FlowChronicle: Synthetic Network Flow Generation through Pattern Set Mining

JOSCHA CÜPPERS*, CISPA Helmholtz Center for Information Security, Germany
ADRIEN SCHOEN*, Inria, Univ Rennes, IRISA, France
GREGORY BLANC, Télécom SudParis, Institut Polytechnique de Paris, France
PIERRE-FRANCOIS GIMENEZ, CentraleSupélec, Inria, Univ Rennes, IRISA, France

Network traffic datasets are regularly criticized, notably for the lack of realism and diversity in their attack or benign traffic. Generating synthetic network traffic using generative machine learning techniques is a recent area of research that could complement experimental test beds and help assess the efficiency of network security tools such as network intrusion detection systems. Most methods generating synthetic network flows disregard the temporal dependencies between them, leading to unrealistic traffic. To address this issue, we introduce *FlowChronicle*, a novel synthetic network flow generation tool from mined patterns and Bayesian networks. As a core component, we propose a novel pattern miner in combination with statistical models to preserve temporal dependencies. We empirically compare our method against state-of-the-art techniques on several criteria, namely realism, diversity, compliance, and novelty. This evaluation demonstrates the capability of *FlowChronicle* to achieve high-quality generation while significantly outperforming the other methods in preserving temporal dependencies between flows. Besides, in contrast to deep learning methods, the patterns identified by *FlowChronicle* are explainable, and experts can verify their soundness. Our work substantially advances synthetic network traffic generation, offering a method that enhances both the utility and trustworthiness of the generated network flows.

CCS Concepts: • **Information systems** → **Data mining**; • **Networks** → **Network simulations**; **Network performance modeling**.

Additional Key Words and Phrases: Network Traffic; Synthetic Data Generation; Pattern Mining; Network Simulation; Minimum Description Length

1 Introduction

Evaluating network security tools, such as intrusion detection systems (IDS) [69] or firewalls [19], and conducting network measurement campaigns, such as applications testing [24] or device identification [50], requires the systematic collection and sharing of network traffic datasets.

However, multiple studies have highlighted recurring issues with network traffic datasets, such as quality [34], density [31], and labeling accuracy [22]. In fact, instead of using actual network

---

*Equal contribution, listed alphabetically.

Authors' Contact Information: Joscha Cüppers, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, joscha.cueppers@cispa.de; Adrien Schoen, Inria, Univ Rennes, IRISA, Rennes, France, adrien.schoen@inria.fr; Gregory Blanc, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France, gregory.blanc@telecom-sudparis.eu; Pierre-Francois Gimenez, CentraleSupélec, Inria, Univ Rennes, IRISA, Rennes, France, pierre-francois.gimenez@centralesupelec.fr.

captures, which may not be shared due to confidentiality and privacy risks [31], researchers have proposed to generate traffic in a controlled environment. This synthetic traffic does not result from human network activities but from network automatons, such as web crawlers [60], email generators [24], or bots that operate specific applications following predefined user profiles [65]. Simulating traffic effectively sidesteps the issues associated with human-generated traffic [60]: e.g., the risk of leaking confidential information is minimized since the *users* are simulated and not real. Furthermore, the controllable behavior of these simulated users allows easier labeling of the resulting traffic compared to traffic generated from human interactions [66].

One major drawback of traffic simulation is scalability: once the simulation is launched and the traffic is being recorded, the behavior cannot be adapted to a new constraint that was not implemented at the starting time. The resulting traffic cannot be adapted to produce unplanned behavior and, therefore, hardly corresponds to another configuration in terms of hosts or activities [1]. Such adaptation will often necessitate re-running the entire simulation, which is time-consuming and costly. To address the issues of both real and simulated traffic, the research community has resorted to synthetic data generation, which relies on a modeling algorithm that learns existing traffic characteristics to reproduce them [1, 4, 49]. Such algorithms enable the generation of synthetic traffic free from sensitive user information and with more precise labeling. For example, in the case of *data augmentation*, generative algorithms can be used to produce more samples of a given traffic class, greatly reducing manual and error-prone labeling effort during dataset creation [49]. In addition to these two features, it is expected that such algorithms enable the generation of new traffic to assess the generalization of network security measures to new environments [1] while preserving consistency/compliance [47].

This article focuses on the generation of benign traffic in the format of *network flows*. For this specific task, several classes of synthetic data generation models have been proposed so far, including *Generative Adversarial Networks* (GAN) [3, 4, 6, 54, 59], *Variational AutoEncoders* (VAE) [17, 45], autoregressive model [75], Bayesian networks [63]. A problem these models have with network flow generation is their lack of modeling the temporal dependencies among flows [4]: for example, several works [3, 59, 63] are sampling network flows independently. We argue that this type of generation is insufficient for real-life applications.

We therefore propose *FlowChronicle*, a novel synthetic network flow generation method based on pattern mining [2, 72]. Our first contribution is a powerful pattern language especially designed to match network flows, that not only captures relevant value combinations within flows but also between different network flows. We formalize this problem as mining a set of non-redundant patterns that best summarize the training network flow dataset.

Our second contribution is *FlowChronicle*, a data generation mechanism based on the learned representation. As we will show, this approach generates highly realistic network traffic and respects the protocol specifications. Moreover, since *FlowChronicle* generates synthetic data directly from the temporal patterns mined from the training dataset, this results in synthetic traffic that preserves temporal relations among network flows. In addition, the patterns mined by our generating model are interpretable and auditable.

Finally, we compare our model to other state-of-the-art generators to show that our model, on top of providing the best overall quality of synthetic traffic, also preserves time dependencies.

The remainder of the paper is articulated as follows: we introduce useful preliminaries in Section 3, before detailing our contributions, namely the pattern language, the network flow generation method, and our evaluation method, in Sections 4, 5 and 6, respectively. The evaluation against other generators and their results are discussed in Section 7, before concluding in Section 8.

2 Related Works

Although many statistical models can be used for synthetic data generation, in recent years deep-learning-based methods have been increasingly favored due to their ability to generate high-dimensional data such as images. Especially GAN [20], VAE [32], and Transformers [58, 71] have been shown to produce highly realistic synthetic data. These methods have also been applied to generate multiple types of network traffic data, including raw packet contents [8, 12, 25, 44], sequences of headers [64, 77], flow features [3, 40, 43, 59, 63, 77] or even temporal series of features [26, 39]. In the following, we will focus on methods for synthetic network flow generation.

The first use of these generative methods for creating synthetic legitimate network flow generation has been proposed by Ring et al. [59]. It was quickly followed by Manocchio et al. [43] who have shown that, when applied to network traffic, WGAN-GP [23] is prone to a phenomenon called *mode collapse*, which is when the generated data only cover a part of the training data distribution. Anande et al. [3] and Bourou et al. [6] have shown that synthetic data generation methods for tabular data [74] also work well for network flow datasets.

The major limit of these solutions is that while the generation preserves the dependencies across network features within a flow, it does not consider the dependencies among the flows. For example, before establishing an HTTP connection, a client might have to reach a DNS server to resolve the domain name of the requested website. That is, a single action of the client will, hence, lead to two flows, one to the DNS server and the other to the website host. All the previous solutions, due to sampling new network flows independently, do not model those inter-flow dependencies [4]. To solve that issue, Xu et al. [75] implement a solution that not only model dependencies within one network flow but also dependencies across network flows by using an autoregressive model. Lin et al. [39] propose a different approach where they model the problem of network traffic generation as a temporal series generation, where a multidimensional time series represents the activity. This method was then adapted by Yin et al. [77] to generate complete network flows. Recently, Schoen et al. [63] have shown that this solution does not generate realistic network flows and tend to produce flows that do not comply with basic network-specific checks. Therefore, generating realistic network flows that also include temporal dependencies remains an open challenge.

We propose a different approach based on pattern mining. Traditional pattern miners discover all patterns that occur more frequently than a user-set threshold [2, 36]. Such approaches often result in an exaggerated number of patterns, the so-called *pattern explosion*. Closed episode mining alleviates this [73, 76], but is very sensitive to noise. More recent approaches focus on reporting all patterns that are significant under a null hypothesis, e.g. that all events occur independently of each other [28, 41, 57, 70]. We use a pattern set mining approach where the goal is to find a set of patterns that describe the data as a whole [72]. This approach should return a relatively small set of patterns that capture the data distribution well. Pattern set miners have been proposed for many different data modalities, like sequences [68] and graphs [33]. Recent approaches go beyond 'and' combinations and capture rules [13], 'xor' relations [14], and generalized patterns that capture similar structures within one pattern [10]. Closely related to our setting are recent approaches that focus on the delays between events [9, 15]. As such, none of the above methods addresses our needs, we hence introduce our own novel pattern language that can model transactional relations, i.e., structure in a flow, as well as sequential relations, i.e., structure between flows.

Researchers have successfully used pattern mining for various tasks related to network data, such as near-live network monitoring [38], efficient computation of heavy hitters [52], and to provide succinct visualization of network flow traces [18]. Most works use pattern mining in the context of anomaly detection: Jakhale and Patil [27] build on the work of Li and Deng [38] to detect anomalous flows. In contrast, Brauckhoff et al. [7] use frequent pattern mining to summarize flows that cause

anomalies. Paredes-Oliva et al. [53] propose to classify extracted patterns as either anomalous or not, in contrast to individual flows or time intervals. Unlike us, none consider patterns over multiple flows, and all use frequent pattern mining. To the best of our knowledge, we are the first to use pattern mining to generate synthetic network flows.

3 Background

In this section, we present some background knowledge we rely upon for *FlowChronicle*, in particular, related to pattern mining and machine learning.

3.1 Network flows

Network traffic encompasses all the packets that are exchanged among hosts within a specific network over a designated time frame. A *network flow* is an abstraction that describes a sequence of packets that share five common key attributes: *source IP address*, *destination IP address*, *source port*, *destination port*, and *transport protocol*. Two scopes of network flows exist: unidirectional and bidirectional. Unidirectional flows only contain packets sent from the designated source to the designated destination, while bidirectional flows group packets in both directions. A network flow record embeds the statistical data related to the communication identified by the 5-tuple, such as the *Duration* of the communication or the *Number of Bytes* transmitted. Such extra features depend on the network flow format, and several competing formats exist. In the following, for brevity sake, "network flow record" will be abbreviated to "flow".

3.2 Minimum Description Length

The Minimum Description Length (MDL) principle [21] is a model selection criterion based on the idea of Occam's razor, i.e., that out of all possible descriptions, the shortest one is the best one. Formally, it is based on the Kolmogorov complexity [37]. For a given model class $\mathcal{M}$, MDL identifies the best model $M \in \mathcal{M}$ as the one that minimizes the number of bits needed to describe both model and data given the model, $L(D, M) = L(M) + L(D|M)$ where $L(M)$ is the length of model M in bits and $L(D|M)$ the length of data D given M. We use this score to identify the most relevant patterns. Remark that, in practice, we do not need to actually describe the encoding of the data: only the length of the encoded data is relevant. For example, one can describe any value among a set of k different values with (asymptotically) $\log_2(k)$ bits, and can describe a value v associated to a probability distribution P with $-\log_2(P(v))$ bits [21]. In the following, all logarithms are base 2 and we define $0 \log(0) = 0$.

3.3 Bayesian networks

Bayesian networks are a class of generative statistical models that represent probability distributions [56]. They are widely used in statistics due to their ability to be able to represent any probability distribution. Syntactically, they are described as a directed acyclic graph, where each node is a random variable, and a set of conditional probability tables. These conditional probability tables, one per node, describe the probability of their associated node depending on the values of its parents. Due to their structure, Bayesian networks are considered to be explainable models: the edges between nodes are indicative of their statistical correlations.

3.4 Notations

In the following, we denote F each feature of the network flow (such as source IP, protocol, etc.) and $\underline{F}$ its domain, i.e., the set of possible values for F. Let us denote n the number of features. A flow is simply a tuple of the n features: the domain of flows is therefore $\underline{F}_1 \times \underline{F}_2 \times \cdots \times \underline{F}_n$. We will

typically use the letter t to denote timestamps. Finally, a dataset D is a sequence of timestamped network flows (t, f).

4 Pattern Language of FlowChronicle

In this section, we formalize the language of patterns that are identified by *FlowChronicle*.

4.1 Intuition

Given a dataset D of network flows, we aim to identify patterns that can describe which combinations of flows occur frequently in D. Some patterns can concern values inside a flow. For example, destination port 53 is frequently associated with protocol UDP so, intuitively, we could use a pattern to automatically complete the protocol given the destination port. Some patterns can also concern several flows. For example, HTTP(S) requests are typically preceded by a DNS request. Similarly, an IMAP request (to read emails) can be followed by HTTP(S) requests if URLs of images are present in an email. For this reason, our patterns can span over multiple flows.

Classical pattern mining searches for deterministic relations, e.g., destination port 53 implies the UDP protocol. We consider they are not sufficient to properly encompass network flows dependencies. For this reason, we propose to also include in our pattern statistical relations. For example, if a machine has both an SSH and an HTTPS server, then when this machine is contacted, the destination port will probably be 22 or 443 but not any other port, e.g., 21.

4.2 Pattern language

A pattern is composed of two parts: the partial flows, to mine discrete temporal dependencies, and the dependency structure, to mine statistical temporal dependencies. Since a pattern can span across several flows and can specify the values of features for these flows, we need to store this information. For that purpose, patterns contain a table, called the partial flows, which columns are the network features and each row corresponds to a flow. We name *cell* each cell of such table.
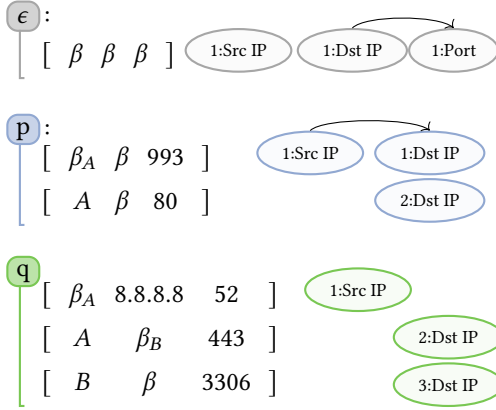
Each cell of the partial flow can be one of three types. Firstly, there are *fixed* cells: these are cells which values are directly defined in the partial flows. For example, the first partial flow could have destination port 53 and the second, destination port 443. Secondly, there are *free* cells: these are the cells not defined by the partial flows, and their values are determined by the dependency structure (described below). Lastly, there are *reuse* cells: the values of these cells are equal to the values of other cells in preceding partial flows of the pattern. A common illustration of this type of cell could be the Source IP address of a first flow that is reused as the Destination IP of a subsequent flow.

Because some cells can be free, each pattern also contains a *dependency structure*. The dependency structure is a Bayesian network that represents the joint probability distribution of the free cells.

More precisely, a pattern p is a tuple (X, BN), where X is a sequence of partial flows, and BN is the Bayesian network representing the dependency structure between free cells. We write $X[j]$ to denote the jth partial flow. Each cell of a partial flow can be either fixed (i.e., associated with a f of $\underline{F}$), free (denoted β, for "Bayesian") or reuse (denoted with an uppercase identifier). Besides, free cells can be marked for reuse. This is denoted by adding an uppercase identifier in subscript to β. For example, the value of β_A will be used for the reuse cell denoted A. If an identifier is defined in $X[j]$, then it can only be used in later partial flows, i.e., in $X[k]$ such that $k > j$. BN is a Bayesian network defined over all the free cells in partial flows, i.e. all $\beta \in p$.

Examples of patterns are shown in Figure 1. Three patterns are defined: ϵ has one partial flow with only free cells. Pattern p has two partial flows. The identifier A in the reuse cell is used to ensure that the source IP is the same in both flows. The ports are fixed while the IP addresses are free. Finally, pattern q has three partial flows. The identifiers A and B and the reuse cell ensure that

Model — Pattern and Bayesian Network:

Data and Pattern Windows:

ϵ :

$$\left[\begin{array}{ccc} \beta & \beta & \beta \end{array} \right]$$

1:Src IP   1:Dst IP   1:Port

p :

$$\left[\begin{array}{ccc} \beta_A & \beta & 993 \end{array} \right]$$
$$\left[\begin{array}{ccc} A & \beta & 80 \end{array} \right]$$

1:Src IP   1:Dst IP

2:Dst IP

q

$$\left[\begin{array}{ccc} \beta_A & 8.8.8.8 & 52 \end{array} \right]$$
$$\left[\begin{array}{ccc} A & \beta_B & 443 \end{array} \right]$$
$$\left[\begin{array}{ccc} B & \beta & 3306 \end{array} \right]$$

1:Src IP

2:Dst IP

3:Dst IP

Time	Src IP	Dst IP	Port
12	134.96.235.78	142.251.36.5	993
56	134.96.235.129	8.8.8.8	52
89	134.96.235.78	212.21.165.114	80
113	134.96.235.129	198.95.26.96	443
145	198.95.26.96	198.95.28.30	3306
156	134.96.235.78	134.96.234.5	21
178	134.96.235.36	185.15.59.224	993
206	134.96.235.36	128.93.162.83	80

Fig. 1. Toy example: On the left side we show a model with 3 patterns, on the right side we show the dataset and how the patterns of the model cover the dataset.

the source IP of the second flow is equal to the source IP of the first flow and that the source IP of the third flow is equal to the destination IP of the second flow.

4.3 Dataset cover

To select the best model able to capture the patterns in a dataset, we need to compute the term $L(D|M)$, as seen in Subsection 3.2. For this, we have to use the patterns to compress the data, i.e., *cover* the datasets with patterns and encode the locations of the patterns and the values of their non-fixed values. To properly define this cover, we first define a window of a pattern p, which indicates for each partial flow of p an index in the data, such that the partial flow matches the flow at that index in the data.

For example, in Figure 1, pattern p is associated to two windows, (12,89) and (178,206), and pattern q is associated to the window (56,113,145). Remark that the fixed values of the pattern always match the data. Besides, the reuse cells also match: for example, in the window (12,89) of pattern p, both source IP are indeed identical.

A *dataset cover* is a set of windows such that all flows of the dataset are associated to exactly one window. To ensure that this is always possible, we define a "catch-all pattern", denoted ϵ (see Fig. 1), that has only one partial flow and whose cells are all free. ϵ is called the *empty pattern*. Remark that multiple covers can explain the data for a given set of patterns, and that finding the optimal cover is a NP-hard problem [30]. For this reason, we use a greedy algorithm to estimate a cover.

4.4 Model encoding

Now that we have given the intuition, we formally describe our MDL encoding, which has two parts: model encoding and data encoding given a model. We start with the model encoding.

As a model M is a set of patterns, we need to encode the number of patterns and each pattern. Hence, we require $L(M)$ bits to encode M, with

$$L(M) = L_{\mathbb{N}}(|M|) + \sum_{p \in M} L(p) \quad ,$$

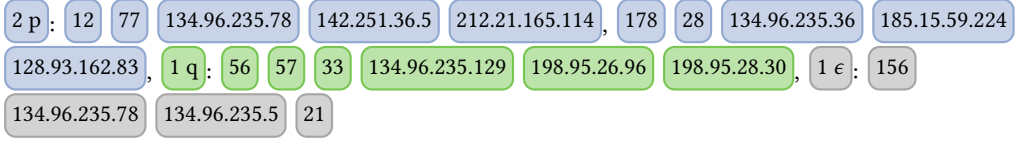Code Sequence — encoding of data with model:

| 2 p |: | 12 | | 77 | | 134.96.235.78 | | 142.251.36.5 | | 212.21.165.114 |, | 178 | | 28 | | 134.96.235.36 | | 185.15.59.224 |
128.93.162.83 |, | 1 q |: | 56 | | 57 | | 33 | | 134.96.235.129 | | 198.95.26.96 | | 198.95.28.30 |, | 1 ϵ |: | 156 |
134.96.235.78 | | 134.96.235.5 | | 21 |

Fig. 2. Encoding of the data shown in Fig. 1, i.e. how the data is described using the model.

where $|M|$ refers to the number of patterns in M. We encode $|M|$ using the MDL-optimal encoding for integers $z \geqslant 1$ [61]. It is defined as $L_{\mathbb{N}}(z) = \log^* z + \log c_0$ where $\log^* z$ is the expansion $\log z + \log \log z + \cdots$ where we only include the positive terms. To ensure this is valid encoding, i.e. one that satisfies the Kraft inequality, we set $c_0 = 2.865064$ [61].

To encode a pattern p, we first encode the number of partial flows a pattern contains, then all the partial flows, and finally the Bayesian network,

$$L(p) = L_{\mathbb{N}}(|p|) + \left(\sum_{j=1}^{|p|} L(X[j]|p) \right) + L(BN_p) \quad .$$

We encode the Bayesian network by encoding for each node its number of parents c, in $\log K$ bits, where K is the maximum number of parents passed as a parameter to the learning algorithm, and then select the parents out of all $|B| - 1$ possible parents, where B is the set of free cells described by the Bayesian network (formally $B = \{(j,i) \mid X[j]_i = \beta \lor X[j]_i = \beta_A\}$). So:

$$L(BN_p) = \sum_{(j,i) \in B} \log K + \log \binom{|B| - 1}{c_{j,i}} \quad .$$

We do not encode the conditional probability tables, as explained in the next subsection.

We split the partial flow encoding into three parts: we encode 1) the fixed cells, 2) which free cells that are marked for reuse, and 3) if there are values marked for reuse in earlier flows, where and if we want to use them. We will explain the encoding in turn.

We start by encoding for how many of the n flow features we want to encode a fixed value, with $\log n$ bits. To select which k cells, we require $\log \binom{n}{k}$ bits. Finally, we encode the respective values by choosing one value from the respective domain. To encode which cells we mark for reuse, we first encode how many, out of the $n - k$ remaining, and then, choose the l cells we want to mark. Finally, we encode for which cells we want to reuse values: we again encode how many of the remaining $n - k - l$ and choose which m cells. For each of the m selected cells, we select which of the previously marked cells we reuse. Formally this is,

$$L(X[j]|p) = \log(n) + \log \binom{n}{k} + \left(\sum_{i \in S_j} \log |\underline{F}_i| \right) + \log(n - k) + \log \binom{n - k}{l} +$$

$$\mathbb{1}(|\pi(j,p)| > 0) \left(\log(n - k - l) + \log \binom{n - k - l}{m} + m \log(|\pi(j,p)|) \right),$$

where S_j is the set of all features with a fixed cell, and $\pi(j|p)$ is a set of all cells marked for reuse before the jth flow.

4.5 Data encoding given a model

Now that we know how to encode a model M, we describe how to encode D using a model M. We define the encoding of D by M as the *Cover* of D. Its length in bytes is $L(D|M)$. Because M is a set of patterns, the cover is a set of pattern windows. In Figure 1, we show a toy example of a dataset, a model and its cover. Before we define how we encode a dataset D, using a set of patterns, let us give the intuition by describing how we decode a dataset from a given cover. More precisely, a cover is a sequence of codes, encoding how often each pattern occurs, where these occurrences are, and the values of the free cells.

In Figure 2, we show the sequence of codes corresponding to the toy example in Figure 1. To decode the cover, we start by reading the first code from the cover, in our toy example $\boxed{2\ \text{p}}$, indicating that we use pattern 'p' twice in the cover. Next, we read for each partial flow in the pattern the codes corresponding to the timestamps $\boxed{12}$ and $\boxed{77}$, the first one being the start time, and the second one the delay to the first partial flow. Next, we read for each free cell one code—decoding the values. We repeat these steps for the second occurrence of pattern 'p'. We do the same for pattern 'q'. Finally, code $\boxed{1\ \epsilon}$, tells us there is one flow is covered by the empty pattern. We read again the timestamp code as well as the encoded values. With that, we have fully decoded the cover.

Now that we have seen how data encoding works, we will formally describe how many bits we need to encode it. For each p, we encode how often we want to use it, i.e., the number of windows in the cover. We then encode each window. Formally this is:

$$L(D \mid M) = \sum_{p \in M} \left(L_{\mathbb{N}}(|W_p|) + L(W_p) \right) \quad ,$$

where W_p denotes the set of windows of pattern p used to describe D. The length of W_p is the timestamps plus the free cells, encoded based on the probabilities given by the Bayesian network,

$$L(W_p) = \sum_{i=1}^{|W_p|} \left(L(t_1 \text{ of } w_i) + \sum_{k=2}^{|p|} L(t_k \text{ of } w_i \mid t_{i-1}) \right) - \log(Pr(w_i|BN_p, \{w_j|j < i\}))$$

where $L(t) = \log(t_{\max} - t_{\min})$ and $t_{\max}$ (resp., $t_{\min}$) refers to the maximum (resp., lowest) timestamp in the data D, $L(t_i \mid t_j) = L_{\mathbb{N}}(t_i - t_j)$. As we expect lower delays between flows, we chose $L_{\mathbb{N}}$ to encode the difference between time points. It closely follows a geometric distribution, hence giving higher probability mass to lower delays, and thereby requiring fewer bits for those.

To avoid having to encode the contingency table of the BN and make arbitrary encoding choices in the process, we use prequential codes [21]. The basic idea is to start with a uniform distribution and update the probabilities after each encoding, thereby always maintaining a valid probability distribution. More precisely, after each encoding, the probability distribution is recomputed given the sets of all encoded vectors using a Laplace smoothing with $\lambda = \frac{1}{2}$ [29].

5 Algorithm

In this section, we present the whole generation pipeline: data preprocessing, pattern identification—the previous section describes the MDL loss used to choose a model but does not explain how to find the candidate model—and data sampling from the selected model.

5.1 Preprocessing a network flow dataset

Network flow data are tabular data, where each network flow is a line in the table, and each feature is a column. The features are either categorical (e.g. *Transport Protocol*), or continuous (e.g. *Duration of the flow*). To mine patterns in that tabular dataset, we first need to discretize the numerical

features in our network flow description. Similar to Schoen et al. [63], we discretize the numerical features into 40 categories, such that each category contains the same number of samples.

5.2 Pattern Miner

In this section, we explain how to discover a good model and a description of the data under a given model M. We begin with the latter.

5.2.1 Finding a cover. Given a model M, we want the cover C that minimizes the encoding cost $L(D \mid M)$. Finding the optimal cover is a NP-hard problem, so we propose a greedy method. For this, we have to find out where we can use a pattern p, i.e., we have to find the windows of p. We only consider minimal windows, i.e., windows for which there does not exist a window $\overline{w}(p)$ whose interval $I(\overline{w}(p))$ is a proper sub-interval of $I(w(p))$. We sort all windows by 1) the number of covered flows (decreasing), and 2) inter-flow delays (increasing). Note the empty pattern ϵ is defined to cover 0 flows, i.e., it should only be used to cover flows that are not covered by any other pattern. Finally, we greedily add windows to the cover until all flows are covered. If one window overlaps with precedent windows, we skip it. With that, we have a description of D in terms of pattern.

5.2.2 Iterative Pattern Search. The general approach is an iterative search procedure: at each iteration, we generate pattern candidates and test if these candidates help in reducing the description length. If so, we add them to our model. At each step of the search, we ensure no source or destination IP has a fixed value. Indeed, we do not want to learn the behavior of specific IP addresses. Besides, we restrict reuse cells and cells marked for reuse to only be source IP or destination IP.

Candidate Generation. The initial set of candidates is created as follows: for each couple of features, and for each combination of values of these two features (except the source IP and destination IP), we create a pattern with a single flow with two fixed cells. The rest of the cells are free and described by a Bayesian network.

During the search, we build new candidates by extending existing patterns. Given a pattern, we have three different ways to generate new candidates: 1) by directly creating a fixed cell, either from a previously free cell or by adding a new row of free cells and transforming one into a fixed cell—once again, this fixed cell cannot be a source IP or destination IP; 2) by merging existing patterns. If both patterns have only one partial flow and have no conflicting fixed cells, we merge them into a new single-flow pattern. For patterns over multiple flows, we create candidates by appending them; 3) by transforming a free cell into a reuse cell. Such cells can only appear in multi-flow patterns because a reuse cell can only reference a marked cell from previous partial flows. This reuse cell can reference previously marked cells or mark new previous cells to reference them.

Algorithm 1: *FlowChronicle*

Input : set of flow D,
            continous misses threshold t
Output: model M and Cover of D

1 $M \leftarrow \{\epsilon\}$
2 $C \leftarrow$ all pairwise combinations
3 mode $\leftarrow$ *single-flow*, misses$\leftarrow$ 0
4 **while** misses$< t$ or mode $=$ *single-flow* **do**
5      $C \leftarrow C \cup$ BuildCandidates$(M,$ mode$)$
6      $c \leftarrow \arg\max_{c \in C} cs(c)$
7   **if** $L(D, M) > L(D, M \cup c)$ **:**
8          $M \leftarrow$ Prune$(M \cup c)$
9   misses$\leftarrow$ 0
10   **else**
11   misses$\leftarrow$ misses$+1$
12   **if** misses $> t$ and mode $=$ *single-flow* **:**
13   mode $\leftarrow$ *multi-flow*
14   misses $\leftarrow$ 0
15 **return** M, C

Candidate score. As testing all candidates is not feasible in a reasonable time, we want to test the most promising candidates first. To this end, we derive a candidate score. The candidate scores capture how many values a pattern can cover: the number of non-overlapping windows multiplied by the number of fixed or reuse cells in the pattern.

Mining a Model. We show the pseudocode of *FlowChronicle* in Algorithm 1. We initialize our model with the empty pattern. We start our search with the initial set of patterns. The basic idea is to take the best candidate c according to the candidate score $cs(c)$, and if it reduces $L(D, M)$, we add it to the model. If a pattern fails to reduce $L(D, M)$, we will not test it again in future iterations[1].

As testing all candidates is not feasible, we propose an early stopping criteria. We propose to stop when we exceed a *consecutive misses* threshold t. This threshold is defined by the user.

To avoid building uninformative patterns spanning many rows, we begin by searching for patterns within single flows. In practice, we do that by only generating single-flow candidates. We continue this until we surpass the *consecutive misses* threshold; at this point, we reset the *consecutive misses* threshold and also allow the construction of candidates that cover multiple flows.

Adding a new pattern to M can make existing patterns redundant. We hence prune redundant patterns by testing for all patterns p where the usage in the cover has been reduced: if $L(D, M \setminus \{p\}) < L(D, M)$, we remove it from the model M.

5.3 Parallelization

To improve run-time on larger data sets, we propose to split the preprocessed data into n chunks. We learn independently a model for each chunk, so each chunk can be processed in parallel. The learned models then capture local characteristics and provide a cover of the respective chunk. Since each model is a set of patterns and the corresponding BN, we can simply take the union of all models resulting in a new model for the entire dataset. The cover of the entire dataset can be constructed by appending all individual covers. Finally, we relearn the BN of the empty pattern (the pattern used to cover all flows not covered by any other pattern).

5.4 Synthetic data generation

Once we have a set of patterns and the cover, we can use them to generate a synthetic dataset. From the cover, we can learn the probability distribution over the usage of each pattern (including the empty pattern) and sample from this distribution. Next, we sample the initial timestamps for each pattern. As some patterns might occur more frequently during some periods (e.g., fewer emails during lunch, more OS updates in the morning, etc.), and to not make any assumption about the shape of the distribution, we estimate the frequency over time via a *Kernel Density Estimation* (KDE). We then sample the timestamps from this distribution. For patterns with multiple partial flows, we estimate a distribution of the delay between consecutive partial flows, again with KDE, and sample from this distribution. Finally, we have to fill the cells of all sampled pattern occurrences. Cells with a *fixed* value are already set. For the free cells, we sample from the Bayesian network BN associated with the pattern. Finally, we set the values of *reuse* cells, and this completes the generated flow dataset.

6 Evaluation method

Our goal in this part is to provide an evaluation framework to compare the generation of the different models. This evaluation will be twofold: we will first evaluate the different generated flows independently, without any consideration for any temporal dependencies, and second, we will study how well the generation preserves temporal dependencies present in real data.

---

[1]For better readability, we omitted this part from the pseudocode.

6.1 Independent evaluation

Independent evaluation involves analyzing the network flow distribution generated by a model and comparing it with the training data to determine if the model has captured the essential characteristics needed to create new data. For each evaluated model, we compare its synthetic network flow distribution with the real network flows from the *week-3* dataset (the training dataset). As described by Schoen et al. [62, 63], this comparison should elucidate four key attributes of the generated data: **Realism**, **Diversity**, **Compliance**, and **Novelty**. Realism ensures that a generated network flow should be sampled from the same distribution as the real network flows. Diversity ensures that the generated network flows should cover the entire real network flow distributions. Compliance refers to the criterion that checks if the generated network flow adheres to specific network rules. Lastly, Novelty ensures that the generated network flows are not mere replicas of the real network flows.

6.1.1 Realism. Similarly to [63], we evaluate the realism of the joint distribution with the Density metric [46], the realism of the conditional probability distributions with CMD (Contingency Matrix Difference) and PCD (Pairwise Correlation Difference). CMD is the difference between the generated data's correlation matrix and the training data's correlation matrix (for numerical features). PCD is the difference between the generated data's contingency matrix and the training data's contingency matrix (for categorical features).

6.1.2 Diversity. Similarly to [63], we evaluate the diversity of the joint distribution with the Coverage metric [46]. A low score indicates that the generated distribution does not cover the entire training distribution. We also evaluate the marginal distributions of each features. with the JSD (Jensen Shannon Divergence) for categorical features and the EMD (Earth Mover's Distance) for numerical features.

6.1.3 Compliance. Compliance is the property that the generated network flow should respect network protocol specifications. For example, a generated UDP flow should not contain any TCP flags. We evaluate this property with the DKC (Domain Knowledge Check) [59], a succession of boolean tests for the generated network flow, each test representing one property that we want to enforce in the generation. We use the implementation proposed by [63]. A lower DKC means fewer tests have failed and a more compliant generation.

6.1.4 Novelty. We evaluate the novelty similarly to [63] with he Membership Disclosure (MD) metric. This metric evaluates the privacy risk of synthetic datasets generated by models trained on real datasets. It involves comparing the synthetic samples to the training and testing sets from the original data by computing the Hamming distances between each pair of generated and real samples. When a synthetic sample is sufficiently similar to a real sample (i.e., the Hamming distance is below a certain threshold), the real sample is considered a potential leak from the training set. By varying the threshold, a detection method for training samples is established, and the effectiveness of this detector is measured using the F1-score. The overall privacy risk is quantified by integrating the F1-scores over all possible threshold values. In a network context, very similar flows like DNS or NTP requests are not uncommon, so the level of novelty in synthetic data should mirror that of a reference set of real data.

6.2 Preservation of temporal correlation

The above metrics only account for individual network flows, but we seek to generate data that preserves temporal dependencies, so we also need to evaluate this aspect. We propose to use feature-wise metrics to assert whether a generated dataset preserves the temporal dependencies

present in the training set. We will consider the numerical features on one hand and the categorical features on the other.

6.2.1 Numerical features. Inspired by several papers [42, 48, 51, 67], we evaluate the temporal dependencies between a generated dataset and a training dataset by comparing the Autocorrelation Functions (ACF) of every numerical features. The ACF of a numerical feature is the (linear) autocorrelation between the value of the feature at a timestamp t and its value at a later timestamp $t+l$, where l is the lag.

However, since not all lags exhibit strong autocorrelation, calculating the difference between ACFs across all lags may smooth out the differences for those lags that do reveal significant temporal dependencies. To address this, we discard any lags whose autocorrelation in the training data does not exceed the Bartlett confidence interval [5]. This ensures that we compute the ACF difference only for the lags that demonstrate strong temporal dependencies. This approach allows us to verify whether a temporal dependency present at a certain lag in a training feature is accurately reproduced at the same lag in the generated feature, and we apply this procedure across all numerical features of the training dataset.

6.2.2 Categorical features. For categorical features, we decided to implement a TSTR (Train on Synthetic, Test on Real) method [79]. It is commonly reused when it comes to evaluate the preservation of temporal dependencies [11, 48, 67]. This method compares the performance of a model in a machine learning task when it is trained on the training dataset and when it is trained on the generated dataset. To apply this methodology and highlight how a model preserves temporal dependencies, we compare the performance of an LSTM (a temporal deep learning model) when it is trained on training data versus when it is trained on generated data for a feature-wise autoregressive task.

For one categorical feature, we first encode its values in a one-hot encoded vector. Then, we train an LSTM to predict the next one-hot encoded value of that feature given the previous values in a context window. This first training is done on the training dataset. Afterward, we create another LSTM with the same hyperparameters (same number of hidden dimensions, same context size, etc.) and we train it on the generated dataset. We compute the accuracy of the two LSTMs on the evaluation set, and the final TSTR score is the difference between the two values. We do this process (illustrated in Figure 3) for every categorical feature. In practice, because we do not want our score to rely too heavily on one configuration of the LSTM, we repeat the operation multiple times while varying its hyperparameters. The score for each categorical feature will be the average of the differences in accuracy for every LSTM.
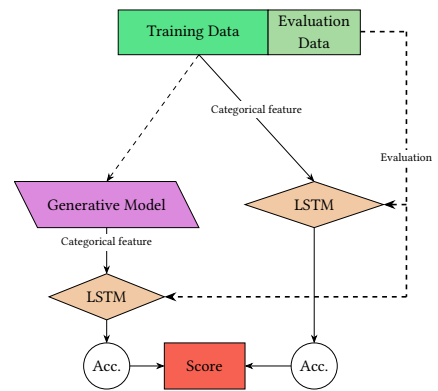
Fig. 3. TSTR methodology: two similar LSTMs are trained, one on the training data, and the other on the generated data. Their accuracy are then compared on evaluation data. This is repeated for each categorical feature.

7 Experiments

In this section, we would like to verify whether *FlowChronicle* is able to generate network flows with a higher quality than other model-based generation methods. The implementation of *FlowChronicle*, as well as the experimental setup, are available online as open source software[2].

---

[2]https://github.com/joschac/FlowChronicleCoNEXT

7.1 Competing methods

We compare *FlowChronicle* with **CTGAN** [74], **TVAE** [74], **E-WGAN-GP** [59] and **NetShare** [77]. **NetShare** does model temporal dependencies, so we can compare our model against another method with temporal dependencies. We were also interested in adding STAN to our benchmark, but despite our best efforts, we were unable to reproduce the work of Xu et al. [75] from the author's repository. We therefore omit it from comparison. We also compare our method to the Bayesian Network proposed by Schoen et al. [63]: we call it **IndependentBN**. We also propose a variation that generates a sequence of five network flows instead of generating every network flow independently: we call it **SequenceBN**. Recently **Tranformer**-based methods have been shown to be great synthetic data generators. As a representative, we compare against the GPT2 model [58]. We tokenize the data as we do for our method, and use a context window of 60 tokens.

7.2 Experimental protocol

We evaluate the methods on the CIDDS-001 dataset [60]. It is a simulated dataset of 4 weeks of traffic from 30 terminals (5 servers, 3 printers, 4 Windows clients, and 15 Linux clients). Because we focus on generating benign traffic, we only kept the data recorded in the OpenStack environment.

We use *week-3* as a training set, and *week-4* as a held-out **Reference** set. The creation of *week-4* followed the same process as *week-3*, and thus, we consider this Reference set as the best possible synthetic generation. Because some parts of our evaluation methodology also require another evaluation subset (see paragraph 6.2.2), we consider the benign traffic of *week-2* as an evaluation set. We process the original unidirectional flows into bidirectional flows. We consider the 11 flow features shown in Table 1. We did not include the

Feature	Description of the feature
Date first seen	Timestamp of the first packet of the flow
Proto	Transport protocol
Src IP Addr	Source IP Address (Client)
Dst IP Addr	Destination IP Address (Server)
Dst Pt	Destination Port
In Byte	Number of Bytes coming to the client
In Packet	Number of Packets coming to the client
Out Byte	Number of Bytes sending from the client
Out Packet	Number of Packets sending from the client
Flags	Type of flags contained in the flow
Duration	Duration of the flow

Table 1. Set of Features in our dataset

Source Port because it is generally randomly sampled in a particular range (cf. RFC6056 [35]). In the original dataset, the external public IP addresses were anonymized [60]. The value generated by our method will, therefore, be anonymized too.

7.3 Time-independent Evaluation

We begin by evaluating the flows independently, without considering temporal dependencies between flows. The different metrics were computed 20 times on 20 different subsamples of both the generated and training data. Each subsample includes 10000 flows. The average value of each metric as well as the ranking of all our models according to each of them are reported in Table 2. With this global benchmark, we can see that *FlowChronicle* is on average above the other model-based methods, with CTGAN being a close second.

7.3.1 Realism. We see that Transformer and CTGAN achieve a pretty high Density (0.62 and 0.56 respectively). However, Transformer seems unable to represent cross-feature correlation as illustrated by its CMD and PCD (0.78 and 3.62, respectively). Moreover, E-WGAN-GP seems unable to create realistic data (0.02 of Density, 0.34 of CMD and 3.63 of PCD). This might be because our dataset is bidirectional, whereas the encoding IP2Vec was originally intended for unidirectional datasets. *FlowChronicle* creates above-average data regarding the Realism of its synthetic network flows.

	Density	CMD	PCD	EMD	JSD	Coverage	DKC	MD		Rank
	Real.	*Real.*	*Real.*	*Real./Div.*	*Real./Div.*	*Div.*	*Comp.*	*Nov.*		*Average*
	↑	↓	↓	↓	↓	↑	↓	=		*Ranking*
Reference	(0.69)	(0.06)	(1.38)	(0.00)	(0.15)	(0.59)	(0.00)	(6.71)		-
IndependentBN	7 (0.24)	5 (0.22)	6 (2.74)	8 (0.11)	4 (0.27)	4 (0.38)	4 (0.05)	4 (5.47)		5.25
SequenceBN	6 (0.30)	2 (0.13)	5 (2.18)	7 (0.08)	3 (0.21)	3 (0.44)	2 (0.02)	3 (5.51)		3.875
TVAE	3 (0.49)	4 (0.18)	3 (1.84)	2 (0.01)	5 (0.30)	5 (0.33)	6 (0.07)	5 (5.17)		4.125
CTGAN	2 (0.56)	3 (0.15)	2 (1.60)	3 (0.01)	2 (0.15)	2 (0.51)	8 (0.11)	2 (5.70)		3.0
E-WGAN-GP	8 (0.02)	7 (0.34)	8 (3.63)	5 (0.02)	7 (0.38)	8 (0.02)	7 (0.07)	6 (4.66)		7.0
NetShare	5 (0.32)	6 (0.28)	**1 (1.47)**	6 (0.03)	6 (0.36)	6 (0.22)	5 (0.05)	7 (3.82)		5.25
Transformer	**1 (0.62)**	8 (0.78)	7 (3.62)	**1 (0.00)**	8 (0.55)	7 (0.03)	3 (0.05)	8 (3.75)		5.375
FlowChronicle	4 (0.41)	**1 (0.03)**	4 (2.06)	4 (0.02)	**1 (0.10)**	**1 (0.59)**	**1 (0.02)**	**1 (5.87)**		**2.125**

Table 2. Ranking of our different models without considering the preservation of temporal dependencies. For each metric, the average of the score is given between parentheses. Real.: Realism, Div.: Diversity, Comp.: Compliance, Nov.: Novelty, ↓: Lower is better, ↑: Higher is better. =: closer to Reference is better.

7.3.2 Diversity. While the Transformer produce a rather realistic result, it fails to produce diverse results: it has a low Coverage (0.03) and a high JSD (0.55). This is because the Transformer model fell into a well-known behavior of autoregressive generative models during training called *degeneration* [16, 78]. This phenomenon consists of the model learning to generate one specific sequence of network flow and keep repeating it during the generation process. We also see the difficulty for Bayesian Networks to work with numerical variables (EMD of 0.08 and 0.011 for SequenceBN and IndependentBNs, respectively) – a phenomenon already highlighted by Schoen et al. [63]. *FlowChronicle* and CTGAN are among the best models for covering the entire training distribution.

7.3.3 Compliance. Apart from CTGAN (DKC of 0.11), the models are able to generate traffic that is compliant with our set of rules. *FlowChronicle* produces data with the least compliance issues.

7.3.4 Novelty. With its MD of 5.87, *FlowChronicle* is closest to the reference data set (6.71), denoting its ability to generate fresh data. On the other hand, Transformer and NetShare introduce too little novelty in the synthetic data.

7.4 Preservation of temporal correlation

The previous evaluation did not consider preserving temporal dependencies between the flows during the generation. This is the goal of this subsection. Overall, *FlowChronicle* preserves temporal dependencies in both categorical and numerical features, making it closest to the reference.

7.4.1 Numerical features: Difference of autocorrelation functions (ACF). In Figure 4, we have represented the differences of ACF across all numerical features between the real training dataset and the generated dataset. CTGAN and TVAE are the worst models for preserving temporal dependencies in numerical features. Both models come from the same library [55] and do not take into account temporal dependencies. Both Transformer and NetShare preserve the temporal dependencies well since these models are designed to preserve such dependencies. More surprisingly, E-WGAN-GP, which samples network flow independently has also a low score. *FlowChronicle* is better than those methods and reproduces well the different autocorrelation across the different numerical features. The differences in ACFs between the set generated by *FlowChronicle* and the Reference set are almost equivalent.

7.4.2 Categorical features: Impact of generated sequences on the accuracy of an LSTM. In Figure 5, we see the difference in accuracy between two similar LSTMs trained on the training data and on synthetic data generated by every model, and this, for every categorical feature in the dataset.
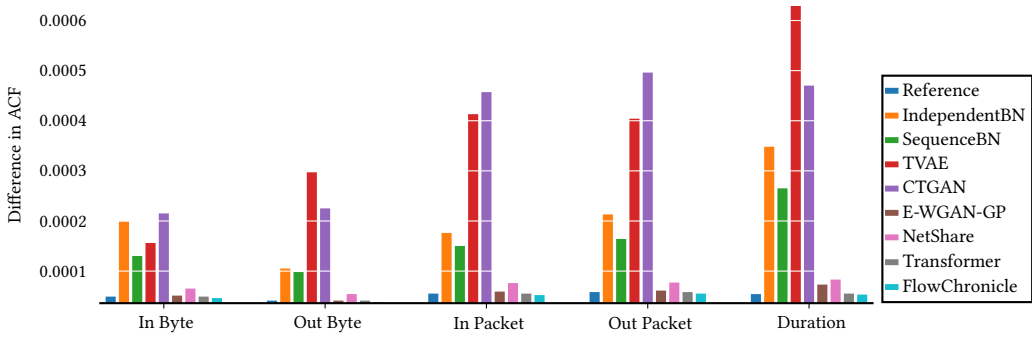
Fig. 4. Difference of ACF between the generated data and the train data across all the numerical features for our different generative methods. Lower is better.

FlowChronicle is the best among the other generative models for preserving temporal dependencies across categorical features, with a score once again close to the Reference set.
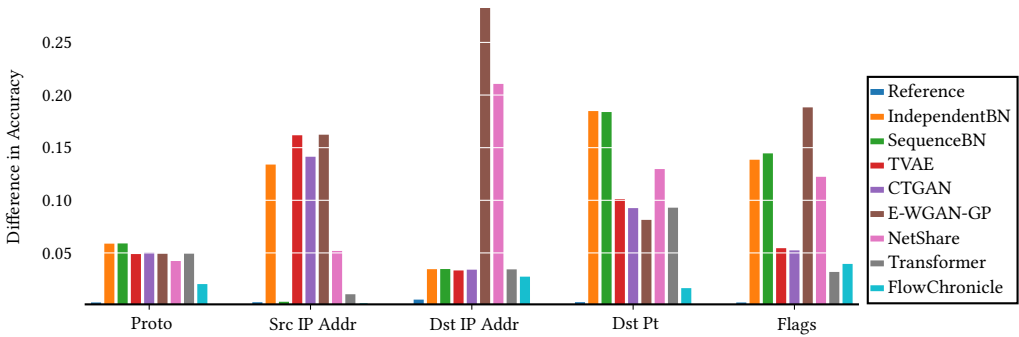

Fig. 5. Average difference of accuracy of various LSTMs trained on the train data and our generated data from our different generative models. Each subgroup is one feature, and each bar is one generation method. Lower is better.

7.5 Computational Cost

Comparing computing costs can be useful for choosing the right generation method. In Table 3, we report the time taken for training each method and generating synthetic data. All our experiments have been carried out on a server with 500 GB of RAM, 2 AMD EPYC 7413 CPUs, and 3 A40 Nvidia GPUs.

One drawback of *FlowChronicle* is the time required to train and generate new data. Even if *FlowChronicle* obtained on average the best performances on independent and temporal metrics, it is also the longest to train and produce new data. While a long training time is a known issue of MDL-based methods, we are confident the generation time could be largely lowered due to the simplicity of the process.

| | Duration (hh:mm) | |
Model	*Training*	*Generating*
IndependentBN	00:12	<0:01
SequenceBN	00:31	<0:01
CTGAN*	29:12	00:02
TVAE*	02:01	00:03
E-WGAN-GP*	00:36	01:59
NetShare*	59:39	05:00
Transformer*	84:02	34:41
FlowChronicle	106:54	85:16

Table 3. Training and generating runtimes. Methods annotated with * rely on GPU.

7.6 Explainable patterns

Besides a good-quality generation, *FlowChronicle* has the advantage of learning explainable patterns. In this section, we present a few interesting multi-flow patterns.

The first one contains three partial flows, from the same source IP and the same destination IP. All three partial flows are HTTPS requests (TCP protocol, destination port 443). When a browser requests a webpage, there can be many resources that it needs to download from the same server (images, scripts, styles, etc.) that can be fetched with a different HTTPS connection.

A second pattern is a DNS request (UDP protocol, destination port 53) followed by an HTTP flow (TCP protocol, destination port 80). The source IP is the same. This is a classical network pattern: before a device can access a domain for an HTTP request, it must know its IP. It could be cached locally but sometimes requires a DNS request to obtain it.

A third pattern contains two partial flows, from the same source IP but to different destination IPs. The first partial flow is an HTTPS request (TCP protocol, destination port 443) and the following one is a DNS request (UDP protocol, destination port 53). It can be explained by the fact that there can be some resources on a web page that are stored on other web servers (scripts or images). In that case, the browser needs to ask for the IP address of the server that stores those resources. This pattern is probably missing some later HTTPS connections. We did not find multi-flow patterns related to non-Web protocols though. We consider that such patterns explanation strongly indicates that *FlowChronicle* is indeed capable of learning relevant patterns that can be verified and explained by experts.

8 Conclusion

In this article, we consider the issue of generating synthetic network traffic, with a focus on preserving the temporal dependencies within the generated data. To achieve this goal, we introduced an innovative data-generating approach, dubbed *FlowChronicle*, which is based on pattern set mining. Initially, *FlowChronicle* learns a set of patterns that effectively encapsulate the data distribution and describe the data within the model. We formalize the problem with the Minimum Description Length (MDL) principle, by which our method is naturally robust against overfitting.

During the evaluation phase, we observed that *FlowChronicle* not only upholds the diversity and realism of the data but also maintains the temporal dependencies among flows. Even without taking into account any temporal dependencies, the generation through pattern mining allows us to reproduce network flows that are really close to the training data. In the non-temporal evaluation, the second-best method was CTGAN, but it struggled to capture temporal dependencies. Conversely, the Transformer model preserved temporal dependencies well but failed to generate high-quality individual flows. *FlowChronicle*, however, consistently ranked highest in both evaluations, excelling in both flow quality and temporal dependency preservation.

Finally, contrary to other methods, *FlowChronicle* outputs patterns that can be manually analyzed. This way, the generation method can be audited, and possibly manually verified and corrected. It is also easy to manually include new pattern to modify the generation without a relearning procedure.

Looking ahead, we identify two primary avenues for improvement. Firstly, a more powerful pattern language, Although our language is already robust, there are certain concepts, such as repeating flows as observed in video streaming, that we cannot represent effectively yet. A more expressive language will come with additional challenges, such as the increased search space. Secondly, our current approach employs a greedy search due to the necessity of testing numerous combinations. A differential approach could potentially expedite the learning process significantly, but how to do so is still an open research question.

Acknowledgments

References

[1] S. Abt and H. Baier. 2014. A Plea for Utilising Synthetic Data When Performing Machine Learning Based Cyber-Security Experiments. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop* (Scottsdale, Arizona, USA) *(AISec '14)*. Association for Computing Machinery, New York, NY, USA, 37–45. https://doi.org/10.1145/2666652.2666663

[2] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. IEEE Computer Society, Los Alamitos, CA, USA, 3–14.

[3] Tertsegha J. Anande, Sami Al-Saadi, and Mark S. Leeson. 2023. Generative adversarial networks for network traffic feature generation. *International Journal of Computers and Applications* 45, 4 (2023), 297–305. https://doi.org/10.1080/1206212X.2023.2191072 arXiv:https://doi.org/10.1080/1206212X.2023.2191072

[4] Tertsegha J Anande and Mark S Leeson. 2022. Generative adversarial networks (gans): a survey of network traffic generation. *International Journal of Machine Learning and Computing* 12, 6 (2022), 333–343.

[5] M. S. Bartlett. 1946. On the Theoretical Specification and Sampling Properties of Autocorrelated Time-Series. *Supplement to the Journal of the Royal Statistical Society* 8, 1 (1946), 27–41. http://www.jstor.org/stable/2983611

[6] S. Bourou, A. El Saer, T.-H. Velivassaki, A. Voulkidis, and T. Zahariadis. 2021. A review of tabular data synthesis using GANs on an IDS dataset. *Information* 12, 09 (2021), 375.

[7] Daniela Brauckhoff, Xenofontas Dimitropoulos, Arno Wagner, and Kavé Salamatian. 2012. Anomaly Extraction in Backbone Networks Using Association Rules. *IEEE/ACM Transactions on Networking* 20, 6 (Dec. 2012), 1788–1799. https://doi.org/10.1109/TNET.2012.2187306

[8] A. Cheng. 2019. PAC-GAN: Packet Generation of Network Traffic using Generative Adversarial Networks. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 0728–0734.

[9] Joscha Cüppers, Paul Krieger, and Jilles Vreeken. 2024. Discovering Sequential Patterns with Predictable Inter-event Delays. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 8346–8353.

[10] Joscha Cüppers and Jilles Vreeken. 2023. Below the Surface: Summarizing Event Sequences with Generalized Sequential Patterns. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 348–357.

[11] DataCebo, Inc. 2023. *Synthetic Data Metrics*. DataCebo, Inc. https://docs.sdv.dev/sdmetrics/ Version 0.12.0.

[12] B. Dowoo, Y. Jung, and C. Choi. 2019. PcapGAN: Packet Capture File Generator by Style-Based Generative Adversarial Networks. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. 1149–1154. https://doi.org/10.1109/ICMLA.2019.00191

[13] Jonas Fischer and Jilles Vreeken. 2019. Sets of robust rules, and how to find them. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 38–54.

[14] Jonas Fischer and Jilles Vreeken. 2020. Discovering succinct pattern sets expressing co-occurrence and mutual exclusivity. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 813–823.

[15] Esther Galbrun, Peggy Cellier, Nikolaj Tatti, Alexandre Termier, and Bruno Crémilleux. 2018. Mining Periodic Patterns with a MDL Criterion. In *ECMLPKDD18*. Springer, 535–551.

[16] Jun Gao, Di He, Xu Tan, Tao Qin, Liwei Wang, and Tie-Yan Liu. 2019. Representation Degeneration Problem in Training Natural Language Generation Models. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=SkEYojRqtm

[17] Aceto Giuseppe, Fabio Giampaolo, Ciro Guida, Stefano Izzo, Antonio Pescape, Francesco Piccialli, and Edoardo Prezioso. 2023. Synthetic and Privacy-Preserving Traffic Trace Generation using Generative AI Models for Training Network Intrusion Detection Systems. *Available at SSRN 4643250* (2023).

[18] Eduard Glatz, Stelios Mavromatidis, Bernhard Ager, and Xenofontas Dimitropoulos. 2014. Visualizing Big Network Traffic Data Using Frequent Pattern Mining and Hypergraphs. *Computing* 96, 1 (Jan. 2014), 27–38. https://doi.org/10.1007/s00607-013-0282-8

[19] K. Golnabi, R.K. Min, L. Khan, and E. Al-Shaer. 2006. Analysis of Firewall Policy Rules Using Data Mining Techniques. In *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. 305–315. https://doi.org/10.1109/NOMS.2006.1687561

[20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[21] Peter Grünwald. 2007. *The Minimum Description Length Principle*. MIT Press.

[22] Jorge Luis Guerra, Carlos Catania, and Eduardo Veas. 2022. Datasets are not enough: Challenges in labeling network traffic. *Computers & Security* 120 (2022), 102810. https://doi.org/10.1016/j.cose.2022.102810

[23] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. 2017. Improved Training of Wasserstein GANs. *CoRR* abs/1704.00028 (2017). arXiv:1704.00028 http://arxiv.org/abs/1704.00028

[24] Arash Habibi Lashkari., Gerard Draper Gil., Mohammad Saiful Islam Mamun., and Ali A. Ghorbani. 2017. Characterization of Tor Traffic using Time based Features. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - ICISSP*. INSTICC, SciTePress, 253–262. https://doi.org/10.5220/0006105602530262

[25] L. Han, Y. Sheng, and X. Zeng. 2019. A Packet-Length-Adjustable Attention Model Based on Bytes Embedding Using Flow-WGAN for Smart Cybersecurity. *IEEE Access* 7 (2019), 82913–82926. https://doi.org/10.1109/ACCESS.2019.2924492

[26] S. Hui, H. Wang, Z. Wang, X. Yang, Z. Liu, D. Jin, and Y. Li. 2022. Knowledge Enhanced GAN for IoT Traffic Generation. In *Proceedings of the ACM Web Conference 2022* (Virtual Event, Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA, 3336–3346. https://doi.org/10.1145/3485447.3511976

[27] A R Jakhale and G A Patil. 2014. Anomaly Detection System by Mining Frequent Pattern Using Data Mining Algorithm from Network Flow. *International Journal of Engineering Research* 3, 1 (2014).

[28] Steedman Jenkins, Stefan Walzer-Goldfeld, and Matteo Riondato. 2022. SPEck: Mining Statistically-Significant Sequential Patterns Efficiently with Exact Sampling. *Data Min Knowl Disc* 36, 4 (July 2022), 1575–1599. https://doi.org/10.1007/s10618-022-00848-x

[29] Zhiwei Ji, Qibiao Xia, and Guanmin Meng. 2015. A Review of Parameter Learning Methods in Bayesian Network. In *Advanced Intelligent Computing Theories and Applications*, De-Shuang Huang and Kyungsook Han (Eds.). Springer International Publishing, Cham, 3–12.

[30] Richard M Karp. 2010. *Reducibility among combinatorial problems*. Springer.

[31] A. Kenyon, L. Deka, and D. Elizondo. 2020. Are public intrusion datasets fit for purpose characterising the state of the art in intrusion event datasets. *Computers & Security* 99 (2020), 102022. https://doi.org/10.1016/j.cose.2020.102022

[32] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[33] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. 2015. Summarizing and understanding large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 8, 3 (2015), 183–202.

[34] M. Lanvin, P.-F. Gimenez, Y. Han, F. Majorczyk, L. Mé, and E. Totel. 2023. Errors in the CICIDS2017 Dataset and the Significant Differences in Detection Performances It Makes. In *Risks and Security of Internet and Systems*, Slim Kallel, Mohamed Jmaiel, Mohammad Zulkernine, Ahmed Hadj Kacem, Frédéric Cuppens, and Nora Cuppens (Eds.). Springer Nature Switzerland, Cham, 18–33.

[35] M. Larsen and F. Gont. 2011. Recommendations for Transport-Protocol Port Randomization. RFC 6056. https://doi.org/10.17487/RFC6056

[36] Srivatsan Laxman, P. S. Sastry, and K. P. Unnikrishnan. 2007. A fast algorithm for finding frequent episodes in event streams. 410–419. https://doi.org/10.1145/1281192.1281238

[37] Paul Li and Ming Vitányi. 1997. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer.

[38] Xin Li and Zhi-Hong Deng. 2010. Mining Frequent Patterns from Network Flows for Monitoring Network. *Expert Systems with Applications* 37, 12 (Dec. 2010), 8850–8860. https://doi.org/10.1016/j.eswa.2010.06.012

[39] Z. Lin, A. Jain, C. Wang, G. Fanti, and V. Sekar. 2020. Using GANs for Sharing Networked Time Series Data: Challenges, Initial Promise, and Open Questions. In *Proceedings of the ACM Internet Measurement Conference* (Virtual Event, USA) *(IMC '20)*. Association for Computing Machinery, New York, NY, USA, 464–483. https://doi.org/10.1145/3419394.3423643

[40] Z. Lin, Y. Shi, and Z. Xue. 2022. IDSGAN: Generative Adversarial Networks For Attack Generation Against Intrusion Detection. In *Advances in Knowledge Discovery and Data Mining: 26th Pacific-Asia Conference, PAKDD 2022, Chengdu, China, May 16–19, 2022, Proceedings, Part III* (Chengdu, China). Springer-Verlag, Berlin, Heidelberg, 79–91. https://doi.org/10.1007/978-3-031-05981-0_7

[41] Cecile Low-Kam, Chedy Raissi, Mehdi Kaytoue, and Jian Pei. 2013. Mining Statistically Significant Sequential Patterns. In *ICDM*. IEEE, Dallas, TX, USA, 488–497. https://doi.org/10.1109/ICDM.2013.124

[42] Ralf Korn Magnus Wiese, Robert Knobloch and Peter Kretschmer. 2020. Quant GANs: deep generation of financial time series. *Quantitative Finance* 20, 9 (2020), 1419–1440. https://doi.org/10.1080/14697688.2020.1730426 arXiv:https://doi.org/10.1080/14697688.2020.1730426

[43] L. D. Manocchio, S. Layeghy, and M. Portmann. 2021. Flowgan-synthetic network flow generation using generative adversarial networks. In *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*. IEEE, 168–176.

[44] A. Meddahi, H. Drira, and A. Meddahi. 2021. SIP-GAN: Generative Adversarial Networks for SIP traffic generation. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*. 1–6. https://doi.org/10.1109/ISNCC52172.2021.9615632

[45] Fabien Meslet-Millet, Sandrine Mouysset, and Emmanuel Chaput. 2022. NeCSTGen: An approach for realistic network traffic generation using Deep Learning. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 3108–3113. https://doi.org/10.1109/GLOBECOM48099.2022.10000731

[46] M. F. Naeem, S. J. Oh, Y. Uh, Y. Choi, and J. Yoo. 2020. Reliable fidelity and diversity metrics for generative models. In *International Conference on Machine Learning*. PMLR, 7176–7185.

[47] S. Nakayama and D. Watling. 2014. Consistent formulation of network equilibrium with stochastic flows. *Transportation Research Part B-methodological* 66 (2014), 50–69. https://doi.org/10.1016/J.TRB.2014.03.007

[48] Muhammad Haris Naveed, Umair Sajid Hashmi, Nayab Tajved, Neha Sultan, and Ali Imran. 2022. Assessing Deep Generative Models on Time Series Network Data. *IEEE Access* 10 (2022), 64601–64617. https://doi.org/10.1109/ACCESS.2022.3177906

[49] Hojjat Navidan, Parisa Fard Moshiri, Mohammad Nabati, Reza Shahbazian, Seyed Ali Ghorashi, Vahid Shah-Mansouri, and David Windridge. 2021. Generative Adversarial Networks (GANs) in networking: A comprehensive survey & evaluation. *Computer Networks* 194 (05 2021), 108149. https://doi.org/10.1016/j.comnet.2021.108149

[50] Euclides Carlos Pinto Neto, Sajjad Dadkhah, Raphael Ferreira, Alireza Zohourian, Rongxing Lu, and Ali A. Ghorbani. 2023. CICIoT2023: A Real-Time Dataset and Benchmark for Large-Scale Attacks in IoT Environment. *Sensors* 23, 13 (2023). https://doi.org/10.3390/s23135941

[51] Hao Ni, Lukasz Szpruch, Marc Sabate-Vidales, Baoren Xiao, Magnus Wiese, and Shujian Liao. 2021. Sig-Wasserstein GANs for time series generation. In *Proceedings of the Second ACM International Conference on AI in Finance*. 1–8.

[52] Ignasi Paredes-Oliva, Pere Barlet-Ros, and Xenofontas Dimitropoulos. 2013. FaRNet: Fast Recognition of High-Dimensional Patterns from Big Network Traffic Data. *Computer Networks* 57, 18 (Dec. 2013), 3897–3913. https://doi.org/10.1016/j.comnet.2013.09.017

[53] Ignasi Paredes-Oliva, Ismael Castell-Uroz, Pere Barlet-Ros, Xenofontas Dimitropoulos, and Josep Sole-Pareta. 2012. Practical Anomaly Detection Based on Classifying Frequent Traffic Patterns. In *2012 Proceedings IEEE INFOCOM Workshops*. IEEE, Orlando, FL, USA, 49–54. https://doi.org/10.1109/infcomw.2012.6193518

[54] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. 2018. Data synthesis based on generative adversarial networks. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1071–1083. https://doi.org/10.14778/3231751.3231757

[55] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. 2016. The Synthetic data vault. In *IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 399–410. https://doi.org/10.1109/DSAA.2016.49

[56] Judea Pearl. 2009. *Causality*. Cambridge university press.

[57] François Petitjean, Tao Li, Nikolaj Tatti, and Geoffrey I. Webb. 2016. Skopus: Mining Top-k Sequential Patterns under Leverage. *DAMI* 30, 5 (Sept. 2016), 1086–1111. https://doi.org/10.1007/s10618-016-0467-9

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

[59] M. Ring, D. Schlör, D. Landes, and A. Hotho. 2019. Flow-based network traffic generation using Generative Adversarial Networks. *Computers & Security* 82 (may 2019), 156–172. https://doi.org/10.1016/j.cose.2018.12.012

[60] Markus Ring, Sarah Wunderlich, Dominik Grüdl, Dieter Landes, and Andreas Hotho. 2017. Flow-based benchmark data sets for intrusion detection. In *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS)*. ACPI, 361–369.

[61] Jorma Rissanen. 1983. A Universal Prior for Integers and Estimation by Minimum Description Length. 11, 2 (1983), 416–431.

[62] A. Schoen, G. Blanc, P.-F. Gimenez, Y. Han, F. Majorczyk, and L. Mé. 2022. Towards generic quality assessment of synthetic traffic for evaluating intrusion detection systems. In *RESSI 2022-Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information*.

[63] A. Schoen, G. Blanc, P.-F. Gimenez, Y. Han, F. Majorczyk, and L. Mé. 2024. A Tale of Two Methods: Unveiling the limitations of GAN and the Rise of Bayesian Networks for Synthetic Network Traffic Generation. In *Proceedings of the 9th International Workshop on Traffic Measurements for Cybersecurity (WTMC 2024)*. in press. https://doi.org/10.1109/EuroSPW61312.2024.00036

[64] M. R. Shahid, G. Blanc, H. Jmila, Z. Zhang, and H. Debar. 2020. Generative Deep Learning for Internet of Things Network Traffic Generation. In *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 70–79. https://doi.org/10.1109/PRDC50213.2020.00018

[65] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. 2018. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In *International Conference on Information Systems Security and Privacy*. https://api.semanticscholar.org/CorpusID:4707749

[66] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. 2018. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. In *International Conference on Information Systems Security and Privacy*.

[67] Michael Stenger, Robert Leppich, Ian Foster, Samuel Kounev, and André Bauer. 2024. Evaluation is key: a survey on evaluation measures for synthetic time series. *Journal of Big Data* 11, 1 (May 2024), 66. https://doi.org/10.1186/s40537-024-00924-7

[68] Nikolaj Tatti and Jilles Vreeken. 2012. The Long and the Short of It: Summarizing Event Sequences with Serial Episodes. ACM, 462–470.

[69] Ankit Thakkar and Ritika Lohiya. 2020. A review of the advancement in intrusion detection datasets. *Procedia Computer Science* 167 (2020), 636–645.

[70] Andrea Tonon and Fabio Vandin. 2019. Permutation Strategies for Mining Significant Sequential Patterns. In *ICDM*. IEEE, Beijing, China, 1330–1335. https://doi.org/10.1109/ICDM.2019.00169

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[72] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. 2011. Krimp: Mining Itemsets that Compress. 23, 1 (2011), 169–214.

[73] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient Mining of Frequent Closed Sequences. In *ICDE*. 79–90.

[74] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling Tabular data using Conditional GAN. In *Advances in Neural Information Processing Systems*.

[75] Shengzhe Xu, Manish Marwah, Martin Arlitt, and Naren Ramakrishnan. 2021. *STAN: Synthetic Network Traffic Generation with Generative Neural Models*. 3–29. https://doi.org/10.1007/978-3-030-87839-9_1

[76] Xifeng Yan, Jiawei Han, and Ramin Afshar. 2003. CloSpan: Mining: Closed Sequential Patterns in Large Datasets. In *SDM*. SIAM, 166–177.

[77] Y. Yin, Z. Lin, M. Jin, G. Fanti, and V. Sekar. 2022. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 458–472.

[78] Zhong Zhang, Chongming Gao, Cong Xu, Rui Miao, Qinli Yang, and Junming Shao. 2020. Revisiting Representation Degeneration Problem in Language Modeling. In *Findings*. https://api.semanticscholar.org/CorpusID:226283524

[79] Pasquale Zingo and Andrew Novocin. 2021. Introducing the TSTR Metric to Improve Network Traffic GANs. In *Advances in Information and Communication*, Kohei Arai (Ed.). Springer International Publishing, Cham, 643–650.