

$$\Delta M_i = \bigotimes_{r \in \Delta R_i} \left\{ \left(\text{eff}(r, R_i'), \text{vectorize}_i(a_r) \right), \left(\neg \text{eff}(r, R_i'), \vec{\mathbf{0}} \right) \right\}.$$

We can $\text{eff}(r, R_i')$ *the* master predicate *of the atomic overwrite.*

Atomic Overwrites Guarantee Equivalence. We now show that by applying the atomic overwrites to the inverse model of R, we can get a model that is equivalent of R'. To prove that, we first introduce *equivalent inverse models*, and a few lemmas which can be trivially proved.

Definition 15 (Model Equivalence). *We say two inverse models M_1 and M_2 are equivalent, denoted as $M_1 \equiv M_2$, if and only if $\forall x \in \mathcal{X}$, $\vec{b}_{M_1}(x) = \vec{b}_{M_2}(x)$.*

Lemma 2. *If $R \sim M_1$ and $R \sim M_2$, $M_1 \equiv M_2$.*

Lemma 3. *If $M_1 \equiv M_2$, $M_1 \otimes M \equiv M_2 \otimes M$.*

Lemma 4. *If $M_1 \equiv M_2$ and $R \sim M_1$, $R \sim M_2$.*

Theorem 2. *Let R and R' denote the forward model before and after the update, and ΔM denote the atomic overwrites. If $R \sim M$, $R' \sim M \otimes \Delta M$.*

Proof. The key to the proof is based on the following derivation. First, we already know $R \sim \phi(R)$ (Theorem 1). With Lemma 2, we have $M \equiv \phi(R)$. Then with Lemma 3, we have $M \otimes \Delta M \equiv \phi(R) \otimes \Delta M$. Again, we have $R' \sim \phi(R')$ (Theorem 1). If we can prove that $\phi(R) \otimes \Delta M \equiv \phi(R')$, with Lemma 4, we have $R' \sim M \otimes \Delta M$.

Now we prove that $\phi(R) \otimes \Delta M \equiv \phi(R')$.

Consider $N = 1$. Let ΔR denote the set of expanding rules in R', numbered from 1 to K. Let r_k^δ denote the k-th rule in ΔR. $\forall x \in \mathcal{X}$, assume it is matched by $r' \subset R'$. There are two cases.

First, consider the case where $r' \in \Delta R$. Without loss of generality, assume it is matched by the k-th rule in ΔR, *i.e.*, $r' = r_k^\delta$. Let $y^* = b_R(x)$ and assume x is matched by p^* in $\phi(R)$, we have (p^*, y^*) in $\phi(R)$. Note that the effective predicates of the rules for the same output are mutually exclusive. As x is matched by the k-th rule in ΔR, it will not be matched by any other rule in ΔR, *i.e.*, $\forall j \neq k \in [1, K]$, $x \notin \sigma_{r_j^\delta.pred}$. Thus, $\forall j \neq k \in [1, K]$, x will match the complementary predicate $\neg \text{eff}(r_j^\delta, R')$. Then, we have $b_{\phi(R) \otimes \Delta M}(x) = y^* \leftarrow \mathbf{0} \leftarrow \cdots \leftarrow r_k^\delta.y \leftarrow \cdots \leftarrow \mathbf{0} = y^* \leftarrow r_k^\delta.y = r'.y = b_{\phi(R')}(x)$.

Now consider the case where $r' \notin \Delta R$. Thus, $r' \in R$. This time, $\forall j \in [1, K]$, x will match the complementary predicate $\neg \text{eff}(r_j^\delta, R')$. Then we have $b_{\phi(R) \otimes \Delta M} = r'.y \leftarrow \mathbf{0} \leftarrow \cdots \leftarrow \mathbf{0} = r'.y = b_{\phi(R')}$.

Then we can use induction to prove the theorem. We omit the details here. □

Correctness of MR2. With Theorem 2, we now prove the correctness of MR2, in particular, Reduce I and Reduce II. A key to the proof is that atomic overwrites are commutative. Thus, we can move the atomic overwrites that has the same output vector together. As the model overwrite operator ($\otimes$) is associative (lemma:assoc), they can be computed first without hurting the equivalence. Then the idea is to prove that the result is the same as Reduce I (aggregation by action), and will prove the correctness of Reduce I. Similarly, we move aggregated overwrites that are from different outputs but have the same predicate together. We follow the same strategy to prove Reduce II (aggregation by predicate).

We now present the proofs.

Theorem 3 (Atomic Overwrites are Commutative). *Let χ and χ' denote two atomic overwrites, $M \otimes \chi \otimes \chi' = M \otimes \chi \otimes \chi'$.*

Proof. There are two cases. First, consider that χ and χ' are both on the i-th device and are computed for $r \in R_i'$ and $r' \in R_i'$ respectively. We already show in Theorem 2 that $\text{eff}(r, R_i') \neq \text{eff}(r', R_i')$. Let $p = \text{eff}(r, R_i')$ and $\vec{y} = \text{vectorize}_i(r.y_i)$, and $p' = \text{eff}\, r', R_i'$ and $\vec{y}' = \text{vectorize}_i(r'.y_i)$, we have

$$\begin{bmatrix} (p, \vec{y}) \\ (\neg p, \vec{\mathbf{0}}) \end{bmatrix} \otimes \begin{bmatrix} (p', \vec{y}') \\ (\neg p', \vec{\mathbf{0}}) \end{bmatrix} = \begin{bmatrix} (p, \vec{y}) \\ (p', \vec{y}') \\ (\neg\,(p \vee p')\,, \vec{\mathbf{0}}) \end{bmatrix} = \begin{bmatrix} (p', \vec{y}') \\ (\neg p', \vec{\mathbf{0}}) \end{bmatrix} \otimes \begin{bmatrix} (p, \vec{y}) \\ (\neg p, \vec{\mathbf{0}}) \end{bmatrix}$$

Now consider the case χ and χ' are on different devices. Assume that χ is on the i-th device and computed for $r \in R_i'$, and that χ' is on the j-th device and computed for $r' \in R_j'$ respectively. Let $p = \text{eff}(r, R_i')$ and $\vec{y} = \text{vectorize}_i(r.y_i)$, and $p' = \text{eff}\, r', R_j'$ and $\vec{y}' = \text{vectorize}_j(r'.y_j)$, we have

$$\begin{bmatrix} (p, \vec{y}) \\ (\neg p, \vec{\mathbf{0}}) \end{bmatrix} \otimes \begin{bmatrix} (p', \vec{y}') \\ (\neg p', \vec{\mathbf{0}}) \end{bmatrix} = \begin{bmatrix} (p \wedge p', \vec{y} \leftarrow \vec{y}') \\ (p \wedge \neg p', \vec{y}) \\ (p' \wedge \neg p, \vec{y}') \\ (\neg\,(p \vee p')\,, \vec{\mathbf{0}}) \end{bmatrix},$$

and

$$\begin{bmatrix} (p', \vec{y}') \\ (\neg p', \vec{\mathbf{0}}) \end{bmatrix} \otimes \begin{bmatrix} (p, \vec{y}) \\ (\neg p, \vec{\mathbf{0}}) \end{bmatrix} = \begin{bmatrix} (p \wedge p', \vec{y}' \leftarrow \vec{y}) \\ (p \wedge \neg p', \vec{y}) \\ (p' \wedge \neg p, \vec{y}') \\ (\neg\,(p \vee p')\,, \vec{\mathbf{0}}) \end{bmatrix}.$$

Only the first entry in the product is different. However, as we know, $\neq i$, $\text{vectorize}_j(y_j)_i = \mathbf{0}$, and vice versa. Thus, $\text{vectorize}_i(r.y_i) \leftarrow \text{vectorize}_j(r'.y_j) = \text{vectorize}_j(r'.y_j) \leftarrow \text{vectorize}_i(r.y_i)$. □

Theorem 4 (Correctness of Reduce I). *Assume there are K atomic overwrites on the i-th device with the same output $\text{vectorize}_i(y_i)$: $\chi_1, \ldots, \chi_K$, and p_k is the master predicate (see Definition 14) of the k-th atomic overwrite.*

$$\bigotimes_{k\in[1,K]} \chi_k = \begin{Bmatrix} (\bigvee_{k\in[1,K]} p_k, \text{vectorize}_i(y_i)) \\ (\neg(\bigvee_{k\in[1,K]} p_k, \vec{\mathbf{0}}) \end{Bmatrix}$$

Proof. The proof is quite straight-forward, according to the definition of model overwrite operator (Definition 9). □

Theorem 5 (Correctness of Reduce II). *Assume there are K overwrites $\chi_1, \ldots, \chi_K$, with the same master predicate p, and the output vector of the k-th overwrite is $\text{vectorize}_{i_k}(y_{i_k})$.*

$$\bigotimes_{k\in[1,K]} \chi_k = \begin{Bmatrix} (p, \vec{y}^*) \\ (\neg p, \vec{\mathbf{0}}) \end{Bmatrix}$$

where $\vec{y}^*$ is defined as

$$y_i^* = \begin{cases} y_{i_k} & \text{if } \exists k \in [1,K], i_k = i \\ \mathbf{0} & \text{otherwise} \end{cases}, \forall i \in [1,N]$$

D APPENDIX FOR CONSISTENT, EFFICIENT EARLY DETECTION

D.1 Consistent Model Construction for Vector-Based Control Planes

We extend the mechanism presented in Section 4.1 for state-sync protocols to vector based control planes (*e.g.*, BGP). In particular, inspired by the distributed convergence detection mechanism for interdomain routing [68], Flash lets switches running vector based control planes append causal relation information: what is the direct cause of an FIB update (*e.g.*, receiving a BGP announcement), and what is the immediate action after computing an FIB update (*e.g.*, sending a BGP announcement), in every FIB update sent to the dispatcher. With such information, the dispatcher can then use a centralized version of the convergence detection algorithm in [68] to decide what FIB updates belong to the same event and hence need to be put in the same model.

D.2 FCPV Algorithm for Regular Expression Based Requirements

The basic pseudocode of fast, consistent partial verification of regular expression based requirement is in Algorithm 2. The algorithm maintains a global data structure, *ecTable*, which stores a product graph for each equivalent class. In the beginning, it contains an entry whose EC matches all packet headers, and the product graph is complete (L1). Once a new model is ready, the algorithm iterates through all equivalent classes to update their product graph and verify reachability (L6-16). Specifically, it first checks whether the product graph for *ec* already exists. If not, *ec* is split from an old one, denoted as *ec*′ (L9-10), the algorithm duplicates the product graph of *ec*′ and creates a new entry for *ec*.[12] Then the algorithm updates the product graph for *ec* by pruning the edges that are not compatible with the *ec*'s action on each newly synchronized node v (L12-13). If an *ec* fails the reachability test, an error is reported that the requirement cannot be satisfied. After processing all current equivalent classes, the split ECs are removed from *ecTable* (L17-18).

Handle anycast and multicast. Algorithm 2 is presented implicitly assuming requirements are on unicast flows, *i.e.*, there is only one accept state in the product graph. When the requirement is for anycast or multicast flows, a product graph has a set of K destinations (*i.e.*, accept states). In such cases, Algorithm 2 needs to be changed in Line 17. For anycast, given one source *src*, out of K *reachable*(src, dst) queries, there must be one and only one returning true; otherwise, an error is found early. For multicast,

[12] This is always correct because in each epoch, the FIB updates belong to a set of newly synchronized nodes, *i.e.*, the outputs on these nodes in the old model are the same ($\mathbf{0}$, actually). Thus, an old EC either maps to a new EC if there is exactly one entry in the update model with the same predicate, or is split to multiple ECs otherwise.

Algorithm 2: Fast Consistent Partial Verification for Regular-expression Requirements.

```
1  Initialization: ecTable = {H ↦ CreateProductGraph(G, rexpr, Port_in)};
2  Function ConsistentPartialVerification(ctx):
3      EC ← GetEC(ctx.M);
4      Δsync ← ctx.sync \ sync;
5      sync ← ctx.sync, D ← ∅;
6      foreach ec ∈ {ec ∈ EC | ec ∩ H ≠ ∅} do
7          if ec ∉ ecTable then
8              (ec', G'_P) ← FindEntryToSplit(ecTable);
9              ecTable[ec] ← G_P;
10             D ← D ∪ {ec'};
11         end
12         G_P ← ecTable[ec];
13         foreach v ∈ Δsync do
14             G_P ← PruneIncompatibleEdges(G_P, ec, v);
15         end
16         ecTable[ec] ← G_P;
17         if not Reachable(ec, G_P) then
18             return Unsatisfied
19         end
20     end
21     foreach ec' ∈ D do
22         Delete(ecTable[ec']);
23     end
24     return UNKNOWN
```

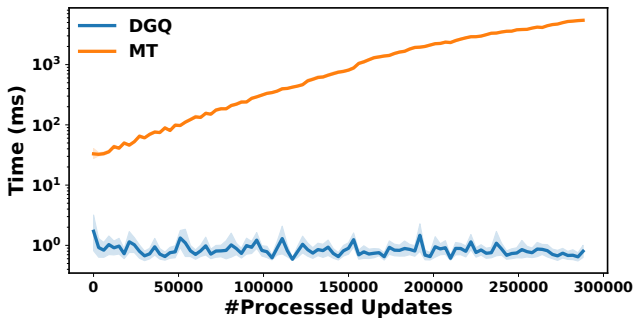


Figure 18: Verification time after processing different numbers of updates.

given one source *src*, all K $reachable(src, dst)$ queries must return true; otherwise, an error is found early.

Handle coverage requirements. Algorithm 2 focuses on "existence" requirements, *i.e.*, a valid path that match a regular expression must exist. Another category of important requirements is the "coverage" requirements, *i.e.*, all paths that match a regular expression must exist [27]. One example of such requirements is the intent "all redundant shortest paths should be available" in Azure [27]. Flash handles this type of requirements by first constructing the product graph, based on regular expressions. Then the early detection of coverage requirement is equivalent to check whether each node in the product graph has an FIB that forwards to all its neighbors in the product graph at all time; If not, an error is found early.

D.3 FCPV Algorithm for All-Pair Loop-Freeness

The pseudocode of the algorithm is shown in Algorithm 3. It first constructs the hyper graph (L2), and then starts to detect loops (L6).

Algorithm 3: Fast Consistent Partial Loop Detection

```
1  Function ConsistentPartialLoopDetect(ctx)
2      G_hyper ← BuildHyperGraph(G, ctx.sync)
3      Δsync ← ctx.sync \ sync
4      sync ← ctx.sync, potentialResults ← ∅
5      for v ∈ Δsync do
6          results ← DetectLoop(v, {ec ∈ ctx.M}, ∅, false)
7          if a deterministic loop is found then
8              return {Loop}
9          potentialResults ← potentialResults ∪ results
10     return potentialResults
11 Function DetectLoop(v, EC, path, hyper)
12     if EC = ∅ then
13         return ∅
14     potentialResults ← ∅
15     if v is external then
16         return {NoLoop}
17     else if v.is_hyper and v.is_biconnected then
18         potentialResults ← potentialResults ∪ {Loop}
19     else if v ∈ path and hyper = true then
20         return {Loop}
21     else if v ∈ path and hyper = false then
22         abort Loop
23     foreach (v, u) ∈ G_hyper do
24         validEC ← EC ∩ EC(v, u), path' ← path ∪ {v}
25         r ← DetectLoop(u, validEC, path', hyper ∨ u.hyper)
26         potentialResults ← potentialResults ∪ r
27     return potentialResults
```

If a deterministic loop, *i.e.*, loop with only synchronized nodes, is found (L7-8, L22), it gives the consistent early detection result of a loop. The detection function verifies a path where the last node is v, for a set of equivalence classes *EC*. *path* denotes the path segment without attaching v and *hyper* denotes whether the path contains a hyper node. The detection method checks various conditions (L15-22) to determine whether the result of the path is already known. If not, it extends the path to each potential next hop u for potential ECs that can take the path from v to u (L24), and collects all potential results of those extended paths (L25-26).

D.4 Proof of Achieving Consistent Early Detection

We first define consistent, early detection. Let M_K denote the inverse model of the final data plane state after applying K updates, and M_k be the model after applying only $k < K$ updates. Early detection means that a verification function *ver* checks a requirement *req* on model M_k, *i.e.*, $ver(req, M_k)$. We focus on the common verification requirements, *i.e.*, regular expression based path requirements and loop-freeness. Since M_k only has partial information of the network, $ver(req, M_k)$ can give either a concrete result (*i.e.*, satisfied/unsatisfied) or *unknown*.

Definition 16 (Consistent early detection). *For a verification requirement req, for any initial model M and K updates, if* $\exists k \leq K$, $ver(M_k, req)$ *gives a concrete result and* $ver(M_k, req) = ver(M_{k+1}, req) = \cdots = ver(M_K, req)$, *we say that ver achieves consistent early detection for req.*

We prove that the verification approach of Flash on the consistent model achieves consistent early detection. Formally, given a

consistent inverse model M_k. For a loop detection requirement req_l, let ver_l denote the verification function that checking loops using the hyper abstraction in §4.3, when $ver_l(M_k, req_l)$ gives concrete result, ver_l achieves consistent early detection.

Proof. If ver_l does not achieve consistent early detection, then according to Definition 16, $\exists k' > k : ver_l(M_k, req_l) \neq ver_l(M_{k'}, req_l)$. That means the loop check result is changed after applies k' updates. Since ver_l only checks loops among synchronized nodes, the only way to break a loop is to update the FIB of the synchronized nodes, however, M_k is a consistent model that guarantees no updates on synchronized nodes, thus such k' doesn't exist. □

For checking regular expression based path requirements on consistent models, the proof is similar (doesn't exist k' to change a path among the synchronized nodes) and omitted.

E COMPLEMENTARY EVALUATION RESULTS

E.1 Analysis on Decremental Verification Graph

In addition to the results in §5.4, we show how the verification time changes as the number of processed updates increases. The results are in Figure 18. As we can see, the verification time of MT increases as more rule updates are finished, while the trend does not show in the DGQ approach. The reason is that the computation complexity of model traversal is $O(|V|\times(|V|+|E|))$, and as more rule updates are finished, $|E|$ is increasing. Meanwhile, DGQ computes connected components of the verification graph in the beginning, whose complexity is $O(|V|+|E|)$, and $|E|$ is decreasing as more rule updates are finished.

F ARTIFACT APPENDIX

Abstract

The artifact provides an implementation of Flash using Java. It includes all key components in the paper and the necessary datasets for reproducing the evaluation results in the paper.

Scope

The artifact allows to validate the following evaluation results:

(1) The effects of Fast IMT: Table 3.
(2) The effects of CE2D: Figure 8, Figure 9, Figure 10, and Figure 12.
(3) The micro benchmark: Figure 7 and Figure 11.

Note that the exact values may vary on different machines (even with the same CPU and memory configuration).

The artifact is only allowed for research purpose.

Contents

The artifact includes the following contents:

(1) An implementation of Flash.
(2) The implementations of APKeep [26] and Delta-net [25] according to their pseudocode.
(3) The datasets (LNet topology is anonymized) evaluated in the paper.

Hosting

The artifact is hosted at GitHub (commit *b22993* on the *sigcom22-artifact* branch).

Requirements

A server with at least 32GB free memory is required to run the artifact. And the following dependencies are required: JDK 17, Maven 3.8+, and Python 3.6+. We recommend using Ubuntu Server(x64) 18.04.4 LTS, which has been tested, as the operating system and using the scripts in the artifact to install all necessary software.