Resolving Packets from Counters: Enabling Multi-scale Network Traffic Super Resolution via Composable Large Traffic Model

Xizheng Wang, *Tsinghua University and Zhongguancun Laboratory;*
Libin Liu and Li Chen, *Zhongguancun Laboratory;* Dan Li, *Tsinghua University;*
Yukai Miao and Yu Bai, *Zhongguancun Laboratory*

https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-resolving

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Resolving Packets from Counters: Enabling Multi-scale Network Traffic Super Resolution via Composable Large Traffic Model

Xizheng Wang[1,2], Libin Liu[2], Li Chen[2], Dan Li[1], Yukai Miao[2], Yu Bai[2]

[1]*Tsinghua University* [2]*Zhongguancun Laboratory*

Abstract

Realistic fine-grained traffic traces are valuable to numerous applications in both academia and industry. However, obtaining them directly from devices is significantly challenging, while coarse-grained counters are readily available on almost all network devices. None of existing work can restore fine-grained traffic traces from counters, which we call network traffic super-resolution (TSR). To this end, we propose ZoomSynth, the first TSR system that can achieve packet-level trace synthesis with counter traces as input. Following the basic structure of the TSR task, we design the Granular Traffic Transformer (GTT) model and the Composable Large Traffic Model (CLTM). CLTM is a tree of GTT models, and the GTT models in each layer perform upscaling on a particular granularity, which allows each GTT model to capture the traffic characteristics at this resolution. Using CLTM, we synthesize fine-grained traces from counters. We also leverage a rule-following model to comprehend counter rules (*e.g.* ACLs) when available, guiding the generations of fine-grained traces. We implement ZoomSynth and perform extensive evaluations. Results show that, with only second-level counter traces, ZoomSynth achieves synthesis quality comparable to existing solutions that takes packet-level traces as input. CLTM can also be fine-tuned to support downstream tasks. For example, ZoomSynth with fine-tuned CLTM outperforms the existing solution by 27.5% and 9.8% in anomaly detection and service recognition tasks, respectively. To promote future research, we release the pre-trained CLTM-1.8B model weights along with its source code.

1 Introduction

Fine-grained traffic traces are valuable to numerous applications in both academia and industry. Packet-level traces can benefit performance evaluation of systems and algorithms in all layers of the networking stack, from packet scheduling to congestion control. Realistic traffic traces are also useful in optimizing various tasks, such as traffic engineering [48], network telemetry [53, 54], anomaly detection [75], and service recognition [27]. However, obtaining fine-grained traffic traces from devices is challenging due to factors such as business confidentiality, device processing/capacity limits, and privacy constraints [77].

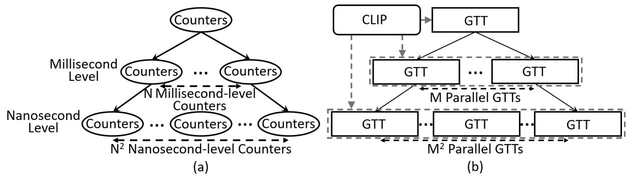


Figure 1: CLTM is a tree of Granular Traffic Transformers (GTTs).

Meanwhile, counters are readily available on almost all network devices and can be collected by network management systems (NMSs) [26, 43] (§2.2). On any network interface, we can easily find counters that accumulate byte-count or packet-count statistics. Counters are usually deeply embedded in the chip registers of the networking hardware; thus, they are high-performance, accurate, and reliable. Networking engineers frequently use them to monitor network behaviors or diagnose failures. But the problem with counters is that they are coarse–grained — the frequency at which NMSs collect them is typically in tens of seconds or even minutes. Without involving deep modification using SDK of the networking chip, a typical operating system of a networking device can only report counters on the scale of seconds. This is far from enough for the collection of packet-level fine-grained traces.

Given the abundance of counter statistics in NMSs and the scarcity of fine-grained traffic traces, we seek to answer a natural research question: *Can we restore fine-grained traffic traces from coarse-grained counters?* Although there have been continuous efforts in the synthesis of packet traces [28, 42, 50, 51, 71, 74, 77], none of them can recreate packet-level traffic from coarse-grained counter statistics.

Recent advances in generative artificial intelligence give us hope: for computer vision, image super-resolution (ISR) is a similar and well-studied task [35, 49]. ISR is the task of recreating high-resolution images (*e.g.* 1080p) from low-resolution ones (*e.g.* 360p), and it has many existing algorithms and models that achieve high-quality results [78, 79].

However, our experiments show that ISR algorithms and models cannot be applied directly to network traffic super-resolution (TSR) (§2.3). We summarize the unique challenges of TSR:

C1 **Input Format**: ISR and TSR work on different domains of data. A TSR system should take in a time-series (counters), and output a time-series (packets), while an ISR system deals with images (grids of pixels).

C2 **Extreme Upscaling Ratio**: The required upscaling ratio of TSR is many orders of magnitude larger than that of ISR. Packet-level traces have timestamps at resolutions of nanoseconds. To recreate a packet trace from a per-second counter trace, the upscaling ratio is 10^9. In contrast, the maximum ISR upscaling ratio of the popular Stable Diffusion model is only 16 [21].

C3 **Generality**: Network traffic in different network environments, such as the Internet, data centers, and access networks, exhibits diverse characteristics. For TSR, even when the coarse-grained traffic statistics remain consistent, the features of the corresponding fine-grained traffic should be different.

In addition, we define three more design requirements that serve to broaden our scope of application.

R1 **Multi-scale Synthesis**: Some downstream tasks do not require nanosecond-level timestamps, and a coarser-grained trace would suffice. For example, traffic engineering systems do not need traces with granularity of nanoseconds but will benefit if we can upscale a trace's resolution from 1 hour to 1 minute. Thus, we intend to generate traces with different upscaling ratios.

R2 **Understanding Counter Rules**: Due to the finite number of registers in network devices, operators often use user-defined rules (*e.g.* ACL) to specify which flows to track. Understanding the semantic meaning of these rules can guide the TSR to generate traces adhering to the rules. The rules can be accessed by operators from their configuration databases or network devices [76], and shared with researchers anonymously [72].

R3 **Real-time Synthesis**: Recreating high-resolution traffic from counters in real time can enable various downstream tasks such as network telemetry, failure diagnosis, and network digital twins [25].

Clearly, tackling these challenges and requirements simultaneously is ambitious, and an added difficulty is that we also need to design a new base model architecture for TSR, given that the existing models for ISR are unsatisfactory. We draw inspiration from the basic structure of the problem: A coarse-grained trace is the summary of many fine-grained traces. For example, the trace of a second-level counter is the *summary* of ten 100ms-level counters. This summarizing relationship forms a tree structure (Fig. 1), and we design the base model for TSR in the same way, which we name the Composable Large Traffic Model (CLTM).

The core of CLTM is a tree of Granular Traffic Transformers (GTTs). Each layer of the tree consists of the same GTT modules only look at the traffic characteristics at this particular granularity. Concentrating on a particular scaling phase (e.g. 1s to 100ms) better captures the characteristics of traffic at this granularity. For instance, with the same $10\times$ upscaling factor, ten 100ms-level counters generated from a second-level counter may have relatively average values, whereas the subsequent 10ms-level counters may show a more uneven distribution.

With CLTM, we design and implement ZoomSynth, the first network traffic synthesis system that supports TSR. ZoomSynth effectively tackles the aforementioned challenges and meets the specified requirements through a strategic combination of innovative ideas in addition to CLTM. Firstly, the tree of GTTs in CLTM addresses **C1**, **C2**, and **R1**, simultaneously. For **C3**, to adapt to new synthesis scenarios, we incorporate the Low-Rank Adaptation Model (LoRA) [40]. This adaptation model borrows the concept from domain adaptation in large language models (LLMs). Furthermore, we develop a rule-following model based on Contrastive Language-Image Pre-training (CLIP) [59], linking counter rules with the generation of traffic traces, to understand counter rules (**R2**). Finally, to achieve real-time generation (**R3**), we introduce pipeline-parallelism to accelerate CLTM's inference.

In summary, we make the following contributions.

- We illustrate the practical need for TSR and the availability of counters, underscoring the importance of resolving fine-grained traces from coarse-grained counters (§2).
- We design and implement ZoomSynth[1], the first TSR system that facilitates multi-scale TSR based on counters, with counter rules available as optional input (§3, §6).
- We propose CLTM, a novel base model architecture for TSR, which breaks down the TSR process into multiple phases, each employing a GTT tailored to the specific upscaling stage (§4).
- We evaluate ZoomSynth with extensive testbed experiments (§7) and confirm it meets the design goals. Notably, across all distributional metrics and traces [6, 7, 9, 22, 31, 55, 57, 63], ZoomSynth achieves packet trace synthesis with up to 25.8% JSD and 20.1% EMD improvements compared to the state-of-the-art schemes [46, 77]. ZoomSynth can also achieve real-time generation with over one billion packets per second with pipelined inference.

This work raises no ethical concerns. All traces used in this paper contain no personally identifiable information (PII).

2 Background and Motivations

In this section, we first introduce the applications of TSR (§2.1). Then, we demonstrate the wide availability of counters (§2.2) and show that existing approaches cannot be applied to TSR (§2.3). Finally, we motivate the design choices of ZoomSynth for TSR (§2.4).

[1]https://github.com/wxzisk/ZoomSynth_NSDI2025

2.1 Applications of TSR

We believe that any application that requires realistic fine-grained traffic traces can benefit from TSR, because TSR can turn counter traces into fine-grained packet traces, and counter traces are much easier to collect. Previous works [44, 50, 71, 74, 77] have explored a range of applications, including telemetry algorithms, the evaluation of machine learning models [77], and service recognition [46].

Accurate and real-time TSR can also enable new applications. One example is failure diagnosis using counters. Previously, counters could only offer operators a low-resolution view of network state. With TSR, operators can *expand* the counter trace into a high-likelihood packet trace, which can help understand and diagnose network events. TSR is also a potential enabling technology for network digital twins (NDTs) [25]. A major roadblock for NDT is the large amount of data that needs to be transferred from the physical network to the NDT system to realistically recreate the traffic in the digital twin, especially when physical networks today have interfaces $\geq$100Gbps. Using TSR, NDT systems can collect only the coarse-grained data from the physical network, and restore high-resolution traffic.

2.2 Availability of Counters

Counters are almost available on all network devices and network operating systems (NOS) and software. We conclude this by surveying the device configuration manuals of Cisco, one of the mainstream vendors, and widely used open-source NOS and software, such as software for open networking in the cloud (SONiC) [20], Linux kernel, and data plane development kit (DPDK) [10].

In the survey, we find 41 out of 42 categories of router products of Cisco listed at its official site [5] exactly declare supporting packet counters in their configuration manuals. Only the virtual router emulator states that some hardware counters are not supported by it [8]. We summarize the router products, configuration commands to display counters, and the corresponding online configuration manuals, in Table 6 in appendix §.5. Also, the vendor NOS, such as Cisco IOS XR [1, 2] and Cisco IOS XE [3, 4], that these routers run support the standard management protocols, such as SNMP [43] and NetFlow [26], which is commonly used to collect counters by the network management systems.

Besides, existing open-source NOS and software [10, 12, 20] support multiple types of counters as well. Taking SONiC as an example, its switch abstraction interface (SAI) defines eight types of counters [17] in its router interface API [19], which include the ingress or egress byte counters, ingress or egress packet counters, byte or packet counters for packets having errors on router ingress, and byte or packet counters for packets having errors on router egress. And Linux kernel implements 25 types of counters including packet and byte counters [12], while DPDK supports packet and byte counters in its packet framework library [15, 18].

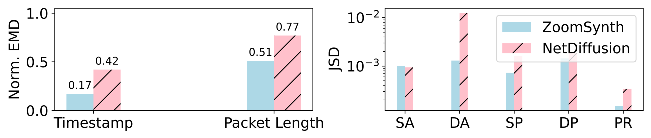


Figure 2: NetDiffusion's performance for TSR to achieve 10^9 end-to-end upscaling.

2.3 Applicability of ISR to TSR

Recent advances in ISR [35, 49] address a similar research question to TSR, aiming to restore high-resolution images (*e.g.*, 1080p) from low-resolution images (*e.g.*, 360p). Many ISR algorithms and models have achieved high-quality results [78, 79]. This prompts us to investigate whether existing ISR algorithms and models can be applied directly to TSR.

We notice a recent pioneering work that applies ISR to traffic synthesis [45]. NetDiffusion utilizes Stable Diffusion 1.5 [65] as the base model, which is recognized as one of the state-of-the-art ISR models. We deploy the open-source implementation of NetDiffusion [13] and evaluate its performance for TSR, *i.e.*, synthesizing packet-level traces from counters. To ensure consistency, we download the model and follow the same fine-tuning process as NetDiffusion, while employing the same datasets as those used by ZoomSynth. Fig. 2 illustrates the experimental results in terms of Jensen-Shannon Divergence (JSD) [56] and Earth Mover's Distance (EMD) [66], and we also show the performance of ZoomSynth. For these two metrics, lower is better, and we observe that NetDiffusion has worse performance than ZoomSynth.

We dive into the implementation of NetDiffusion and identify two primary reasons. First, there is a mismatch between input formats (or data representation) required by the Stable Diffusion model and the inherent nature of traffic data, which is naturally represented as time series. NetDiffusion needs to fold the time series into matrices, which may induce the ISR model to capture unwanted or non-existent relationship between data points. Second, NetDiffusion directly uses the pre-trained model based on image datasets as the basic model and only fine-tunes it for traffic synthesis. Thus, the direct application of an ISR model for TSR, without sufficient customization to the characteristics of traffic data, results in unsatisfactory performance.

2.4 ZoomSynth Design Motivations

We first summarize the challenges and requirements in §1 into five design goals, and then we motivate the design choices of ZoomSynth to achieve these goals.

Design goals. Based on §1, we summarize the goals for ZoomSynth.

G1 **Counter-to-packet Synthesis**: This goal ensures practical applicability, given that many downstream tasks necessitate packet-level traces and coarse-grained counters are widely available.

G2 **Multi-scale Synthesis**: In addition to packet traces (nanosecond trace), ZoomSynth should generate traffic traces with varying upscaling ratios.

G3 **Counter-rule-based Synthesis**: ZoomSynth should comprehend the semantic meaning of these rules, such as ACL policies, and generate traces that adhere to them.

G4 **Generality**: Traffic traces in different network environments have different characteristics. ZoomSynth should adapt to the features of these varying environments and generate traces with user-defined granularity.

G5 **Real-time Synthesis**: Real-time synthesis enables ZoomSynth to be used in real-time tasks. We define real-time synthesis as follows: for a counter collected every n seconds ($n \geq 1$), ZoomSynth should generate the corresponding packet-level trace within n seconds.

To achieve the goals, our five design choices for ZoomSynth are as follows:

1. GTT-based TSR: With the unsatisfactory performance of ISR models for TSR, we seek a model which is more suitable to handle time-series data. We choose Transformer [70] for ZoomSynth (§4.1), because Transformer can capture long-range dependencies in time-series sequences, and is well-suited for modeling time-series traffic traces where temporal dependencies span over extended periods [67, 68].

We do not use a single Transformer model for TSR: each Transformer model is responsible for only one upscaling granularity (*e.g.*, 1s to 100ms), and it captures the characteristics of traffic at this particular granularity. We name this type of models the Granular Traffic Transformer (GTT). We use GTT $^{k}_{\delta}$ to denote a GTT with upscaling factor k and generation resolution δ. For example, a GTT $^{10}_{100ms}$ can transform a trace with 1s resolution to one with 100ms resolution.

2. Tree-structured CLTM Model: A coarse-grained counter summarizes the underlying fine-grained counters, forming a classic tree structure at different granularities, as illustrated in Fig. 1(a). With this observation, we then compose GTTs at different granularities into a tree, which we name the Composable Large Traffic Model (CLTM).

CLTM can achieve high upscaling ratios (**G1**). CLTM contains a tree of GTTs, illustrated in Fig. 1(b). This hierarchical structure allows the upscaling ratios to be multiplied across GTTs in distinct layers of CLTM. Consequently, CLTM can achieve exponential increases in upscaling ratios by incorporating additional layers of GTTs. This enables CLTM to attain high upscaling ratios.

CLTM supports multi-scale traffic synthesis (**G2**). Within the CLTM tree, each layer of GTTs is dedicated to a specific granularity. Each layer can generate traffic traces with the targeted granularity, enabling multi-scale traffic synthesis.

3. Counter Rule Understanding with CLIP: **G3** requires us to use counter rules (*e.g.*, text strings of ACL policies) to guide the generation of traffic traces, which is a multimodality task of two domains: text and time-series. We learn from mainstream multimodal models [32, 52], and use CLIP [59] to develop a conversion model to link counter rules with traffic traces. This ensures that synthetic traffic aligns with specified rules in every layer of the CLTM (§4.2).

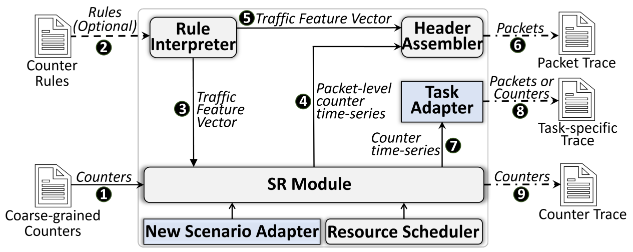


Figure 3: ZoomSynth end-to-end workflow (New Scenario Adaptor and Task Adaptor only run in the training phase). Note that counter rules are not mandatory input but optional.

4. Generality with a LoRA-based Approach: To generalize ZoomSynth, we seek a method that allows users to use their own data to *fine-tune* CLTM, since it is impossible to collect data in all scenarios. Thus, for **G4**, we leverage a LoRA-based approach, borrowing from the LLM literature [40], to facilitate lightweight fine-tuning of CLTM for adapting to new scenarios (appendix §.1.5).

5. Pipeline-parallel Inference: To achieve **G5**, we introduce pipeline paralellism and execute GTTs in different layers of CLTM concurrently. We dynamically allocate computational resources to different layers by scheduling GTTs automatically to ensure efficient inference. We also develop a prefetching technique to further improve the performance (§4.3).

3 ZoomSynth Overview

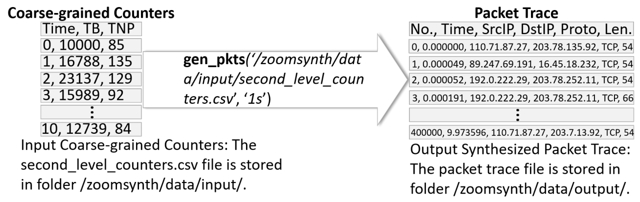
We present the overview of ZoomSynth in Fig. 3. ZoomSynth is a TSR system designed for fine-grained traffic synthesis using counter traces and counter rules.

3.1 Architecture and Components

ZoomSynth has six key modules, shown in Fig. 3, and we describe them as follows:

- **SR Module**: The SR Module runs the CLTM model. It takes as input a coarse-grained counter time-series. Through the application of the SR module, each counter within the time series undergoes an upscaling process, resulting in a more fine-grained counter time series. Users' required upscaling ratio can be achieved by composing the GTT models to the target number of layers. We set the upscaling factor of each layer to a fixed value of 10 and justify this in appendix §.1.2.
- **Rule Interpreter**: The Rule Interpreter runs a CLIP-based rule-following model. It takes counter rules as input and outputs a feature vector representing the traffic traces complying with the rules. As shown in Fig. 3, the output vector of the Rule Interpreter is fed into each layer of CLTM to help align the time-series features of counters, while in

APIs	Function
`gen_pkts(counter, orig_res)`	Packet trace synthesis
`gen_counters(counter, orig_res, upscaling_ratio)`	Multi-scale synthesis
`gen_for_task(counter, orig_res, upscaling_ratio, task)`	Downstream Task



Coarse-grained Counters

```
Time, TB, TNP
0, 10000, 85
1, 16788, 135
2, 23137, 129
3, 15989, 92
⋮
10, 12739, 84
```

Input Coarse-grained Counters: The second_level_counters.csv file is stored in folder /zoomsynth/data/input/.

gen_pkts(*'/zoomsynth/data/input/second_level_counters.csv', '1s'*)

Packet Trace

```
No., Time, SrcIP, DstIP, Proto, Len.
0, 0.000000, 110.71.87.27, 203.78.135.92, TCP, 54
1, 0.000049, 89.247.69.191, 16.45.18.232, TCP, 54
2, 0.000052, 192.0.222.29, 203.78.252.11, TCP, 54
3, 0.000191, 192.0.222.29, 203.78.252.11, TCP, 66
⋮
400000, 9.973596, 110.71.87.27, 203.7.13.92, TCP, 54
```

Output Synthesized Packet Trace: The packet trace file is stored in folder /zoomsynth/data/output/.

Figure 4: APIs of ZOOMSYNTH's applications and an example for packet trace synthesis from second-level counters using the API `gen_pkts(·)` of ZOOMSYNTH (TB: Total Bytes, TNP: Total Number of Packets).

the header generation process, it is used by the Header Assembler to constrain the scope of the packet headers.

- **Header Assembler**: The Header Assembler runs a GPT-2 based header generation model. As shown in Fig. 3, it takes the outputs from both CLTM and Rule Interpreter as input and generates packet headers to synthesize packet traces.
- **Resource Scheduler**: This module allocates resources to run the above modules. We use pipeline parallelism (a type of model parallelism) to accelerate the inference of CLTM. For the other models, running Rule Interpreter, Header Assembler and Task Adapter is trivial as their model is small and do not need model parallelism, while New Scenario Adaptor only runs offline and does not affect the inference performance of real-time traffic synthesis.
- **Task Adaptor**: This module runs task-specific model with domain knowledge. It can take fine-grained traces with various granularity as input, and leverages task-specific model to align the traces to the domain of the downstream task. Taking an ML-based anomaly detection task as an example, the Task Adaptor uses the ML-based anomaly detection model trained with real traffic data to align the type labels of the fine-grained traces.
- **New Scenario Adaptor**: This module aims to make ZOOMSYNTH adapt to new scenarios efficiently. It leverages a LoRA-based approach to update the CLTM with a small amount of traffic traces of the new scenarios, while purely retraining the models of Header Assembler and Task Adapter as they are small and quick to train. We explore the efficiency of this module in appendix §.4.1.

3.2 Workflow

User of ZOOMSYNTH can programmatically determine their desired resolution by configuring the CLTM. In this way, ZOOMSYNTH supports arbitrary upscaling ratios and can also act as *base model* for many downstream tasks. For each application, ZOOMSYNTH needs to adapt its relative components to synthesize required traces from coarse-grained counters. ZOOMSYNTH provides APIs summarized in Fig. 4 to run the applications. We summarize the workflow of ZOOMSYNTH in Fig. 3 and describe its applications as follows.

Packet Trace Synthesis: The finest-grained traces that ZOOMSYNTH can generate are packet trace synthesis with nanosecond resolution. ZOOMSYNTH utilizes Rule Interpreter, SR Module, and Header Assembler to compose the assembly line for packet trace synthesis. It takes the ❶ counters or both ❶ counters and ❷ counter rules as input, and processes them with ❸ ❺ the Rule Interpreter, ❹ SR Module, and ❻ Header Assembler in succession and outputs packet traces. To run these modules, users can use the `gen_pkts(·)` API, where the parameter `counter` is the path to the file of counters and rules and the parameter `orig_res` is the granularity of the input counters.

Taking Fig. 4 as an example to illustrate packet trace synthesis based on second-level counters by using `gen_pkts(·)` API of ZOOMSYNTH. When invoking the API, users need to specify the input file of the coarse-grained counters with its granularity, ZOOMSYNTH outputs the packet trace after running the API.

Multi-scale Fine-grained Trace Synthesis: Fine-grained counter traces are counter time-series on a finer scale than the input counter time-series. They follow the same counter rules and do not contain packet header information. Therefore, ZOOMSYNTH only takes the ❶ counters or both ❶ counters and ❷ counter rules as inputs, and adopts ❸ the Rule Interpreter and ❾ the SR Module to synthesize fine-grained counter traces. Users can adopt the API `gen_counters(·)` to run the application, where the parameter `upscaling_ratio` is the target end-to-end upscaling ratio.

Downstream Task: Specific downstream network tasks generally require task-specific fine-grained traffic traces. Before synthetic traces are used for a specific task, ZOOMSYNTH leverages Task Adapter to align them to task-specific characteristics with domain knowledge. Therefore, ZOOMSYNTH takes the ❶ counters or both ❶ counters and ❷ counter rules as inputs, and adopts ❸ Rule Interpreter, ❼ SR Module, and ❽ Task Adapter to synthesize counter or packet traces needed by downstream tasks. ZOOMSYNTH provides the API `gen_for_task(·)` for users to specify their input file of coarse-grained counters, counters' original resolution, target end-to-end upscaling ratio, and the downstream task model. In §5.2, we implement three downstream tasks based on ZOOMSYNTH, and evaluate their performance in §7.2.

4 Composable Large Traffic Model

CLTM is composed of GTT models and CLIP-based rule-following model and is the key for enabling multi-scale TSR. In CLTM, GTT is organized as a tree structure, by combining it with the rule-following model, CLTM empowers rule-based synthesis. CLTM supports real-time synthesis with pipelined inference. In the following, we illustrate its design in detail.

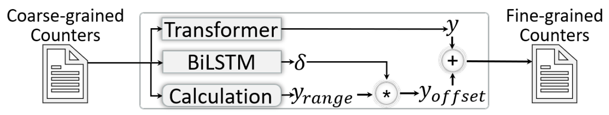


Figure 5: The structure of GTT.

4.1 GTT Model

GTT model consists of two key modules: a Transformer model [70] and a bidirectional LSTM (BiLSTM) [38]. GTT uses Transformer to learn temporal characteristics and leverages BiLSTM to deal with extreme values of counter time series. For each stage of CLTM, a GTT $\frac{k}{\delta}$ make $k\times$ upscaling for counters at a δ resolution. We describe more details about the training data preparation for GTT and the mechanism to determine the upscaling factor k in appendix §.1.1 and appendix §.1.2, respectively.

Handling Extreme Values: The counter time-series displays many extreme values, which are caused by many hidden factors, such as network stack, network functions, and scheduling policies. We observe that a Transformer cannot handle extreme values efficiently, since it tends to evenly distribute packets and byte counts, making upscaled counters tend to favor the means. To address this issue, we use a BiLSTM model to learn the distribution of extreme values, which is borrowed from extreme value prediction methods [33]. GTT integrates Transformer and BiLSTM, as shown in Fig. 5, and uses BiLSTM to learn the distribution frequency of extreme values and their deviations from the means. Unlike Transformer, which directly uses coarse-grained and the corresponding fine-grained counter time-series as training data, we leverage the same input time-series but uses the deviations from the mean of the fine-grained counter time-series as output to BiLSTM in the training phase.

As shown in Fig. 5, in the inference phase, when GTT receives the counter time-series, it feeds the input into both the Transformer and BiLSTM, and calculates the range of the counter time-series. GTT takes the offset ratios (δ) output by BiLSTM to multiply by the calculated range (y_{range}) to obtain the offsets (y_{offset}), and then add them with the output (y) of Transformer to produce the final output.

In order to make GTT models learn counter time-series distributions, we adopt both Mean Square Error (MSE) and EMD in the loss function, while guiding it to satisfy the counter equality constraints described as follows.

Counter Equality Constraints: Clearly, the counters obtained from the fine-grained counter time-series outputted by a GTT should align with the input coarse-grained counters, which relation we call counter equality constraints, denoted as $E(C_{s-1},\hat{C}_s)$.

$$E(C_{s-1},\hat{C}_s)=0,$$
$$\text{where } E(C_{s-1},\hat{C}_s)=C_{s-1}-F(\hat{C}_s).$$

Here, C_{s-1} represents the realistic input counter time-series of upscaling stage s, and $\hat{C}_s$ is the corresponding counter time-series outputted by the GTT model based on C_{s-1}. F is the counter filtering function.

Loss Function of GTT: We integrate MSE and EMD into our loss function, since MSE can make the means of GTT's output closely align with the target values of the ground truth, while EMD facilitates learning the intrinsic structure of the target distribution. By minimizing both MSE and EMD, the loss function guides GTTs to capture the counter time-series characteristics at each upscaling stage. Thus, the combined loss function can be expressed as:

$$L_{combine}=MSE(\hat{C}_s,C_s)+\lambda EMD(\hat{C}_s,C_s),$$

where $\hat{C}_s$ is the output of a GTT and C_s is the corresponding ground truth at the upscaling stage s, and λ is a hyperparameter to balance MSE and EMD metrics.

Besides finding model parameters that minimize $L_{combine}$ across the training data, GTTs also need to adhere to the counter equality constraints. Formally, we translate it to solving an optimization problem:

$$\min(L_{comibine}),$$
$$\text{s.t. } E(C_{s-1},\hat{C}_s)=0.$$

Inspired by Zoom2Net's design for knowledge augmented loss [37], in order to make GTTs learn and respect the constraints during the training phase, we employ the augmented Lagrangian method [34] to establish an augmented Lagrangian loss function:

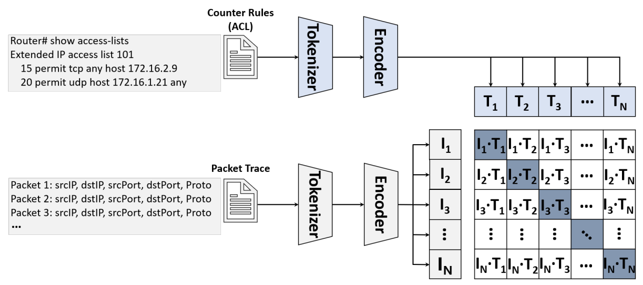
$$L=L_{combine}+\sigma E(C_{s-1},\hat{C}_s),$$

where σ is a hyperparameter to balance $L_{combine}$ and $E(C_{s-1},\hat{C}_s)$. Training GTTs with the loss function L enables them to capture the consistency between the input and output counter time-series, as enforced by the counter equality constraints. Different from Zoom2Net [37], we do not need to consider inequality constraints, since counters within the same time-series do not have such constraints.

4.2 Composing GTTs and Rule-following Model into CLTM

ZoomSynth automatically composes GTTs in CLTM to the target number of stages based on upscaling ratio parameter in its API summarized in Fig. 4. For instance, when users specify the end-to-end upscaling ratio to 10^9 for a second-level counter time-series, ZoomSynth automatically composes GTT $\frac{10}{100ms}$, GTT $\frac{10}{10ms}$, ..., and GTT $\frac{10}{1ns}$ into CLTM in a hierarchical manner. The number of GTTs increases by a factor of k in each subsequent layer, which guarantees that each GTT processes the same size of counter time-series.

Rule-following Model: The counters are the summarization of traffic traces processed following specific counter rules. Properly understanding the counter rules and applying them to the TSR process benefits the synthesis process when restoring fine-grained trace from counters. This is similar to text-based image synthesis, which utilizes CLIP [59] to align the embeddings of images and their corresponding textural descriptions, thereby establishing an association between them. Inspired by


Counter Rules (ACL)
Tokenizer
Encoder
T_1 T_2 T_3 ··· T_N
Packet Trace
Tokenizer
Encoder
I_1 I_2 I_3 ⋮ I_N
$I_1{\cdot}T_1$ $I_1{\cdot}T_2$ $I_1{\cdot}T_3$ ··· $I_1{\cdot}T_N$
$I_2{\cdot}T_1$ $I_2{\cdot}T_2$ $I_2{\cdot}T_3$ ··· $I_2{\cdot}T_N$
$I_3{\cdot}T_1$ $I_3{\cdot}T_2$ $I_3{\cdot}T_3$ ··· $I_3{\cdot}T_N$
$I_N{\cdot}T_1$ $I_N{\cdot}T_2$ $I_N{\cdot}T_3$ ··· $I_N{\cdot}T_N$

```
Router# show access-lists
Extended IP access list 101
   15 permit tcp any host 172.16.2.9
   20 permit udp host 172.16.1.21 any
```

```
Packet 1: srcIP, dstIP, srcPort, dstPort, Proto
Packet 2: srcIP, dstIP, srcPort, dstPort, Proto
Packet 3: srcIP, dstIP, srcPort, dstPort, Proto
...
```

Figure 6: The pre-training process of rule-following model.

this, we employ CLIP to construct a conversion model, called the rule-following model, that bridges counter rules and traffic traces. The rule-following model is fine-tuned using counter rules and traffic traces to understand their relationships in a shared latent space.

Fig. 6 shows the pre-training process of rule-following model. The two Transformer-based encoders with 63M parameters encode counter rules and packet trace to the vectors $T=(T_1,T_2,...,T_N)$ and $I=(I_1,I_2,...,I_N)$, respectively, and map them to the same latent space. Here, we set N to 256. In the training phase, we minimize the cosine similarity of T and I and obtain a similarity matrix of counter rules and packet trace. In the inference phase, we encode the counter rules with the encoder to its vector representation T and convert T to another vector representation using the similarity matrix.

In order to prepare the training data for the rule-following model, we process the original pcap data with specific counter rules and generate packet traces which adhere to them. After finishing the training, we compose the model into CLTM as shown in Fig. 1. We compose GTTs and the pre-trained rule-following model into CLTM for training and inference to achieve multi-scale synthesis based on both counters and counter rules.

4.3 Pipelined Inference of CLTM

Fig. 7 showcases the task scheduling of ZoomSynth on 4 GPUs for pipelined inference of CLTM, which is for packet trace synthesis from second-level counters. A task is defined as a GTT model for processing a fixed size of counter time-series. We use a block to represent a task in Fig. 7, and the number within the block indicates the specific GTT model a task runs. For instance, task belongs to stage 1 of CLTM run GTT $^{10}_{100\text{ms}}$. The number of tasks increases by multiplying by the upscaling factor k along with the increase of CLTM's stages. Here, we assume k=10.

ZoomSynth follows two simple rules to schedule tasks on GPUs. (1) Prioritize tasks running GTT models with coarse-grained generation resolutions, because finer-grained tasks depend on them. (2) Prioritize tasks whose GTTs have input data affinity with preceding tasks running on the same GPU. As shown in Fig. 7, ZoomSynth can fully utilize all GPUs after the first stage.

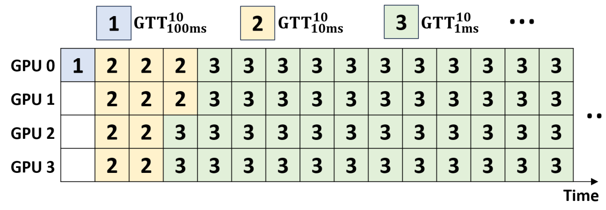

1 $\text{GTT}^{10}_{100\text{ms}}$
2 $\text{GTT}^{10}_{10\text{ms}}$
3 $\text{GTT}^{10}_{1\text{ms}}$
···
GPU 0 1 2 2 2 3 3 3 3 3 3 3 3 3 3 3
GPU 1 2 2 2 3 3 3 3 3 3 3 3 3 3 3
GPU 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
GPU 3 2 2 3 3 3 3 3 3 3 3 3 3 3 3
···
Time

Figure 7: The task scheduling of ZoomSynth on 4 GPUs for pipelined inference of CLTM. Each block represents a task, which is a GTT model for processing a fixed size of counter time-series. The number within each block represents the stage of CLTM the task belongs to. The ratio for the number of tasks between neighbouring CLTM stages is 1:10 assuming upscaling factor k=10.

5 Enabling Downstream Applications

CLTM enables downstream applications of ZoomSynth, and we describe them in this section.

5.1 Header Generation

The final step of packet traces generation is to provide headers for the packets. Similarly to the existing work [62, 77], we use an embedding model to encode the header fields of the packet. Unlike previous work, the header generation model only takes care of the synthesis of five tuples (*i.e.*, source IP, destination IP, source port, destination port and protocol), since CLTM has provided the other information of packets, such as packet timestamp and size.

Packet Header Encoding: We use an embedding model similar to IP2Vec [62] to encode the categorical fields of the packet header, such as IP address, port, and protocol. The embedding is trained on packet traces collected from public datasets. Considering that the IP, port, and protocol fields in our datasets are finite, we utilize an embedding layer with a vocabulary size of 50000 and embedding dimension of 128 to serve as our header encoder. Users can retrain it to cover any additional fields with their new datasets.

Header Generation Model: Building upon the success of previous studies [30, 58] that utilize GPT-2 [60] for packet header generation, we adopt this model to reconstruct packet headers using the finest-grained counter time series from CLTM. We retrain the model using public packet traces. We detail the model training process in appendix §.2.

During the inference phase, the header generation model populates the header fields of packets using the information extracted from the raw training packet traces, alongside the timestamps and sizes derived from the synthesized finest-grained counter time-series. This ensures that the generated packet headers adhere to the traffic semantics and preserve the traffic distribution characteristics. However, this approach has limitations with respect to the generalization of packet headers, which we discuss in §9.

5.2 Task Adaptor

The Task Adaptor fine-tunes CLTM for downstream tasks, allowing the end-to-end upscaling model to be trained with a loss function that is more conducive to downstream tasks. This is similar to *fine-tuning* in the LLM literature [41]. As for most downstream tasks, packet header traces are needed, since the discrete nature of packet header information guarantees that the Task Adaptor does not entail predicting extreme values when synthesizing header fields. As a result, we do not need BiLSTM here. We use a Transformer model for Task Adaptor.

The effectiveness of synthetic traffic used for downstream tasks is the key criterion for evaluating the Task Adaptor. For this, we implement three classic downstream tasks like NetShare [77], including anomaly detection, sketch-based telemetry, and service recognition, and we use the same machine learning (ML) model as NetShare [77].

Anomaly Detection: Existing ML models for this task require packet traces with fields that include packet arrival time, packet size, and five tuples. Based on ZoomSynth, we use five models [77] for anomaly detection: Decision Tree (DT), Logistic Regression (LR), Random Forest (RF), Gradient Boosting (GB), and Multi-layer Perceptron (MLP).

Sketch-based Telemetry: We implement a Count-Min Sketch (CMS) [53] and use it to estimate the proportion of heavy-hitters within the traffic. Sketch-based telemetry commonly uses traffic synthesis as a precursor task, producing a large volume of traffic with consistent characteristics.

Service Recognition: The service recognition task is a more complex ML-based task, different from the above two categories of tasks. It relies on more deep features of traffic to recognize services, instead of looking at IP addresses, port numbers, and protocol types. We adopt the same five models for anomaly detection for this task, and utilize data from CIDDS to train them for service recognition. The services are categorized into three types: email, file copying, and others.

6 Implementation

We implement a prototype of ZoomSynth in Python 3.10. We use PyTorch [16] library to implement all the models in $\sim$950 lines of code (LOC), except for the header generation model, which we implement using the codes from Hugging Face. And for parsing datasets into packets and multi-scale counters, we implement scripts in $\sim$230 LOC. Table 5 in appendix §.3.1 lists the hyper-parameter setups of ZoomSynth for each model. Specifically, besides the APIs summarized in Fig. 4, we also implement 6 APIs to support operations of CLTM. Based on these APIs, users can adjust the number of GTTs in each stage of CLTM by considering their hardware and TSR tasks. For the three downstream tasks, we also utilize the PyTorch library to implement the models for anomaly detection and service recognition in $\sim$420 LOC and implement CMS for sketch-based telemetry in $\sim$170 LOC. We implement the Resource Scheduler in $\sim$130 LOC, which adopts `nvidia-smi` (provided by the GPU driver) to monitor GPU resources and schedule tasks to GPUs.

Datasets: We aggregate seven open-source datasets available online, namely TON (IoT traffic data) [57], CA (data of a cyber defense competition) [6], CIDDS (enterprise traffic) [7], DC (data center traffic) [9], UGR16 (traffic data from Spanish ISP) [55], CAIDA (anonymized trace on backbone network) [22], and MAWI (packet traces from WIDE backbone) [31]. The resulting dataset have a combined size of 16GB, and MAWI and CAIDA predominantly constitute this hybrid dataset, with each accounting for 45% of the data. We obtain counter traces of varying resolutions from this dataset, and the counter traces has a total size of >1.7TB.

Model Size: With k=10, input resolution 1s, and output resolution 1ns (packet-level), the required CLTM has 1.8 billion parameters. We call this model CLTM-1.8B, and release a pre-trained version[1] using the above datasets, accompanied with the source code for its training. We use this model in the evaluations, unless otherwise specified.

7 Evaluation

In this section, we seek to understand: 1) With only counter traces, can ZoomSynth achieve a similar performance in traffic generation as existing traffic synthesizers that take packet-level trace as input? 2) Can the synthesized ZoomSynth traces satisfy the requirements of downstream tasks? 3) Can ZoomSynth achieve real-time synthesis? 4) How does the counter based on the counting bloom filter (CBF) affect TSR? 5) How does each model in ZoomSynth contribute to its performance? We answer the first four questions positively. We summarize the evaluation results below.

Summary of Results:

- For packet-level trace synthesis, ZoomSynth, using only counters, achieves a similar performance as NetShare that takes packet-level trace as input. ZoomSynth achieves up to 25.8% and 20.1% improvements in terms of JSD and EMD, respectively. And across all four datasets, ZoomSynth achieves only 30.6% reduction in JSD.
- ZoomSynth relaxes the requirements for data collection for downstream tasks. With only counter traces, ZoomSynth with fine-tuned CLTM outperforms NetShare by 27.5% and 9.8% in anomaly detection and service recognition tasks, respectively.
- For a CLTM-1.8B on a server with 8$\times$ A100 GPUs, for a per-second counter trace, ZoomSynth can generate at most 10^9 packets in 0.966 second. This exceeds the maximum forwarding capacity of modern core network devices [8, 23, 24], proving its real-time synthesis capacity.
- We observe that CBF-based counters have negative impact on TSR performance, and find that ZoomSynth still outperforms NetDiffusion and Zoom2Net, achieving im-

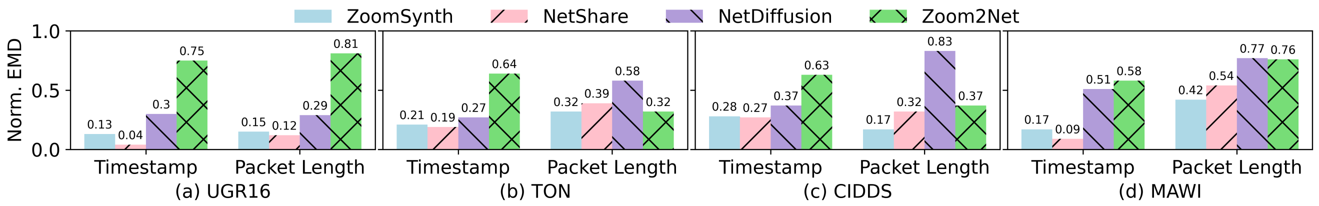


Figure 8: Earth Mover's Distance (EMD) between real and synthetic distributions on UGR16, TON, CIDDS, and MAWI, where the end-to-end upscaling ratio is 10^9.

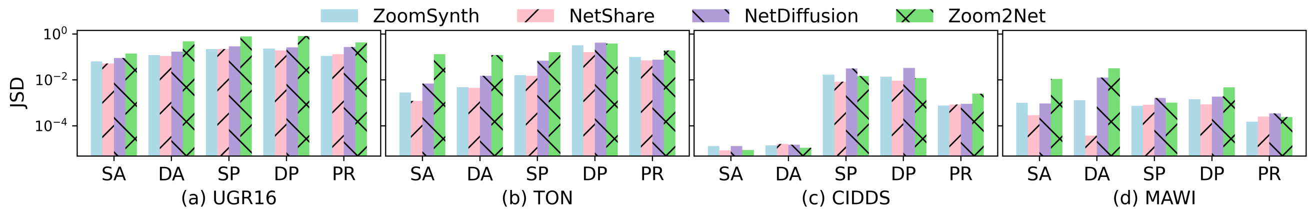


Figure 9: Jensen-Shannon Divergence (JSD) between real and synthetic distributions on UGR16, TON, CIDDS, and MAWI, where the end-to-end upscaling ratio is 10^9.

provements in EMD by 58.3% and 76.0%, and in JSD by 52.3% and 80.1%, respectively.

- We confirm the effectiveness of the header generation and rule-following models in ZoomSynth.

Testbed: We use a machine equipped with 8× NVIDIA A100 GPUs, 2× 64-core 2.90GHz Intel Xeon Platinum CPUs, and 2TB DDR4 RAM. The machine runs Ubuntu 22.04 LTS and PyTorch 2.1.0.

Comparison Targets: We compare ZoomSynth to three existing traffic synthesizers, each of which is the state-of-the-art work in their application scenarios. (1) *NetDiffusion* [45]: This scheme originally uses a stable diffusion model [65] to synthesize the traffic trace from text, such as *generating TCP traffic*. In the TSR scenario, we use it to synthesize traffic trace from counters. (2) *NetShare* [77]: This system must take packet traces as input, and uses the DoppelGANger model [50] with domain-specific insights to generate packet traces. It cannot take counter traces as input. (3) *Zoom2Net*: We extend Zoom2Net [37] for TSR. Following [37], this scheme uses a single Transformer model [70] to complete all tasks for TSR including up-scaling and packet header generation. In addition, we use the same loss function as ZoomSynth to incorporate the domain knowledge in Zoom2Net. Comparison of ZoomSynth with Zoom2Net can demonstrate the performance benefit of ZoomSynth's other design choices. The hyperparameters of Zoom2Net are the same as GTT.

Metrics: We evaluate the fidelity of synthetic traffic traces by computing distance metrics between real and synthetic distributions of packet header fields. Following the common practice in prior work [74, 77], we utilize Earth Mover's Distance (EMD) (also called Wasserstein-1 distance) for continuous fields, such as packet length and timestamp, and Jensen-Shannon Divergence (JSD) for discrete fields, such as source address (SA), destination address (DA), source port (SP), destination port (DP), and protocol (PR).

7.1 Counter-to-Packet Synthesis

Accuracy. To assess the generation quality of timestamp field and packet length field, we employ EMD to compare the generated packet traces with the actual ones, with smaller values indicating closer resemblance. Fig. 8 presents the EMD metrics across four datasets. With only second-level counters, ZoomSynth achieves performance comparable to NetShare (using packet-level traces as input), and significantly outperforms Zoom2Net and NetDiffusion. Specifically, ZoomSynth achieves EMD reductions of 69.5% and 48.4% in the four datasets on average compared to Zoom2Net and NetDiffusion, respectively. In TON, CIDDS, and MAWI datasets, our method surpasses NetShare in terms of packet length's EMD. We further look at the percentiles of deviations for the 1st and 99th synthetic packet lengths generated by ZoomSynth and NetShare in appendix §.4.2, and find that NetShare tries to fit the extreme values, making it perform well in generating packet lengths around the extreme values and also causing it to tend to overfit the variations of the packet length time-series, and thus leading to worse EMD.

For categorical fields such as IP address/port number/protocol in packet headers, we use JSD, a metric that assesses the distribution characteristics of discrete data within the whole set, and a smaller JSD is better. Similarly, we calculate the JSD between the synthetic data and the real data for each scheme, which is shown in Fig. 9. We can see ZoomSynth achieves 49.6% and 35.6% JSD reductions compared to Zoom2Net and NetDiffusion, respectively, and it is slightly inferior to the NetShare scheme, which uses packet-level traces as input.

Error Accumulation. To show how errors accumulate in the multi-level TSR process of ZoomSynth, we assess the results generated at multiple granularities. Counter-level results encompass three dimensions of information: time interval, byte count, and packet count. Since the time interval at the

Table 1: Performance comparison of ZoomSynth, NetShare, NetDiffusion, and Zoom2Net on three downstream tasks.

Downstream Task	Anomaly Detection					Sketch-based Telemetry			Service Recognition				
	MLP	DT	RF	GB	LR	TON_SA	TON_DP	CAIDA_SP	MLP	DT	RF	GB	LR
Real Traces	0.99	1.0	1.0	1.0	0.8	-	-	-	1.0	1.0	1.0	1.0	0.87
ZoomSynth	0.88	0.91	0.81	0.87	0.78	0.22	0.64	0.57	0.87	0.89	0.84	0.84	0.75
NetShare	0.58	0.78	0.64	0.77	0.60	0.14	0.62	0.53	0.83	0.79	0.85	0.78	0.60
NetDiffusion	0.43	0.49	0.43	0.51	0.36	0.29	0.78	0.73	0.60	0.59	0.63	0.55	0.49
Zoom2Net	0.31	0.42	0.32	0.35	0.32	0.41	0.94	0.82	0.58	0.54	0.61	0.49	0.43

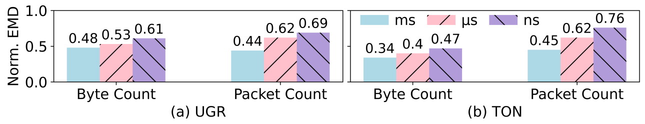


Figure 10: Error accumulation on counters of packets and bytes as the TSR processes.

counter level is fixed that represents the sampling period, we just calculate the EMD for the byte and packet counts. Due to how the EMD is calculated, there are magnitude differences in the EMD across different granularities. Therefore, when comparing the EMD values horizontally, it is necessary to perform normalization. Fig. 10 shows the normalized results. When generating microsecond counters from millisecond counters in the TON dataset, it is evident that even when the EMD values for millisecond data are divided by a thousand (reflecting the granularity difference), they are still significantly lower than the EMD for the microsecond-level results. A similar pattern is observed when nanosecond counters are generated from microsecond counters.

We conclude that, although errors accumulate in multi-level generation, the growth of accumulated error is slow, which has limited impact on overall accuracy, as we observe in Figs. 8 and 9.

7.2 Downstream Task Performance

We evaluate the performance of the three downstream tasks manufactured by ZoomSynth in the following.

Anomaly Detection: We train DT, LR, RF, GB, and MLP with both real and synthetic traces and compare their accuracy with real traces. Table 1 shows that models trained using ZoomSynth's synthetic traces achieve superior accuracy compared to NetShare, especially for DT model, its accuracy is 27.5% higher than that of NetShare, while they improve 93.2% and 149.2% on average accuracy compared to NetDiffusion and Zoom2Net, respectively.

Sketch-based Telemetry: We use E_{real} and E_{syn} to represent the number of errors in identifying heavy-hitters using CMS for real and synthetic data, respectively. If the features of the generated data are closer to those of the real data, these two values are also closer. Therefore, we use the relative error ratio $\frac{|E_{real}-E_{syn}|}{E_{real}}$ to evaluate the similarity between real data and synthetic data, and a lower value means better performance. We calculate the relative error of the heavy-hitters using TON and CAIDA datasets. Table 1 shows that the synthetic traces from ZoomSynth have a performance similar to that of NetShare, with only 23.6% higher average relative error, and reduce the average relative errors with 21.3% and 49.6%, respectively, compared to NetDiffusion and Zoom2Net.

Table 2: The inference time of ZoomSynth for achieving different end-to-end upscaling ratios from second-level counters. The upscaling factor k=10 and the results are obtained using TON dataset.

End-to-end Upscaling Ratio	10	10^2	10^3	10^5	10^9
Inference Time (Seconds)	0.855	0.938	0.951	0.965	0.966

Service Recognition: We compare the trained service recognition models using data from the real dataset and the synthetic ones of the compared schemes. Table 1 presents the results that the synthetic traces of ZoomSynth better represent the characteristics of the real ones. For instance, the models trained with the data from ZoomSynth achieve 9.8%, 46.9%, and 59.7% higher accuracy on average than the models trained with the synthetic data from NetShare, NetDiffusion, and Zoom2Net, respectively.

7.3 Real-time Synthesis

To assess whether ZoomSynth can empower real-time downstream tasks, such as NDT [25], we measure the inference time under various end-to-end upscaling ratios. And to avoid measuring errors that result from inference with insufficient counter time series data, we employ counters of 1-second duration to generate packets and test whether the inference time exceeds 1 second. For example, when evaluating with an upscaling factor of 10^6, we use 1000× ms-level counters and record the end-to-end inference time. We conduct the experiments using CLTM-1.8B and 8× A100 GPUs. Table 2 shows that ZoomSynth can achieve the target upscaling performance goal from second-level counters within 1 second for any end-to-end upscaling ratios, demonstrating that ZoomSynth achieves real-time synthesis for various upscaling ratios.

7.4 Using CBF-based Counters

Due to limited memory and computing resources, many network devices adopt Counting Bloom Filters (CBF) [36] to implement packet or byte counters for obtaining approximate results. To see its impact on TSR performance, we implement a CBF-based counter with an array size of 300 and 2 hash functions, while using packet length as the hash key. Subsequently, we use the CBF-based counter to process the packet traces to obtain the counter time-series data, and then we use

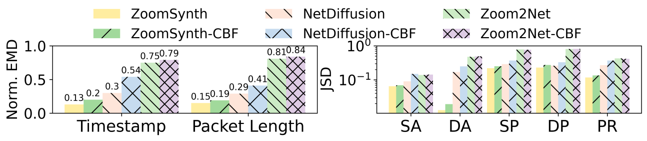


Figure 11: Normalized EMD and JSD between real and synthetic packet traces on UGR dataset using real counters and CBF counters as input, respectively.

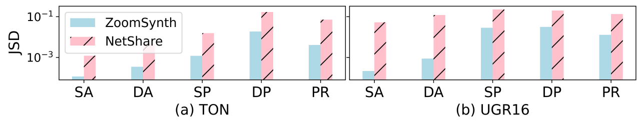


Figure 12: Performance comparison for packet header generation between NetShare and ZOOMSYNTH.

the data to train GTTs. Fig. 11 shows normalized EMD and JSD between real and synthetic packet traces on UGR dataset using real counters and CBF counters as input, respectively. Note that the packet and byte counters obtained using CBF on UGR dataset exceed the actual values by 7% and 12%, respectively. We can see ZOOMSYNTH outperforms NetDiffusion and Zoom2Net, improving EMD by 58.3% and 76.0%, and JSD by 52.3% and 80.1%, respectively. Besides, both ZOOMSYNTH and NetDiffusion have a significant increase in EMD for timestamp (53.8% for ZOOMSYNTH and 80.0% for NetDiffusion) and packet length (26.7% for ZOOMSYNTH and 41.4% for NetDiffusion), indicating that they are more sensitive to the accuracy of CBF.

7.5 Header Generation Model

To solely evaluate the quality of header generation, we use real nanosecond-level counter information as input, as this represents the packet prototype stripped of its header information. Fig. 12 provides an assessment of the capability of the header generation model. Our approach demonstrates an order of magnitude lower JSD values across various fields, proving that our header generation model significantly outperforms NetShare. This also underscores that the effectiveness of header generation is greatly dependent on the TSR results of the counters. We do not compare ZOOMSYNTH against NetDiffusion and Zoom2Net, since they do not have dedicated header generation modules.

7.6 Rule-following Model

We investigate the impact of counter rules on CLTM's header generation process. To do this, we construct multi-scale counter time-series and packet traces by filtering the original traffic traces using the ACL rules and use these data to train CLTM. Subsequently, we gather the packet outputs of ZOOMSYNTH with and without counter rules as input, respectively. Then we quantify the percentage of packets that adhere to the counter rules among all synthetic packets. For instance, when considering a rule like "Deny TCP", if counter rule is absent from the input, ZOOMSYNTH would use CLTM to recreate counters and use the header generation model to generate packet headers purely following the characteristics

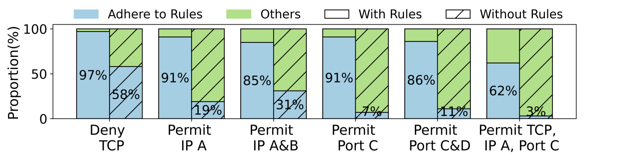


Figure 13: Comparison for the proportion of traffic which adheres to the counter rules with and without them as input.

Table 3: Summary of existing work.

Existing Work	Input	Output	Generality
DYNAMO [28], NS-2 [14]	Config Para.	Packets	✗
Swing [71], Tmix [73], LitGen [64]	Model Para.	Packets	✗
STAN [74]	-	Flows	✓
Redzovic et al. [61]	-	Sizes, Times	✓
DoppelGANger [50], NetShare [77]	Packets	Packets	✓
NetDiffusion [44, 46], PAC-GPT [47]	Prompts	Packets	✓
ZOOMSYNTH	Counters, Counter Rules	Packets, or Counters	✓

of the overall traffic flow.

We conduct this experiment on the CAIDA dataset, and use six counter rules as input to generate packet traces, respectively. Fig. 13 shows the comparison results for the proportion of traffic which adheres to the counter rules with and without counter rules as input. The results demonstrate that using counter rules as input can significantly benefit packet trace synthesis. For example, when the counter rule representing "Deny TCP" is not fed into the model, yet the counters represent traffic filtered by the rules, the proportion of TCP traffic in the final packet trace reaches 42%. In contrast, when we incorporate the "Deny TCP" rule into the packet header generation process, the final packet trace only has 3% TCP traffic. We can filter out the remaining 3% directly in the post-processing process without affecting the whole packet trace, because the proportion of the remaining TCP traffic among all traffic is small enough.

8 Related Work

There has been a large amount of work on traffic synthesis. In Table 3, we compare ZOOMSYNTH against them across three dimensions.

Traffic Synthesis Input: Existing work varies in their inputs. Simulation-/emulation-driven approaches demand detailed configurations, such as workloads and clients, to simulate network traffic in virtual environments. Model-driven approaches require distribution parameters derived from realistic traffic to establish statistical or structural models. ML-based models have diverse needs: some, like STAN [74] and Redzovic et al. [61], require no input, whereas others, such as DoppelGANger [50] and NetShare [77], need packet-level traffic data. In contrast, NetDiffusion [44, 46] and PAC-GPT [47] accept text prompts. Unlike these approaches, ZOOMSYNTH leverages counters to generate fine-grained traffic traces, achieving TSR not natively supported by them.

Traffic Synthesis Output: In addition to the inputs, the outputs of existing approaches also vary. Most approaches can produce packet-level traces, but some of them differ in the fields generated from the packet headers. Specifically, simulation-/emulation-driven approaches acquire packets via direct packet capture within their environments. STAN [74] is limited to generating flow-level traffic, while Redzovic et al. [61] can only produce the header fields of packet sizes and interarrival times. Besides producing packet-level traces, ZoomSynth further enables multi-scale synthesis of fine-grained counters directly from coarse-grained ones, offering flexibility for supporting various network applications.

Generality: In this regard, simulation-/emulation-driven and model-driven approaches are easy to analyze: they both rely on extracting specific features from realistic traffic to configure simulators, emulators, or models manually, making them inherently tied to specific workloads and thus limiting their adaptability for generalization. To overcome the issue, ZoomSynth supports fast new scenario adaptation.

Cross-Domain Techniques: ZoomSynth applies some techniques from other domains to traffic synthesis. ZoomSynth adapts CLIP [59], a model used to align image and text embeddings in image synthesis, to understand the relationship between counter rules and packet traces. LoRA [40], originally used to fine-tune LLMs for new scenarios, is employed in training GTT layers of CLTM, tailored for the traffic synthesis domain. Moreover, ZoomSynth chooses GPT-2 [11] as the header generation model, considering its generation efficiency.

9 Limitations and Future Work

Although ZoomSynth boosts many network applications by enabling TSR, it faces several limitations that require further investigation. We discuss these limitations in the following.

Lowering Device Fingerprinting Risks: Sharing raw counter datasets from devices can increase the risks of device fingerprinting. Counters vary in type, with some accumulating values over time and others resetting at regular intervals. Public access to these counters could allow adversaries to identify devices according to their types. To reduce risk, operators can process counter data into a uniform format, thereby obscuring the original types.

Extending ZoomSynth to Cases Needing Diverse Inputs: ZoomSynth has limitations in accurately synthesizing traffic for cases that require more than just counters as input. For example, generating traffic for video or audio applications, using only counters may not ensure that the traffic aligns with the specific characteristics of these applications. Future research could explore methods to synthesize traces that capture the characteristics of specific applications, incorporating more inputs such as application descriptions along with counters.

Mitigating Adversarial Misuse Risks: There is a potential for adversaries to exploit ZoomSynth for malicious purposes, such as using synthesized normal traffic patterns to evade detection by security devices like IDS and IPS. Although ZoomSynth focuses on TSR, the potential for misuse presents a significant challenge. This highlights the need for future research on preventing adversarial use.

Reducing Adversarial Effects in Traffic Synthesis: ML-based traffic synthesizers are vulnerable to adversarial traffic [29], which can lead to inaccurate output. Adversaries may tamper with training data, introducing deceptive patterns that result in unrealistic traces. To protect against this, the training datasets for GTTs should be carefully filtered to exclude adversarial samples. Identifying adversarial traffic is a well-established research area [39, 69] and is beyond the scope of our work.

Enhancing Support for More Downstream Tasks: ZoomSynth currently has limited support for downstream tasks requiring precise stateful protocol behaviors, such as congestion control and switch buffer monitoring, in line with existing work [45, 47, 50, 74, 77]. Future work will focus on enhancing packet generation with advanced stateful features (*e.g.* sequence numbers) and improving inter-arrival sequence modeling through better protocol and application behavior understanding.

Enhancing Generalization in Header Generation: ZoomSynth has generalization issue in header generation, as it limits header field values to those found in the training data. This reduces diversity compared to realistic traffic. To enhance generality while maintaining semantic conformity, future research could involve training a packet header model with domain-specific semantic knowledge.

10 Conclusion

We present ZoomSynth, the first TSR system to restore fine-grained traffic traces from coarse-grained counters. We design GTTs to capture traffic characteristics at particular granularities, and compose CLTM as a combination of a tree of GTTs and a rule-following model. ZoomSynth uses CLTM and a header generation model to achieve packet-trace synthesis, and can be adapted to new scenarios with LoRA. We prototype ZoomSynth with PyTorch and evaluate it on a real testbed using public datasets. ZoomSynth can satisfy the requirements for three downstream tasks. Compared to NetShare that takes packet-level trace as input, ZoomSynth can achieve comparable accuracy in terms of JSD and EMD, while obtaining more than 42.1% accuracy improvements compared to NetDiffusion.

Acknowledgments

We thank the anonymous NSDI reviewers for their constructive comments. Libin Liu and Li Chen are the corresponding authors of this paper. This work was supported by Zhongguancun Laboratory and the Beijing Outstanding Young Scientist Program (No. JWZQ20240101008), the National Natural Science Foundation of China under Grant U23B2001.

References

[1] Implementing SNMP on the Cisco IOS XR Software. https://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r3-9/system_management/configuration/guide/yc39xr12k_chapter10.html, 2011.

[2] Configuring NetFlow on Cisco IOS XR Software. https://www.cisco.com/c/dam/en/us/td/docs/iosxr/ncs5xx/netflow/63x/b-netflow-cg-63x-ncs5xx.html, 2021.

[3] NetFlow Configuration Guide for Cisco IOS XE. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/netflow/configuration/xe-3s/nf-xe-3s-book/cfg-nflow-data-expt-xe.html, 2022.

[4] SNMP Configuration Guide for Cisco IOS XE. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/snmp/configuration/xe-16-6/snmp-xe-16-6-book/nm-snmp-cfg-snmp-support.html, 2022.

[5] All Router Products of Cisco. https://www.cisco.com/c/en/us/support/routers/index.html, 2024.

[6] Capture files from Mid-Atlantic CCDC. https://www.netresec.com/?page=MACCDC, 2024.

[7] CIDDS - Coburg Intrusion Detection Data Sets. https://www.hs-coburg.de/forschung/forschungsprojekte-oeffentlich/informationstechnologie/cidds-coburg-intrusion-detection-data-sets.html, 2024.

[8] Cisco 8000 Hardware Emulator Datasheet. https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000-emulator/cisco8000-hardware-emulator-datasheet.html, 2024.

[9] Data Set for IMC 2010 Data Center Measurement. https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html, 2024.

[10] DPDK. https://www.dpdk.org/, 2024.

[11] Hugging Face gpt2. https://github.com/AUTOMATIC1111/stable-diffusion-webui, 2024.

[12] Linux Network Interface Statistics. https://www.kernel.org/doc/html/latest/networking/statistics.html, 2024.

[13] NetDiffusion Generator. https://github.com/noise-lab/NetDiffusion_Generator, 2024.

[14] ns-2. https://nsnam.sourceforge.net/wiki/index.php/Main_Page, 2024.

[15] Packet Framework Library of DPDK. https://doc.dpdk.org/guides/prog_guide/packet_framework.html, 2024.

[16] PyTorch Library. https://pytorch.org/, 2024.

[17] Router Interface Counters in SONiC. https://sonic.software/sai/group__SAIROUTERINTF.html, 2024.

[18] rte_table_action.h. https://doc.dpdk.org/api/rte__table__action_8h_source.html, 2024.

[19] SAI - Router Interface Specific API Definitions. https://sonic.software/sai/struct__sai__router__interface__api__t.html, 2024.

[20] SONiC. https://sonicfoundation.dev/, 2024.

[21] Stable Diffusion Web UI. https://github.com/AUTOMATIC1111/stable-diffusion-webui, 2024.

[22] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/catalog/datasets/passive_dataset/, 2024.

[23] Cisco 500 Series WPAN Industrial Routers. https://www.cisco.com/c/en/us/support/routers/500-series-wpan-industrial-routers/series.html, 2025.

[24] NCS 540 Series Routers. https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5xx/interfaces/24xx/b-interfaces-hardware-component-cg-24xx-ncs540/config-ethernet-interfaces.html, 2025.

[25] Paul Almasan, Miquel Ferriol-Galmés, Jordi Paillisse, José Suárez-Varela, Diego Perino, Diego López, Antonio Agustin Pastor Perales, Paul Harvey, Laurent Ciavaglia, Leon Wong, et al. Digital Twin Network: Opportunities and Challenges. *arXiv preprint arXiv:2201.01144*, 2022.

[26] Claise Benoit, Sadasivan Ganesh, Valluri Vamsi, and Djernaes Martin. Cisco systems netflow services export version 9. *RFC 3954*, 2004.

[27] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. Traffic Refinery: Cost-aware Data Representation for Machine Learning on Network Traffic. *Proc. ACM on Measurement and Analysis of Computing Systems*, 5(3):1–24, 2021.

[28] Tobias Bühler, Roland Schmid, Sandro Lutz, and Laurent Vanbever. Generating Representative, Live Network Traffic out of Millions of Code Repositories. In *Proc. ACM HotNets*, 2022.

[29] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *Proc. IEEE SP*, 2017.

[30] Yuxuan Chen, Rongpeng Li, Zhifeng Zhao, Chenghui Peng, Jianjun Wu, Ekram Hossain, and Honggang Zhang. Netgpt: A native-ai network architecture beyond provisioning personalized generative services. *arXiv preprint arXiv:2307.06148*, 2023.

[31] Kenjiro Cho. MAWI Working Group Traffic Archive. *https://mawi.wide.ad.jp/mawi/*, 2023.

[32] W Dai, J Li, D Li, AMH Tiong, J Zhao, W Wang, B Li, P Fung, and S Hoi. InstructBLIP: Towards General-purpose Vision-language Models with Instruction Tuning. In *Proc. NeurIPS*, 2023.

[33] Daizong Ding, Mi Zhang, Xudong Pan, Min Yang, and Xiangnan He. Modeling Extreme Events in Time Series Prediction. In *Proc. ACM SIGKDD*, 2019.

[34] Franck Djeumou, Cyrus Neary, Eric Goubault, Sylvie Putot, and Ufuk Topcu. Neural Networks with Physics-Informed Architectures and Constraints for Dynamical Systems Modeling. In *Proc. Machine Learning Research*, 2022.

[35] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image Super-Resolution using Deep Convolutional Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, 2015.

[36] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[37] Fengchen Gong, Divya Raghunathan, Aarti Gupta, and Maria Apostolaki. Zoom2Net: Constrained Network Telemetry Imputation. In *Proc. ACM SIGCOMM*, 2024.

[38] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *Proc. IEEE ICASSP*, 2013.

[39] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. New Directions in Automated Traffic Analysis. In *Proc. ACM CCS*, 2021.

[40] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*, 2021.

[41] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. 2023.

[42] Shuodi Hui, Huandong Wang, Zhenhua Wang, Xinghao Yang, Zhongjin Liu, Depeng Jin, and Yong Li. Knowledge Enhanced GAN for IoT Traffic Generation. In *Proc. ACM WWW*, 2022.

[43] Case Jeffrey, Mundy Russ, Partain David, and Stewart Bob. Introduction and applicability statements for internet-standard management framework. *RFC 3410*, 2002.

[44] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. NetDiffusion: Network Data Augmentation Through Protocol-Constrained Traffic Generation. *arXiv preprint arXiv:2310.08543*, 2023.

[45] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. NetDiffusion: Network Data Augmentation through Protocol-Constrained Traffic Generation. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 8(1):1–32, 2024.

[46] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Generative, High-Fidelity Network Traces. In *Proc. ACM HotNets*, 2023.

[47] Danial Khosh Kholgh and Panos Kostakos. PAC-GPT: A Novel Approach to Generating Synthetic Network Traffic with GPT-3. *IEEE Access*, 2023.

[48] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-Oblivious Traffic Engineering: The Road Not Taken. In *Proc. USENIX NSDI*, 2018.

[49] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-Realistic Single Image Super-Resolution using a Generative Adversarial Network. In *Proc. IEEE CVPR*, 2017.

[50] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. Using GANs for Sharing Networked Time Series Data: Challenges, Initial Promise, and Open Questions. In *Proc. ACM IMC*, 2020.

[51] Zinan Lin, Hao Liang, Giulia Fanti, and Vyas Sekar. RareGAN: Generating Samples for Rare Classes. In *Proc. AAAI*, 2022.

[52] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual Instruction Tuning. In *Proc. NeurIPS*, 2023.

[53] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proc. ACM SIGCOMM*, 2019.

[54] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with Univmon. In *Proc. ACM SIGCOMM*, 2016.

[55] Gabriel Maciá-Fernández, José Camacho, Roberto Magán-Carrión, Pedro García-Teodoro, and Roberto Therón. UGR'16: A New Dataset for the Evaluation of Cyclostationarity-based network IDSs. *Computers & Security*, 73:411–424, 2018.

[56] ML Menéndez, JA Pardo, L Pardo, and MC Pardo. The jensen-shannon divergence. *Journal of the Franklin Institute*, 334(2):307–318, 1997.

[57] Nour Moustafa. A New Distributed Architecture for Evaluating AI-based Security Systems at the Edge: Network TON_IoT Datasets. *Sustainable Cities and Society*, 72:102994, 2021.

[58] Jian Qu, Xiaobo Ma, and Jianfeng Li. Trafficgpt: Breaking the token barrier for efficient long traffic analysis and generation. *arXiv preprint arXiv:2403.05822*, 2024.

[59] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning Transferable Visual Models from Natural Language Supervision. In *Proc. PMLR ICML*, 2021.

[60] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[61] Hasan Redžović, Aleksandra Smiljanić, and Milan Bjelica. IP Traffic Generator Based on Hidden Markov Models. *parameters*, 1(2):1.

[62] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. Flow-based Network Traffic Generation using Generative Adversarial Networks. *Computers & Security*, 82:156–172, 2019.

[63] Markus Ring, Sarah Wunderlich, Dominik Grüdl, Dieter Landes, and Andreas Hotho. Flow-based Benchmark Data Sets for Intrusion Detection. In *Proc. ECCWS*, 2017.

[64] Chloé Rolland, Julien Ridoux, and Bruno Baynat. LiTGen, a Lightweight Traffic Generator: Application to P2P and Mail Wireless Traffic. In *Proc. International Conference on Passive and Active Network Measurement*, pages 52–62, 2007.

[65] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution Image Synthesis with Latent Diffusion Models. In *Proc. IEEE CVPR*, 2022.

[66] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40:99–121, 2000.

[67] Amin Shabani, Amir Abdi, Lili Meng, and Tristan Sylvain. Scaleformer: Iterative Multi-scale Refining Transformers for Time Series Forecasting. In *Proc. ICLR*, 2023.

[68] Binh Tang and David S Matteson. Probabilistic Transformer for Time Series Analysis. In *Proc. NeurIPS*, 2021.

[69] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. BotFinder: Finding Bots in Network Traffic without Deep Packet Inspection. In *Proc. ACM CoNEXT*, 2012.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Proc. NeurIPS*, 2017.

[71] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and Responsive Network Traffic Generation. In *Proc. ACM SIGCOMM*, 2006.

[72] Yuejie Wang, Qiutong Men, Yao Xiao, Yongting Chen, and Guyue Liu. ConfMask: Enabling Privacy-Preserving Configuration Sharing via Anonymization. In *Proc. ACM SIGCOMM*, 2024.

[73] Michele C Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F Donelson Smith. Tmix: A Tool for Generating Realistic TCP Application Workloads in NS-2. *ACM SIGCOMM Computer Communication Review*, 36(3):65–76, 2006.

[74] Shengzhe Xu, Manish Marwah, Martin Arlitt, and Naren Ramakrishnan. STAN: Synthetic Network Traffic Generation with Generative Neural Models. In *Proc. International Workshop on Deployable Machine Learning for Security Defense*, 2021.

[75] Kun Yang, Samory Kpotufe, and Nick Feamster. A Comparative Study of Network Traffic Representations for Novelty Detection. *arXiv preprint arXiv:2006.16993*, 2020.

[76] 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 *Proc. ACM SIGCOMM*, 2020.

[77] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical GAN-based Synthetic IP Header Trace Generation using NetShare. In *Proc. ACM SIGCOMM*, 2022.

[78] Zongsheng Yue, Jianyi Wang, and Chen Change Loy. ResShift: Efficient Diffusion Model for Image Super-Resolution by Residual Shifting. In *Proc. NeurIPS*, 2023.

[79] Yulun Zhang, Kunpeng Li, Kai Li, Lichen Wang, Bineng Zhong, and Yun Fu. Image Super-Resolution using Very Deep Residual Channel Attention Networks. In *Proc. ECCV*, 2018.

Appendix

The appendix describes additional details that we could not fit into the main paper.

.1 Additional Details of CLTM

.1.1 Training Data Preparation for GTT Models

We need traffic data with various resolutions to train GTT $^{k}_{\delta}$ models. To obtain them, we aggregate raw pcap data from public datasets into counter time-series at various granularities, according to k. We have introduced all the public datasets we use in §6. The number of needed training data couple is $\frac{\theta}{k}$, where the overall end-to-end upscaling raito for CLTM is θ. Taking packet synthesis from second-level counters (overall end-to-end upscaling ratio is 10^9) and k=10 as an example, a total nine couples of counter time-series at different granularities are needed, including (1s, 100ms), (100ms, 10ms), ..., and (10ns, 1ns). These counter time-series is structured as `pandas.DataFrame` with the following columns: time, packet count, and byte count. Moreover, CLTM increases GTTs within each stage by k to process different partitions of counter time-series to improve scalability, as the data size increases by k along with the upscaling stages. Considering the temporal dependency between the neighbouring partitions, we intentionally introduce overlap between them, which enhances the ability of GTTs to capture long-term time-series characteristics.

.1.2 Choosing Upscaling Factor k

As analyzed in §2.4, each GTT concentrates on a particular upscaling stage with a fixed upscaling factor k, which is important for the tradeoff between end-to-end traffic synthesis accuracy and model training cost. A GTT with small k can achieve satisfactory end-to-end synthesis accuracy, yet may incur extra training costs as involving more GTTs for more upscaling steps.

The choices of k are different across different datasets. Generally, a GTT having smaller upscaling factor can have better TSR performance. For example, GTT $^{10}_{1\text{ms}}$ outperforms GTT $^{100}_{1\text{ms}}$. For a fixed end-to-end upscaling ratio, however, smaller k would increase CLTM stages. Each stage may have a degree of synthesis errors, which may accumulate along with upscaling stages. In addition, the performance gains obtained by choosing GTT $^{10}_{1\text{ms}}$ over GTT $^{100}_{1\text{ms}}$ are not consistent across the datasets, since they exhibit various characteristics at different granularities. Therefore, we employ experimental methods to determine the value of k.

We use TON [57] and CIDDS [7] datasets as an example to show how we experimentally choose the value of k. We restore packet trace from second-level counters and make k equal to 5, 10, 10^2, and 10^3, respectively. These choices of k are enough for us to determine the k values for the two datasets. For each k value, we calculate the normalized EMD and JSD between real and synthetic packet traces. Fig. 14 shows the results. We can see, for the two datasets, the normalized EMD and JSD results have different trends along with the increase of k. For TON dataset, the normalized EMD and JSD results increase along with k, and they are similar for k=5 and k=10, and increase significantly when $k>10$. However, for CIDDS dataset, from k=5 to $k=10^2$, both EMD and JSD results of the most packet header fields decrease along with k, and then they increase largely when $k=10^3$, compared to $k=10^2$.

Table 4 shows the corresponding end-to-end model training time for different values of k. The measurements are done on the testbed described in §7. We only show the training time using TON dataset, since it is consistent for all the datasets that more CLTM stages with more GTT models would lead to longer training time. Taking accuracy and training time into account, we recommend to set k as 10 and 10^2 for the TON and CIDDS datasets, respectively. Other datasets adopt the same method to choose their k.

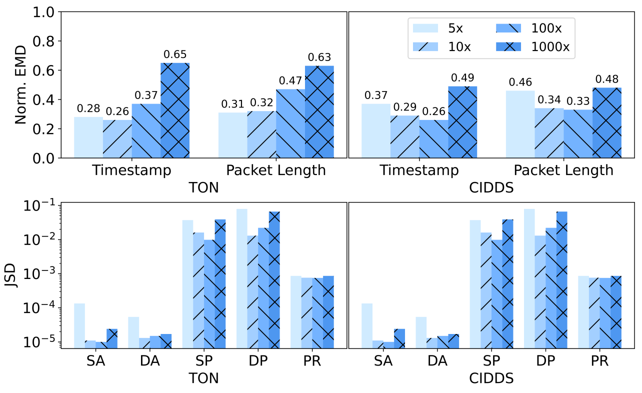


Figure 14: Normalized EMD and JSD between real and synthetic packet traces for different choices of k on TON and CIDDS datasets (SA: source address, DA: destination address, SP: source port, DP: destination port, PR: protocol).

Table 4: The end-to-end training time of ZoomSynth for different values of k using TON dataset.

k	5	10	10^2	10^3
Training Time (GPU Hours)	99.15	61.91	6.77	6.31

.1.3 Training CLTM

In the training phase of CLTM, we combine the output vector of pre-trained rule-following model and the vector of counter time-series at a particular granularity and feed their combinations into GTT to train it. Specifically, when no counter rules are available, its input to GTT is a zero vector. To combine them, we directly copy and extend the output vector of pre-trained rule-following model into 2 dimensions, ensuring that the two vectors are of same shape before concatenating them.

.1.4 Handling Sparsity with Early Stopping

We observe some intermediate fine-grained counter time-series exhibit sparsity. For instance, when upscaling a counter

Table 5: Hyperparameters of ZoomSynth models.

Models	Param.	Value
GTT	Transformer Layers	15
	Multi-Headers	4
	Hidden Size	256
	Dropout	0.1
	Learn Ratio	0.001 - 0.01 (vary with iterations)
	Batch Size	10 - 64 (vary with data volumes)
	λ in Loss Func.	0.1 - 1.5 (vary with iterations)
	σ in Loss Func.	0 - 1 (vary with iterations)
Header Gen. Model	Vocab Size	50000
	Embedding Dim.	128
	Generation Model	GPT2-Small
Rule-following Model	Text Encoder	Transformer-63M
	Similarity Matrix	256*256
Downstream Task Adaptor	Header Generation	Transformer-38M
	Task-specific Model	3-layers DT or MLP

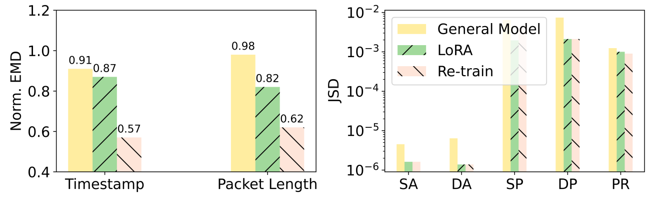


Figure 15: EMD and JSD between real and synthetic distributions on CIDDS using four kind of models: specific model trained on CIDDS, general model trained on combined dataset, LoRA on general model, re-training general model.

time-series from 10ms to 1ms, there are 8 upscaled 1ms-counters whose values are 0; during the inference phase, for these counters, ZoomSynth will stop the upscaling process of the subsequent GTT models in CLTM. In addition, if both the packet count of the other 2 1ms-counters are 1, this represents the 1ms-counter has reached the finest granularity and its byte count is the packet size, we can stop its upscaling process as well. By classifying the sparsity of counter time-series during the upscaling process, ZoomSynth accelerates the inference of CLTM with an early stop mechanism, avoiding useless upscalings.

.1.5 *Fast New Scenario Adaptation*

CLTM is designed to be a general foundation model for TSR and we train it using the common traffic traces, which prevents it from overfitting to the specific features of any specific dataset, while directing the model to focus on the underlying and universal traffic features across various scenarios. However, we also admit that traffic characteristics can vary significantly for different scenarios, and can also change over time. Therefore, we design an approach for adapting CLTM to the new scenarios with low overhead.

We leverage the LoRA-based approach to empower CLTM to adapt to new scenarios quickly. This is a systematic application of LoRA [40] for traffic synthesis building on its success in fine-tuning of LLMs. The core of LoRA is training the low-rank parameter matrices of the whole model with a small amount of data from new scenarios during the re-training phase. After finishing the training, it injects the re-trained parameters to the original model and replaces the corresponding ones. We use the same approach to train the low-rank parameter matrices of CLTM.

We only choose the GTT model layer of CLTM for finer-grained counter synthesis to perform re-training with LoRA approach, which further accelerates the adaptation process. We observe that across various scenarios, coarse-grained traffic statistics remain consistent, the features of the corresponding fine-grained traffic are different. Therefore, as shown in Fig. 1, for GTTs within the upper layers of CLTM, they are similar to perform uniform upscalings, while GTTs within the lower layer vary in capturing the traffic fine-grained features. Based on the observation and analysis, we should re-train GTTs for finer-grained counter synthesis. For instance, we choose GTT $^{10}_{10\text{ns}}$ and GTT $^{10}_{1\text{ns}}$ in the last two stages of CLTM to be retrained for packet synthesis from second-level counter time-series.

.2 Training Header Generation Model

For preparing the training dataset, we process the raw packet trace to the same granularity of counter time series. We use the processed counter time series and packet trace to train the header generation model. The training method is that we mask header fields randomly and use the model to regenerate the relative fields. We retrain the header generation model on our hybrid dataset, with the training process taking approximately 8 days.

.3 Implementation Details of ZoomSynth

.3.1 *Hyperparameters of* ZoomSynth *Models*

Table 5 lists the hyper-parameter setups of ZoomSynth for each model.

.3.2 *Operation APIs of CLTM*

The six APIs to support operations of CLTM introduced in §6 include `def_TSR_ratio(·)` for defining the end-to-end upscaling ratio, `def_GTT_factor(·)` for choosing the factor k of the GTT, `def_GTT_num(·)` for deciding the number of GTTs in CLTM, `def_GTT_model(·)` for configuring model hyperparameters, `def_training_iter(·)` for setting model training parameters.

.4 Additional Experimental Results

.4.1 *Adapting to New Scenarios*

To simulate adapting to a new scenario, we design two datasets: the D_0 dataset combines UGR16, TON and MAWI datasets; and the D_+ dataset which contains only the CIDDS dataset whose size is 5% of that of D_0. We then train three models: 1) the *General Model* is a CLTM trained on D_0; 2) with the *General Model* as base, we train the LoRA layers on D_+, which is our recommended procedure to adapt CLTM to new scenarios; we denote the combined model of *General Model* and LoRA layers as *LoRA*; 3) Using the *General Model* as a starting point, we run 50 iterations of training using D_+, and we call the resulting model *Re-train Model*.

Fig. 15 shows the results. We observe that the *Re-train Model* achieves the best performance, and *LoRA* can greatly

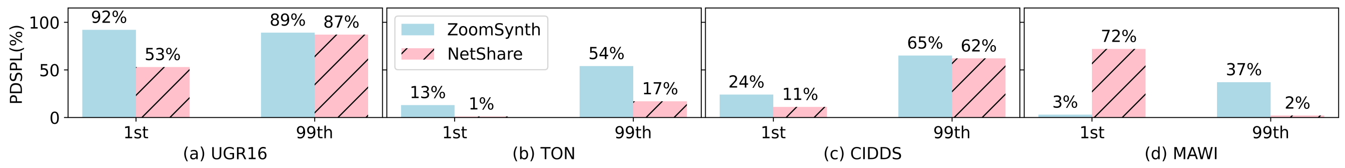


Figure 16: Proportions of deviations for synthetic packet lengths at the 1st and 99th percentile generated by ZOOMSYNTH and NetShare on the UGR16, TON, CIDDS, and MAWI datasets (PDSPL: Proportions of Deviations for the Synthetic Packet Lengths).

improve the *General Model*'s performance in the new scenario. *LoRA* can improve the generation quality of timestamps and packet lengths by 4.3% and 15%, respectively. For header generation, *LoRA* achieves roughly the same performance as the *Re-train Model*. Since re-training the complete CLTM model takes up significantly more resource and time than training a few LoRA layers, we believe adapting to new scenarios with *LoRA* is an attractive alternative for users with limited time, data, and resources.

.4.2 *Deviation Proportions of Synthetic Packet Lengths*

To further explore the generation performance of ZOOMSYNTH and NetShare in synthesizing packet lengths, we look at the proportions of deviation between the synthetic and realistic packet lengths. This metric, referred to as the proportion of deviations for synthetic packet lengths (PDSPL), is defined as follows:

$$PDSPL_i = \frac{|PL_i^{syn} - PL_i^{real}|}{PL_i^{real}} \times 100\%.$$

Here, PL_i^{real} denotes the packet length at the ith percentile of the sorted realistic packet length time-series, arranged from smallest to largest, while PL_i^{syn} represents the corresponding value of the sorted synthetic packet length time-series. This metric measures how closely the synthetic packet lengths match the realistic ones across different percentiles.

Fig. 16 shows the proportions of deviations for the 1st and 99th percentile synthetic packet lengths generated by ZOOMSYNTH and NetShare on the UGR16, TON, CIDDS, and MAWI datasets. We can see NetShare exhibits lower deviations at both the 1st and 99th percentiles, except for the 1st percentile packet length on the MAWI dataset. We analyze that this is because GAN-based models are sensitive to fluctuations and extreme values in time-series data [46], making NetShare perform better in generating packet lengths around extreme values. However, this sensitivity also causes NetShare to tend to overfit the variations of the packet length time-series, leading to worse EMD performance compared to ZOOMSYNTH, as we observe in Fig. 8.

.5 Cisco Router Products Supporting Counters

Table 6 shows Cisco's router products which support counters, their configuration commands to display counters, and the corresponding online configuration manuals. These routers are the 41 out of 42 ones from 500 series to 12000 series listed at Cisco's official site [5]. Their configuration manuals state that they support counters, which can be obtained via command line interface (CLI) with the corresponding configuration commands. The router product which does not appear in Table 6 is 8000 series virtual router emulator, whose data sheet introduces that some hardware counters are not supported by it [8]. It is not that it does not support counters at all.

We list the links of the online configuration manuals referenced in Table 6 below.

[1] https://www.cisco.com/c/en/us/support/routers/500-series-wpan-industrial-routers/series.html

[2] https://www.cisco.com/c/en/us/td/docs/routers/ncs5xx/ncs520/configuration/guide/CE/17-1-1/b-ce-xe-17-1-1-ncs520/using-ethernet-operations-administration-and-maintenance.html

[3] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5xx/interfaces/24xx/b-interfaces-hardware-component-cg-24xx-ncs540/config-ethernet-interfaces.html

[4] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs560/interfaces/24xx/b-interfaces-hardware-component-cg-24xx-ncs560/configuring_ethernet_interfaces.html

[5] https://www.cisco.com/c/en/us/td/docs/routers/access/800/829/software/configuration/guide/b_IR800config/b_cellular.html

[6] https://www.cisco.com/c/en/us/td/docs/routers/access/800M/software/800MSCG/routconf.html

[7] https://www.cisco.com/c/en/us/td/docs/routers/access/900/software/configuration/guide/900SCG/routconf.html

[8] https://www.cisco.com/c/en/us/td/docs/routers/asr903/software/guide/chassis/17-1-1/b-config-guide-xe-17-1-1-asr900/configuring_ethernet_interfaces.html

[9] https://www.cisco.com/c/en/us/td/docs/wireless/asr_901/Configuration/Guide/b_asr901-scg/b_asr901-scg_chapter_01010.html

Table 6: Cisco router products which support counters, their configuration commands to display counters, and the corresponding online manuals.

No.	Router Products	Config Commands	Config Manuals
1	500 Series WPAN Industrial Routers	show wpan packet-count	[1]
2	Network Convergence System 500 Series Routers	show ethernet oam statistics	[2]
3	Network Convergence System 540 Series Routers	show interface [name]	[3]
4	Network Convergence System 560 Series Routers		[4]
5	800 Series Industrial Integrated Services Routers		[5]
6	800 Series Routers		[6]
7	900 Series Integrated Services Routers		[7]
8	ASR 900 Series Aggregation Services Routers		[8]
9	ASR 901 10G Series Aggregation Services Routers		[9]
10	ASR 901 Series Aggregation Services Routers		
11	ASR 920 Series Aggregation Services Router		[10]
12	1000 Series Connected Grid Routers		[11]
13	Cisco 1000 Series Integrated Services Routers		[12]
14	ASR 1000 Series Aggregation Services Routers		[13]
15	Cloud Services Router 1000V Series	show platform packet-trace statistics	[14]
16	Catalyst IR1100 Rugged Series Routers	show interface [name]	[15]
17	Catalyst IR1800 Rugged Series Routers		[16]
18	1900 Series Integrated Services Routers		[17]
19	2000 Series Connected Grid Routers		[18]
20	2900 Series Integrated Services Routers		[17]
21	3900 Series Integrated Services Routers		
22	3000 Series Industrial Compute Gateways	show iox detail	[19]
23	4000 Series Integrated Services Routers	show interface [name]	[20]
24	5000 Series Enterprise Network Compute System	show ip traffic interface [name]	[21]
25	Network Convergence System 5000 Series	show interface [name]	[22]
26	Network Convergence System 5500 Series		[23]
27	Network Convergence System 5700 Series		[24]
28	5900 Series Embedded Services Routers		[25]
29	Network Convergence System 6000 Series Routers		[26]
30	ESR6300 Embedded Series Routers		[27]
31	8000 Series Routers	sh int [name]	[28]
32	Catalyst 8000V Edge Software	show platform packet-trace statistics	[29]
33	Cisco Catalyst 8200 Series Edge Platforms	show interface [name]	[30]
34	Cisco Catalyst 8300 Series Edge Platforms		
35	Cisco Catalyst 8200 Series Edge uCPE		[31]
36	Cisco Catalyst 8300 Series Edge uCPE		
37	Catalyst 8500 Series Edge Platforms	show platform packet-trace statistics	[32]
38	Catalyst 8500L Series Edge Platforms		
39	ASR 9000 Series Aggregation Services Routers	show interface [name]	[33]
40	IOS XRv 9000 Router	show controller dpa statistics global	[34]
41	XR 12000 Series Router	show interface [name]	[35]

[10] https://www.cisco.com/c/en/us/td/docs/routers/asr920/configuration/guide/chassis/17-1-1/b-Chassis-Guide-xe-17-1-asr920/m-eth_im_config-asr920.html

[11] https://www.cisco.com/c/en/us/td/docs/routers/connectedgrid/cgr1000/1_0/software/configuration/guide/wifi/WiFi_Book/wifi_interface.html

[12] https://www.cisco.com/c/en/us/td/docs/routers/access/isr1100/software/configuration/xe-17/isr1100-sw-config-xe-17/troubleshooting.html

[13] https://www.cisco.com/c/en/us/td/docs/routers/asr1000/configuration/guide/chassis/asr1000-software-config-guide/bdi-asr.html

[14] https://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/configuration/b_CSR1000v_Configuration_Guide/packet_trace.html

[15] https://www.cisco.com/c/en/us/td/docs/

routers/access/1101/software/configuration/guide/b-cisco-ir1101-scg/m-basic-router-cli-configuration.html

[16] https://www.cisco.com/c/en/us/td/docs/routers/access/IR1800/software/b-cisco-ir1800-scg/m-basic-router-cli-config.html

[17] https://www.cisco.com/c/en/us/td/docs/routers/access/1900/software/configuration/guide/Software_Configuration/routconf.html

[18] https://www.cisco.com/c/en/us/td/docs/routers/access/2000/CGR2010/software/configuration/guide/CGR_2010/routconf.html

[19] https://www.cisco.com/c/en/us/td/docs/routers/ic3000/deployment/guide/b_IC3000_deployment_guide/b_IC3000_deployment_guide_chapter_0100.html

[20] https://www.cisco.com/c/en/us/td/docs/routers/access/4400/software/configuration/xe-16-6/isr4400swcfg-xe-16-6-book/basicconfg.html

[21] https://www.cisco.com/c/en/us/td/docs/routers/sdwan/configuration/system-interface/ios-xe-17/systems-interfaces-book-xe-sdwan/configure-interfaces.html

[22] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5000/interfaces/711x/configuration/guide/b-interfaces-hardware-component-cg-ncs5000-711x/advanced-configuration-modification-of-management-ethernet-interface.html

[23] https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5500/interfaces/24xx/configuration/guide/b-interfaces-hardware-component-cg-ncs5500-24xx/configuring-ethernet-interfaces.html

[24] https://www.cisco.com/c/en/us/products/collateral/routers/network-convergence-system-5500-series/network-convergence-sys-wp.html

[25] https://www.cisco.com/c/en/us/td/docs/solutions/GGSG-Engineering/15-4-3M/config-guide/Configuration-Guide/Config.html

[26] https://www.cisco.com/c/en/us/td/docs/routers/ncs6000/software/ncs6k-7-6/b-interfaces-hardware-component-cg-ncs6000-76x/configuring-ethernet-interfaces.html

[27] https://www.cisco.com/c/en/us/td/docs/routers/embedded/6300/software/config/b_ESR6300_config/b_ESR6300_config_chapter_010.html

[28] https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/Interfaces/24xx/configuration/guide/b-interfaces-config-guide-cisco8k-r24xx/m-configuring-ethernet-interfaces-8k.html

[29] https://www.cisco.com/c/en/us/td/docs/routers/C8000V/Configuration/c8000v-edge-software-configuration-guide/packet_trace.html

[30] https://www.cisco.com/c/en/us/td/docs/routers/cloud_edge/c8300/software_config/cat8300swcfg-xe-17-book/isr9000swcfg-xe-16-12-book_chapter_010.html

[31] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/interface/command/ir-cr-book/ir-s4.html

[32] https://www.cisco.com/c/en/us/td/docs/routers/cloud_edge/c8500/software-configuration-guide/c8500-software-config-guide/packet_trace.html

[33] https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/24xx/interfaces/configuration/guide/b-interfaces-hardware-component-cg-asr9000-24xx/configuring-ethernet-interfaces.html

[34] https://www.cisco.com/c/en/us/td/docs/routers/virtual-routers/configuration/guide/b-xrv9k-cg/m-cisco-ios-xrv-9000-data-plane-specific-features.html

[35] https://www.cisco.com/c/en/us/products/collateral/routers/xr-12000-series-router/product_data_sheet0900aecd803f856f.html