# Tempus: Probabilistic Network Latency Verifier

Anonymous Author(s)

## ABSTRACT

To combat problems and bugs that a given network might have, network verifiers have emerged as one of the promising solutions. State-of-the-art network verifiers mainly focused on evaluating qualitative properties under various scenarios of network failures, such as reachability under k-link failure. However, as modern networks evolved and performance need becomes more stringent – often expressed in terms of Service Level Agreement (SLA) – there is a need to evolve network verifiers to also reason about quantitative performance properties. Works in quantitative network verifiers that has arose in recent years mainly focused on one side of the network performance metric: bandwidth and load violation properties. Questions about the other side of network performance metric, latency, were left unanswered.

In this work, we introduce a verifier framework, Tempus, that can answer the probability of a given temporal property being true given latency distributions of individual links and nodes in the network. Early evaluation shows that Tempus can verify bounded reachability property – the probability that the average delay of packets traversing from a source to destination node is below $T$ time unit – by only adding a fraction of the state exploration overhead introduced by the qualitative verifier it's built upon.

## 1 INTRODUCTION

Modern networks consisted of various distributed protocols such as OSPF and BGP that exchange routing information so that a packet can reach their intended destination efficiently, even in the event of a component failure. Due to their configuration intricacies however, these protocols are notoriously hard to get right. It is hard for a network engineer to reason about whether their configured network fulfilled some intended property in various possible states of the network. Thus, they are left between the choice of accepting the reality of Murphy's Law or getting a tool to help them in this task, namely, network control plane verifier.

Popular control plane verifiers, like ARC [1], are formulated to answer questions about a given property deterministically. In other words, given a set of states of the network (e.g. $k$-link failures) the verifier are expected to give a yes or no answer about the property (e.g. two nodes are reachable). While useful to some extent, this kind of models are sometimes too restrictive since network operators are usually able to tolerate small fraction of failure. For example, a given network might have an availability SLA of 99.999%. This kind of probabilistic properties have been the focus of a more recent works like NetDice [2].

The network property itself can also be divided into two kinds. Quantitative properties like reachability and loop existence, are the common properties that are studied by most of the existing verifiers. More recent work, like QARC [3], has also explored qualitative properties like link bandwidth violation for a given traffic.

In this work, we're exploring the other side of the network performance metric: latency. Certain network deployment often necessitates some latency requirement such as an ISP that has latency SLA [4] or deployments of Time-Sensitive Networking (TSN) [5]. We proposed a verification framework to probabilistically verify the latency property of packets traversing from a source to destination node under various failure scenarios, by using latency information of each components in the network.

## 2 OVERVIEW

### 2.1 Latency as Path Property

We began our study by first pondering about a basic construct: modeling the end-to-end latency of a single packet. In a packet-switched network, a packet is sent from one transmiting host to one receiving host through the many components of the network (e.g. links, routers, firewall) and each of those components might introduce some latency into the packet transmission process. Figuring out which of those components will actually introduce latency into the packet in question, and by how much, is the next logical step that we must figure out.

Obviously, a given packet doesn't need to visit all network components to reach its destination host. A network engineer will configure the network in such a way that a packet will only need to be routed via a specific subset of its components. That specific subset is dictated by two things: how those components are connected together (i.e. topology) and how the control plane protocols are configured to route a packet (e.g. routing protocol, ACL). Based on a variety of these setups, nodes in the network will form a forwarding table to route a given packet appropriately.

The goal of a classical control plane verifier then, is to use this forwarding table in some form to verify certain properties. However, looking at the forwarding table alone will not give us a conclusive result regarding which components

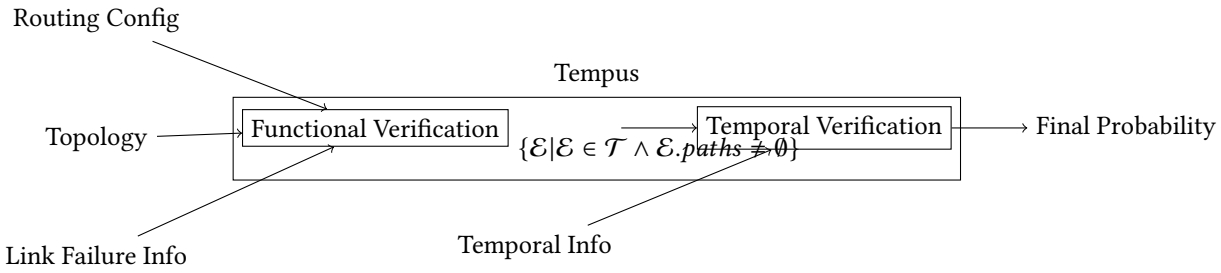


**Figure 1: Overview of Tempus**

are going to be visited by a given packet, making verification of latency properties less clear. For example, the network might be configured with a load-balancing protocol in which a packet departing from a source host might take multiple different *paths* (with certain probability weights) to arrive at the destination host, possibly resulting in a different end-to-end latency measurement.

Therefore, we argue that when it comes to analyzing end-to-end latency, a network path should be the primary unit of reasoning, rather than forwarding table. By being more specific about our unit of reasoning, we could answer verification questions more clearly, and we could design our verifier more efficiently since we could use it to represent multiple different forwarding tables that shares the same path.

## 2.2 Relation to Classical Verifier

Before we analyze the latency of a given packet that propagates through a certain network path however, we must make sure that said path exists in the first place. We note that latency is a property that only make sense after connectivity between two hosts has been established. In other words, if two nodes in a network aren't even functionally connected (e.g. physical link failure, ACL policy), then the latency between them will *always* be infinite, making the verification task trivial.

Fortunately, there are a rich body of work in the network verification literature regarding functional reachability under failure. We could then design our verifier on top of an existing classical control plane verifier. We use it to verify reachability property, and only if the reachability property is fulfilled, we would verify whether the latency between two hosts fulfilled some additional condition.

## 2.3 Path Latency Distribution

Up to this point, we have talked about measuring the latency of a single packet by figuring out its path; analyzing which exact components it has traversed through. However, when we try to generalize this framework and ask questions about the latency of multiple packets, it is apparent that the path alone is not a determinative information, as the latency of two packets propagating through the same path might be different due to a multitude of factors.

The natural extension to the framework then, is instead of representing latency of a path with a single number, we instead represent it with a continuous random variable that signifies the possible delay that a given packet traversing through that path might have. This random variable will have a distribution that marginalizes over all other factors other than the path.

We can then use this latency distribution to verify some temporal properties in a probabilistic manner. For example, we could verify the probability that a packet will be delivered in under a time unit by taking the CDF of the distribution.

## 2.4 Path Decomposition and Convolution

The final question that we had to decide on was how do we actually model path latency distribution based on real component measurements. Considering the complexity of factors that might determine the path latency distribution and the availability of measurement data, we settle on the assumption that the path latency distribution is composed of multiple independent distribution that corresponds to the latency each components in the path might introduce.

Since not all components in the path will introduce latency with a non-negligible value, we specifically choose to model two source of latency in the path's components which we deem significant: **link propagation latency** and **queuing latency**.

Propagation delay is the latency that is introduced by the links in the network, which is independent of the traffic load in the system. Queuing delay is the latency that the queuing process in the node introduced. Unlike propagation delay, queuing delay might be dependent on load in the system, since the more packets there are in the queue, the more delay the node will introduce to a subsequent packets.

For propagation delay, the semantic of this random variable is relatively straightforward: it is the distribution of latency that a given link will introduce. For queuing delay however, this random variable represents the delay that a given queuing process will introduce, marginalized over various traffic pattern that a given network state might have resulted.

In order to obtain the overall path latency distribution from these per-component latency distributions, we do a convolution operations over all the relevant component's latency distributions. Since not all distributions can be convolved in a closed-form fashion, we use a numerical convolution technique with a guaranteed error bound. We initially consider a Monte-Carlo simulation in order to approach numerical convolution, but the lack of a general technique to measure error made us opt for the formerly mentioned method.

We divide the problem of latency verification into two parts: verifying that two nodes are reachable (**functional verification**) and only if the functional property is fulfilled, we would verify whether the latency between two nodes fulfilled some temporal condition (**temporal verification**).

## 3 FUNCTIONAL VERIFICATION

For functional property verification, NetDice [2] has laid the way for verifying reachability between two nodes under failure in a probabilistic and efficient manner. To contextualize our modification, we will briefly describe some parts of NetDice's network encoding and exploration algorithm as a prelude to our alteration. For a more complete complete description of NetDice, please refer to the original paper.

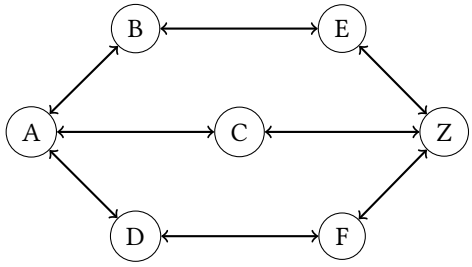


**Figure 2: Example topology, we want to check the reachability of A to Z**

### 3.1 Topology Graph

In this framework, a network is encoded in an edge-labeled directed graph $G_t = (V_t, E_t)$ where $V_t$ represents the nodes in the network and $E_t$ represents the functional connectivity between a source and a destination node. A physical link is then represented as a pair of symmetrical edges that shares the same source and destination node but with opposite direction (e.g. $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_1$). A routing protocol is then defined on top of this graph by some auxilary information.

To explain it with further clarity, we use the topology in Fig. 2 as a running example for this paper. We will assume that the nodes in the network runs OSPF with weight 1 for every edge and ECMP as their load-balancing scheme. We want to check the temporal probability of packets that departs from A to Z.

To represent random failure, NetDice defined a universal link failure rate (i.e. the probability of *any* link in the network randomly failing). We refine NetDice's model slightly by allowing each links to have different failure rate. We label each edge in $E_t$ with a function $r : E_t \rightarrow \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ that represents the failure rate of a given physical link. As a consequence, two symmetrical edges (i.e. two edges that shares the same node pairs but with opposite direction) will also share the same failure rate, and will be disabled in a coupled fashion.

For the sake of example, in our Fig. 2 topology, we will assume that each link has a 10% chance of failure.

### 3.2 State Exploration

A **network state** is defined as a set of failed links. There are $2^{|E_t|}$ network states in a given network and a brute-force strategy would need to explore all of the states to compute the reachability property of a node pair in a given network. NetDice however, will merge multiple network states into an **equivalence class** by marginalizing over *cold edges*, links whose failure is guaranteed not to change the convergent path of a highlighted network state.

NetDice will systematically explore many equivalence classes in the network. It will start from the equivalence class of a perfect network and will fail certain *hot edges* (i.e. edges that are not cold) to explore another equivalence class. While not explicit in its description, NetDice's algorithm will effectively form an **exploration tree**.

In order to preserve some information that will be used in the temporal verification stage, we modify this original exploration algorithm to make it more explicit. We started by formalizing the notion of Equivalence Classes. An Equivalence Class $\mathcal{E}$ is defined as a 3-tuple $(U_h, D_h, paths)$. $U_h$ and $D_h$ refers to a set of hot edges that are up and down respectively. $paths$ refers to the convergent paths between src-dst pair that is produced by the control plane when the links in $D_h$ fails. Unlike the original algorithm, we explicitly store $paths$ in $\mathcal{E}$ so that it could be used to for the temporal verification stage.

We then define the Exploration Tree $\mathcal{T}$ as a tree of Equivalence Classes. At the root, we have an Equivalence Class that

corresponds to a perfect network, where *paths* is the path(s) that the control plane will produce given a perfect network condition and $U_h = D_h = \emptyset$. Each Equivalence Class in this tree will have children which have one of its parent's link in *paths* appended to its $D_h$, essentially failing one of the link in the path. If an Equivalence Class has an empty *paths*, then it would have no children.

To compute the functional probability $P_f$ of a given Equivalence Class, we compute the product of the 'up' probability of all the links in $U_h$ and *paths* and the product of the 'down' probability of all the links in $D_h$.

In our running example, we could see the subset of the exploration tree in Fig. 3. We see that in a perfect network ($\mathcal{E}_1$), OSPF will produce *ACZ* as the shortest path between A to Z. The probability of this Equivalence Class actually materializing ($P_f$) is 0.81, which is the probability of *AC* and *CZ* being up at the same time ($0.9^2$).

In the Equivalence Class where the link *AC* is failing however ($\mathcal{E}_2$), OSPF and ECMP will produce two equaly-weighted paths *ABEZ* and *ADFZ*. The probability of this Equivalence Class actually materializing ($P_f$) is 0.053 ($0.9^6 \cdot 0.1$).

After having the algorithm produced $\mathcal{T}$, we then continue to the temporal verification stage.

# 4 TEMPORAL VERIFICATION

In the temporal verification stage, we will use the path information that we got from functional verification stage and augment them with latency information to produce the final temporal property probability.

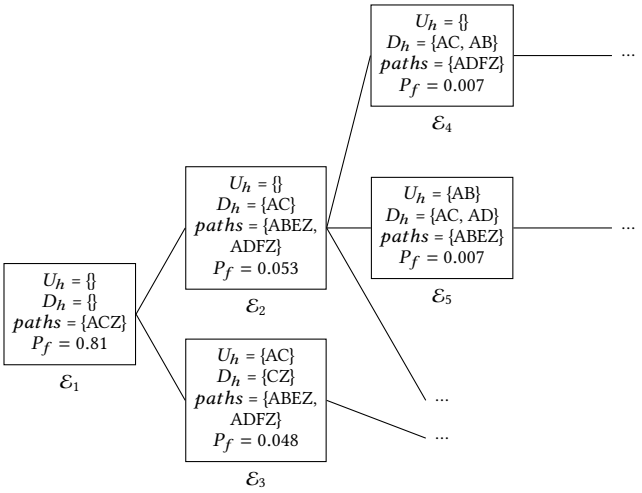


**Figure 3: Subset of Modified Exploration Tree $\mathcal{T}$ for our example topology, $\mathcal{E}_2$ and $\mathcal{E}_3$ will get consolidated**

## 4.1 Latency Label

Similar to link failure $r$, we use edge-labeling technique to represent latency that will get introduced by the connectivity between two nodes. Our model will model two significant sources of latency: **link propagation** and **packet queuing in the node**. We will encode these latencies by equipping the model with two additional labels.

Encoding link propagation latency is fairly straightforward. We define a function $l_p : E_t \rightarrow \mathcal{D}$ where $\mathcal{D}$ is a set of continuous univariate distribution that has a minimum value of 0. This distribution signifies the time it would take for the respective physical link to transmit a packet from one end to another. Because of this, just like $r$, two symmetrical edges will share the same distribution.

To encode queuing latency, we first note that the queuing mechanism in modern switches usually resides in the output port. Since a port in a switch is only connected to one other port, we could effectively assign the latency to the connectivity between switches. To do this, we define another function $l_q : E_t \rightarrow \mathcal{D}$. This distribution signifies the output queue latency in the source node that is going to be forwarded to the destination node. Unlike $l_p$, two symmetrical edges will have two different distributions since they represent two different output queues.

In our running example, we will assume that all edges in the topology will have $l_p$ and $l_q$ of gamma distribution with $\lambda = 1$, except $l_q(EZ)$ which will have a lognormal distribution with $\alpha = 1$ and $\theta = 1$.

Given *paths*, $l_p$, and $l_q$, we could then compute $P_t$, the probability of a given temporal property being fulfilled. We will describe our algorithm for computing this probability in the next few sections.

## 4.2 Weighted Average and Path Convolution

Since a convergent behavior of the forwarding plane *paths* might contain more than one paths, we will compute the temporal probability of *paths* by computing the weighted average of the temporal probability of each of the individual path. The specific weight depends on the load-balancing method used.

To get the temporal probability of a single path, we will use the latency information from $l_p$ and $l_q$ to compute the latency distribution of the whole path. To do this, we resort to the methods of **convolution**. For each edge $e$ in the path, we will get $l_p(e)$ and $l_q(e)$ and convolve them all together to get the path latency distribution.

With this path latency distribution, we could get the temporal probability of a path by computing the statistical property of said distribution. In this work, given a path latency distribution $\mathcal{L}$, we define two temporal properties:

- **Bounded Reachability**: the probability that a packet will get transmited below $t$ time unit. This will get computed as $cdf(\mathcal{L}, t)$
- **Tail Reachability**: the probability that a packet will get transmited above $t$ time unit. This will get computed as $1 - cdf(\mathcal{L}, t)$

In our Fig. 3 Exploration Tree for example, $\mathcal{E}_2$ has two equally probable path, *ABEZ* and *ADFZ*. We will first do a chain convolution of all the distributions in *ABEZ* ($l_p(AB)$, $l_q(AB)$, ..., $l_q(EZ)$) and do the same thing for *ADFZ*. We will then take the *CDF* of $\mathcal{L}_{ABEZ}$ and $\mathcal{L}_{ADFZ}$ with our desired $t$ and average them out to get $P_t$ of $\mathcal{E}_2$.

In short, given *paths*, $l_p$, and $l_q$, we will do the following:

(1) Split *paths* into its individual path
(2) For each path, compute its latency distribution by convolving the latency distribution $l_p$ and $l_q$ of each link
(3) With the resulting path latency distribution, determine the probability of temporal property by computing the statistical property of said distribution
(4) Combine the temporal property of each path by computing the weighted average, based on the load-balancing scheme of the control plane

We do each of these steps to every Equivalence Class in $\mathcal{T}$, and do a sum-product operation of $P_f$ and $P_t$ to get the final probability of the property in question.

By introducing the temporal verification algorithm, we're essentially adding additional overhead to the exploration algorithm (on top of functional verification) that scales to the size of the Exploration Tree. In other words, for $\mathcal{T}$ of size $n$, we will need to do temporal verification in each of those $n$ Equivalence Class. We aim to minimize the overhead of temporal verification by introducing some optimization algorithms later.

### 4.3 Numeric Convolution

However, there is one major problem with a convolution-based technique for computing the latency distribution of a given path: not every distribution pair can be convolved analytically. A closed-form solution of a convolution is usually only available for two distributions of the same type.

In our example, we define $l_q(EZ)$ to be lognormal distribution, which doesn't have a closed form convolution solution with the other latency distribution in the network, which is of gamma distribution form. Therefore, in order to convolve two arbitrary distributions, we need to leverage a numerical convolution method.

We leverage an existing numerical convolution algorithm, DIRECT, to fulfill this role. We choose this method due to their bounded error property: DIRECT guarantee that the computed distribution and the correct theoretical distribution has a KL-divergence below a certain bound. DIRECT has also been implemented in R's popular bayesmeta package.

One subtle detail about DIRECT is while convolution is defined to be a commutative operation, and DIRECT also achieves this property, the order of operation matters for the algorithm's runtime performance. In particular, if we had a chain of DIRECT convolutions, the result of a convolution should not be set as the second input distribution in the later convolutions to avoid performance penalty. This is caused by a nested loop from the following two mechanisms:

- DIRECT works by computing a list of support values by doing many PDF queries of the second input distribution
- Computing the PDF of a DIRECT distribution (the returned distribution of a DIRECT algorithm) itself involves iterating over the current support values

As an ilustration, if we had 3 random variables $A$, $B$, and $C$, and we want to convolve them all together, it would be faster to compute $direct(direct(A, B), C)$ instead of $direct(C, direct(A, B))$

While this problem is trivial to solve (swap the input parameter if the latter is detected as a DIRECT distribution), it does limit the type of optimization that we could have in the chain convolution process.

## 5 OPTIMIZATION

### 5.1 Equivalence Class Consolidation

To minimize the amount of temporal verification overhead, we want to find some symmetry in the existing exploration algorithm so that we could reduce the exploration space even further.

In the original NetDice exploration algorithm, multiple network states were merged into an Equivalence Class based on its cold edges (edges whose failure won't change the convergent path(s)). In other words, network states within the same Equivalence Class will have the same convergent path(s).

However, we observe that each of those Equivalence Classes are not *unique*: while network states within the same equivalence class shares a convergent path(s), **multiple equivalence classes could also share the exact same convergent path(s)**, as shown in $\mathcal{E}_2$ and $\mathcal{E}_3$ in Fig. 3.

Since we define latency distribution to marginalize over all factors other than the path, we could effectively *consolidate* these equivalence classes into one big equivalence class and do temporal verification once.

We implemented this idea by doing memoization on the temporal probability of a given convergent path(s). We will only do temporal probability computation once, when we

first iterate over an equivalence class that has a certain convergent path(s), and we cached the result should another equivalence class with the same convergent path(s) emerged.

## 5.2 Path Memoization

After consolidating many equivalence classes that shares the same convergent path(s), we are left with fewer, bigger, but unique equivalence classes. As significant as it is, we could reduce the computation cost further by drawing connections between these unique equivalence classes.

In a network with a load-balancing protocol, the convergent state of the data plane might forward a packet through multiple possible paths. In the context of our framework, we say that an equivalence class might have more than one convergent paths. Nevertheless, we note that two equivalence classes with different convergent paths might not be two independent subset. In other words, **multiple equivalence classes could share a subset of individual path**.

Since the way we compute convergent path(s) temporal probability is by calculating the weighted average of individual path temporal probability, we could reduce the amount of convolution we will need to do by doing further memoization on the temporal probability of a given path. If we were to compute the temporal probability of an equivalence class with previously cached individual path, we only need to calculate the weights that corresponds to the load balancing protocol without doing any convolution.

In our running example, we could see that in Fig. 3, $\mathcal{E}_2$ and $\mathcal{E}_4$ share an identical path, $ADFZ$. We could therefore memoize $cdf(\mathcal{L}_{ADFZ}, t)$ when we explore $\mathcal{E}_2$ and query the cached result when we explore $\mathcal{E}_4$.

With this optimization, we will do a chain convolution by the amount of path available in the network between the source and destination (which is 3 in the case of our running example in Fig. 2: $ACZ$, $ABEF$, and $ADFZ$).

## 5.3 Convolution Grouping

Aside from optimizing the *amount* of paths we need to do convolution on, we could also do some optimization on the *convolution operation itself.*

While the use of numerical technique to do convolution enabled us to use arbitrary probability distribution in our framework, it comes with two noticeable disadvantages.

The first is *error*: numerical method is fundamentally an approximation technique that always comes with some error. While DIRECT algorithm comes with an error-bound guarantee, it is not zero and could expand if we do chain convolution.

The second is *runtime performance*: even with the input ordering constraints we laid out on the previous subsection, DIRECT is still generally slower than analytical convolution.

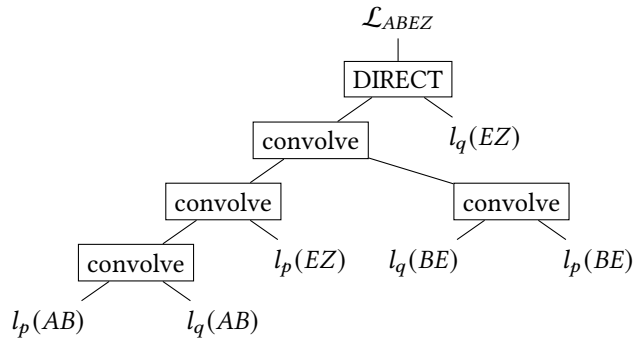


**Figure 4: One example of a convolution tree to get $\mathcal{L}_{ABEZ}$ with optimized grouping, we want to start introducing numerical convolution as late as possible**

This is expected, since a closed form solution of a convolution tipically boils the operation down into a few arithmetic calculation, rather than complex operation like integration. Due to these factors, we feel the need to **prioritize analytical convolution** in the calculation of path latency distribution.

We did this by *grouping* subsets of the path components with the same probability distribution type (e.g. exponential, normal, etc.). We then analytically convolve the distribution within that subset and only numerically convolve the returned distribution, which will be fewer in quantity, thus enabling smaller error and faster performance.

In our running example, in the chain convolution process to get $\mathcal{L}_{ABEZ}$, we will analytically convolve $l_p(AB)$, $l_q(AB)$, $l_p(BE)$, $l_q(BE)$, and $l_p(EZ)$ since they are of the same type (gamma) and then numerically convolve the resulting distribution with $l_q(EZ)$. Fig. 4 illustrates how the chain convolution process will be broken down.

# 6 EVALUATION

To evaluate our idea, we implemented Tempus in ~ 600 lines of Julia code. This code will take a topology configuration, source and destination node, and the temporal property in question and its threshold. It will then output the final probability of the network upholding that temporal property. We ran our prototype on a machine with i7-1065G7 CPU and 16 GB of RAM.

We tested Tempus on 5 real-world WAN topologies from the Topology Zoo and 3 datacenter topologies (fat-tree, with various amount of pods). We use OSPF as the routing protocol with uniform link weights of 1 and 0.1% failure rate in all links, in accordance to the failure rate in previous studies. We then run the functional verification with $10^{-8}$ inaccuracy level.

To evaluate the performance and scalability of Tempus, we first evaluate the running time of the temporal verification on different topologies given a mixture of latency distributions. We then examine the effectiveness of our optimization technique by comparing the number of convolution of the former results with the unoptimized version. Next, we compare the result of the consolidation strategy in terms of the number of equivalence classes, instead of convolution. Finally, we compare the effect of latency distribution type on the performance of Tempus.

Since the CDF operation that we define for temporal property verification is a relatively cheap operation, we expect our result to generalize with any threshold we chose. In other words, while changing the threshold will change the final probability, it won't change the runtime performance of Tempus. Thus, we set an arbitrary threshold for our evaluation.

## 6.1 Runtime Scalability

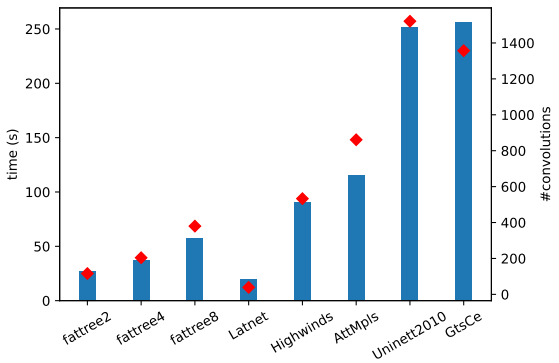


**Figure 5: Runtime performance of temporal verification, red dots represent the number of convolutions**

To begin our evaluation, we first define the $l_p$ and $l_q$ for each component in the topology with varying probability distributions to ensure the robustness of our method. Some distributions (like LogNormal) don't have a closed-form solution for their convolution, even against itself, while others (e.g. Gamma, Chi-Squared) do. Since our algorithm relies on convolution, having a diversity of distributions, especially accross these categories, is important for the representativeness of our evaluation. In light of this, we set all the queuing latency $l_q$ of our topologies' component to be LogNormal and the link latency $l_p$ to be randomly chosen between Gamma and Chi-Squared.

We measure the runtime performance of Tempus by verifying the bounded reachability property of two edge routers. Fig. 5 shows the temporal verification runtime performance (excluding functional verification) and the amount of convolution operation performed in each network. We note that we can precisely verify the temporal property of networks with hundreds of links in the order of minutes.

The convolution step of Tempus takes the most amount of time in the temporal verification process, hence we can see in the same figure that the runtime performance of Tempus and the total amount of convolution is highly correlated.

We conclude that **the performance of Tempus is primarily bottlenecked by the convolution operation** and for a reasonably low imprecision level, **Tempus runs in the order of minutes**.

## 6.2 Optimization Effectiveness

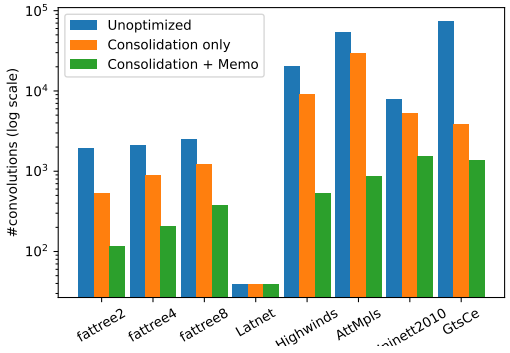


**Figure 6: Amount of convolution (in log scale) depending on what optimization strategy is applied**

We have established that the convolution operation is a relatively expensive operation within Tempus. Now, we inspect the effectiveness of our optimization techniques in reducing them.

Fig. 6 shows the effect of the two optimization techniques, consolidation and memoization, in reducing the amount of convolution (in logarithmic scale). We exclude the third optimization technique (grouping) in this evaluation because they are not designed to reduce the amount of convolution and thus need to be evaluated differently in later subsection.

We could see that in the best case, the optimization strategies reduce the amount of convolution by **98%** (in the case of GtsCe).

- While the amount of path in those states can explain the amount of EC reduction, to get to the amount of convolution, we also need to consider the length of each of those path
- By considering the both amount of path and its length to counting the convolution, we could compare consolidation and memoization strategy to the unoptimized version

- 

## 6.3 Equivalence Class Reduction

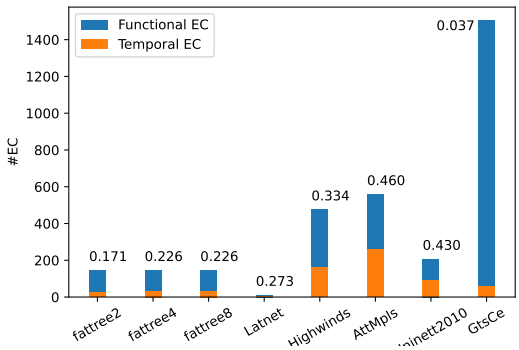


**Figure 7: Amount of temporal EC compared to functional EC, label represents ratio**

- Compare the amount of temporal EC to the functional (unconsolidated) EC
- Corresponds to the consolidation strategy
- **More than 50% reduction**
- Doesn't work equally on all topologies, depend on
  - How many functional EC are explored (for a given inaccuracy level)
  - How many possible paths a given node pair have in those ECs

## 6.4 Effect of Latency Distribution Type

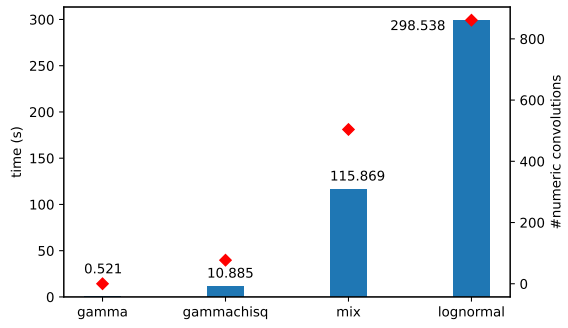


**Figure 8: The effect of distribution type on runtime performance of AttMpls, label represents the performance in seconds and red dots represents the amount of *numerical* convolution**

- While the amount of convolution gave us a pretty good proxy for performance, we could see that there's still some variations
- This is due to the fact that we used two kinds of convolutions: analytical and numerical.
- Since numerical convolution is slower, we could see that for the same topology and the same amount of overall convolution, **more numerical convolution will result in longer runtime performance**.

# 7 RELATED WORKS

- Control plane and data plane verifier
- Probabilistic Verifier
- Quantitative Verifier

# 8 FUTURE WORKS

- Addressing independence: convolution treats each probability distribution as independent with each other. Latency verifier that relaxes this assumption (or a better encoding) would be interesting future works
- Optimizing functional verification: a functional verifiers that produces functional ECs that has already been consolidated would be interesting future works
- Extracting Latency Information from Measurement Data: While Tempus serve as a framework for latency verification, future work needs to be done in translating measurement data into latency distributions. Julia supports distribution fitting given a set of data.

# 9 CONCLUSION

- Tempus is a scalable latency SLA verifier. It could probabilistically verify in the order of minutes
- Consolidation and memoization is an effective optimization strategy, although the exact magnitude of its effectiveness will depend on the network topology, specifically on the amount of paths and their length.
- The grouping optimization strategy is an effective optimization strategy, although the exact magnitude of its effectiveness will depend on the amount of similar distributions that can be convolved analytically

# REFERENCES

[1] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 300–313, 2016.

[2] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," 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*, pp. 750–764, 2020.

[3] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 974–988, 2020.

[4] "Verizon global latency and packet delivery sla." https://www.verizon.com/business/terms/global_latency_sla/. Accessed: 2022-09-20.

[5] "Tsn." https://1.ieee802.org/tsn/. Accessed: 2022-09-20.